Procedurally Generated Levels

Implemented in A Mazing Monk

s042289 Peter Magnus Falk

s042267 Philip Mygind

October 2010

Resumé

Denne rapport diskuterer fordele og ulemper ved proceduralt genererede baner, med særlig fokus på spil i *puzzle* genren. Rapporten konkluderer at løselighed, diversitet og sværhedsgrader er de hovedkrav en banegenerator skal opfylde og en række metoder til at opnå dette gennemgås.

Rapporten gennemgår yderligere en analyse, en designspecification og en implementering af en procedural banegenerator for spillet *A Mazing Monk*, samt en algoritme til at løse en given bane, til verifikation af banernes løselighed. Disse komponenter blev brugt til at udvide spillet med to nye missionstyper, for at illustrerer banegeneratores anvendelighed.

Abstract

This report discusses the advantages and disadvantages of procedural level generation in games, focusing on puzzle games in particular. Solvability, variety and difficulty ratings are found to be the key requirements for level generators and different approaches to meet these are discussed.

Additionally, the report analyzes, designs and implements a procedural level generator for the game *A Mazing Monk* and a solver for verification of the generated levels. These components are used to expand the game with two new games modes to illustrate the functionality of the level generator.

Table of Contents

List of Figures

List of Graphs

Preface

This master thesis was created as the completion of the Digital Media Engineering program at DTU Informatics in 2010. The project was supervised by Michael Rose from DTU Informatics.

The report deals with the problem of creating and utilizing procedurally generated content, and the process of implementing it into a puzzle game. The main objective of the thesis is to evaluate the advantages of using this method, and define the obstacles and problems.

The project could not have been implemented and tested without the game we used, A Mazing Monk; therefore, we would like to thank everyone from the A Mazing Monk project team for creating the game. In addition, we would like to thank DADIU for advice and practical questions regarding the thesis.

1. Introduction

Casual games are on the rise today as smart phones and web applications are getting more numerous and powerful. These games are easy to install and get started with and therefore appeal to non-hardcore gamers. The development tools for these platforms have also matured greatly making it possible for small development teams to produce quality games.

However, when it comes to content generation smaller teams are still at a disadvantage, since the more content, you need the more manpower and hours to put into it. This in turn takes away time and money from creating and fine-tuning game play mechanics. Since most casual games are not story driven, but more often in the puzzle game genre, a huge number of levels are usually required to make the user feel she is getting her money's worth. This can tie up a level designer for the complete duration of the games development phase. In smaller teams however, staff members often have multiple roles and the level designer could likely be part of the programming team as well, meaning a compromise would have to be reached between time spent on coding and developing levels.

For these types of games and this type of development team, procedural level generation could be a valuable time and money saver. If it is possible to define a set of building blocks and heuristics on how a level that is fun and valid is generated, it should be possible to define an algorithm capable of generating the levels needed for the game. However, will the levels be fun, believable, challenging enough whilst not too challenging?

It is our objective in this thesis to locate the key requirements of a level generator for puzzle games and find methods to ensure that these are met.

1.1 Motivation

Our motivation comes from a small casual game we developed at the DADIU March production 2009 called *A Mazing Monk* (1). The game is a turn based puzzle game where the user has to navigate a monk through a maze in order to pick up enough pieces of karma to unlock the exit.

Since the game was supposed to be a prototype of the final game, we were instructed to create only three levels that represented an easy, medium and hard mode of the game respectively. The game play mechanics are simple and the levels are composed from a small set of building blocks, making it a good case for this thesis. We will therefore include a chapter describing the game and its inner workings.

The game received a couple of positive reviews from various websites, but was naturally considered too short, so we have since wanted to create more levels for the game. Unfortunately, the original development team has moved on to other purposes, giving us another reason for developing an algorithm that can generate levels procedurally to give it some much need longevity.

1.2 Goals

Our goals for the thesis are the following:

1. To examine the requirements of a level generator and discuss different methods to ensure they are met.
2. To generate an algorithm that can create “random” levels with specific difficulty ratings and implement it into “A Mazing Monk”.
3. To test and analyze the result.

2. A Mazing Monk

The game was developed at a DADIU March production in 2009 with the following conditions:

- The target audience is 50 years or older.
- The game has to be a casual puzzle game containing three levels of increasing difficulty.

For this project, focus will be on making the game fun and entertaining for all ages and not only people of 50+ years of age. The following chapters will describe *A Mazing Monk* in detail and provide a summary of an interview with the game designer.

2.1 Rules

The goal of *A Mazing Monk* is to get a monk closer to Buddha hood, by completing a set of levels. The levels are completed by moving the monk around to collect pieces of Karma and finally move to the end square when enough Karma has been gathered for the exit to appear. To be able to get all the pieces of Karma and get to the exit, the monk has to puzzle his way through a maze of obstacles while avoiding enemies who will kill him on sight. The maze is set on top of a Rubik's cube that the player can manipulate by rotating the sides. The rotations are limited to a left to right rotation and a front to back rotation.

Figure 1 an *A Mazing Monk* cube

The monk can move from square to square, but only on the top side of the cube and cannot move through obstacles. The maze can be altered by rotating one of the sides, but should a path become available between the monk and the soldiers, the soldiers will kill the monk. The strategy then is to move the monk and rotate the sides in such a way that this does not happen.

When all Karma has been collected, the player must move the monk to the lower right corner of the top side, where a shining light will be shown. When the monk has moved to this square, the level ends and a new one begins.

2.2 Details about the parameters

The game contains a few distinct elements and they have various effects on the level, when changed.

Karma

The amount of Karma on the map determines how easy it will be to finish. A level with only 7 pieces (the amount needed to win the level) will be a lot harder than a level with 12 pieces of Karma for example. The placement of the Karma pieces also has influence on how hard a level is. Obviously, if all the Karma pieces were on the top of the cube collecting them and subsequently moving to the end square, would be very easy. A harder level would have all the Karma on the backside of the cube, so it will take at least three turns to get the Karma to the top side so the monk can grab it.

Obstacles

The monk needs to block the soldiers from attacking. This is done by placing obstacles between the monk and the soldiers so they cannot get to him, most commonly done by first rotating obstacles to the top face before soldiers are rotated to the top face.

The amount of obstacles and their positions have an impact on both game play and difficulty, but not as much as adding or removing soldiers.

Soldiers

The soldiers are a vital part of the puzzle. They force the player to make clever decisions to avoid creating a path for the soldiers to attack. The difficulty increases with the amount of soldiers on the cube, if there are no soldiers there are no consequences for a bad move, as it would just be a simple maze game. Placing many soldiers on the cube may make it might seem hard and intimidating, but it could also leave the player with few courses of action and therefore guide the player through the level. They would however require that the player thinks harder before every move, and therefore it feels harder.

The placement of the soldiers is also influential on the difficulty. It is harder to block soldiers in the middle square for example. In addition, there are certain configurations of soldiers that can increase the level of difficulty considerably.

2.3 Interview with game designer

To be able to create fun, challenging and entertaining levels procedurally, the game has to be analyzed. We interviewed the game designer of *A Mazing Monk* to find out what he and the level designers had thought of when they designed the levels for the prototype of *A Mazing Monk*.

Aesthetics

When the original version of *A Mazing Monk* was designed, the focus was on the visuals and how they integrated into the gameplay. The focus was not on making really challenging and re-playable levels. It was important that there was a good balance between the amount of obstacles, path squares and enemies on the cube.

It was also considered to include six different types of obstacles, four types of trees and two types of rocks, which would be chosen from at random when a level was started. This idea was discarded however because the game designer wanted complete control of the final look for the presentation of the game.

Difficulty

When the original levels were created, the difficulty was mostly adjusted by adding or removing soldiers. During the idea phase, the team would go out and test the paper prototype on different people. It revealed that many of the test subjects were intimidated and confused by the soldiers, if they were visible and in high numbers.

It took a lot of testing to perfect the three levels that made it into the prototype. The goal was to create three levels representing an easy, medium and hard level, which a newcomer to the game should be able to solve in a reasonable amount of time.

Process of creating a level

During the project, the level designers and the game designer created up to thirty different configurations a day, picked out the best ones and tested them on a target audience using a real Rubrik's cube with stickers. The best one was selected and the rest were discarded. Figuring out what did and did not work in the game and fine-tuning the game mechanics was therefore a long and arduous process.

A lot of valuable information came out of these studies on different level configurations, e.g. what makes a level difficult and how do different configurations affect the players; when do they get frustrated, when are they challenged or bored.

The levels could be created faster after the game mechanics were in place, but it was still a big task ensuring they were solvable and had the right difficulty .

3. Theory

This chapter will focus on the general aspects, usage and methods for procedurally generating levels. It will also examine how difficulty can be determined and how to ensure the levels are solvable. Utilizing this research, we are creating a strategy for implementing this in A Mazing Monk.

3.1 Procedural Content

Procedural content has been around for a long time, and was introduced due to the very limited memory available in the first computer games. It was not possible to load a full level into memory and it used up too much space to distribute e.g. a database with the game. Instead, the games generated the levels on the fly through procedural algorithms.

The first usages of this were to generate dungeons for role-playing games (RPG); one of the famous games doing this was Rogue (2). It introduced an algorithm for generating new dungeons and all of the contents in these dungeons. The items and monsters would increase in power as you progressed through the game. However, the game play was not affected much by the procedural dungeons; the goal was always the same.

When memory is no longer a limiting factor when creating a game the focus shifts to creating levels and worlds by hand. This will generate levels that have a much higher degree of detail, however do not offer the variety as procedural content. To add variation, procedural content generation has shifted its use to props, i.e. simple building blocks. Having the same prop used many times all over can however make a game feel dull and lifeless. So techniques have been invented to generate the props procedurally as well. Some examples are vegetation such as grass and trees (3).

Some games, however, do not fall into this category, but instead benefit from having the level procedurally generated. Civilization 4 is a game that utilizes very advanced level generation techniques. It generates its levels based on a large amount of changeable parameters, allowing for the creation of the usual random worlds as well as mirrored levels for competitive multiplayer games. The parameters allow the player to select how in general terms the level should look, and the game procedurally generates the level based on the input.

In the more experimental genre of procedural generated content, there is .kkrieger (4), which generates all of the assets used in the game, however not randomly but procedurally. This makes the game require a huge amount of processing power, but the game is only 96 kilobytes big. Naturally, the advantage is that it is much easier to distribute a game of this size, for example, it can easily fit on a cell phone. However, the drawback is that it relies so heavily on processing power that it would not be able to run on any mobile device.

Today an increasing trend is casual games, which are easy games to get into. Many of these are platform or puzzle games; these make ideal candidates for using procedural generated content. The reason is that they often have quite simple game mechanics that can be mimicked by a generator, the other reason is that they are quite often built in small teams, who might not have the resources to hire a specialist to make 1,000 levels manually.

3.2 Randomness and pitfalls

In this chapter, we will discuss the potential pitfalls in creating a procedural level generator.

Solvability

The first thing to consider when generating levels procedurally is, are the levels solvable? If it is a maze type game, will there always be a path through the maze? If it is a platform game, is it possible for the player character to jump to all the necessary platforms? If it is a puzzle game, is there always at least one solution? Obviously, unsolvable levels are a nuisance that will annoy the player, even if it only occurs rarely.

Verifiability is another thing to consider; is it possible to do it on the fly with respect to time and memory usage? It may annoy the player to have to wait a long time for the level generation and verification. If it uses too much memory, it may not run on the desired platform. The limited power supply of smart phones should also be considered for games that are supposed to run on such a platform.

If it is not practical to generate the levels on the fly, there is also the option of pre-generating them and storing them to a file. This, of course, would mean that the game is no longer “infinite”, but has a limited amount of levels. Creating a huge number of levels might still give the player the illusion of playing a never-ending game, but then disk-space becomes a factor. If the game is going to be run on a smart-phone there might not be enough space on its internal hard drive to store the number of levels needed. If the game is going to be run in a web browser, the time it takes to download the levels from the internet, might put the player off. In these cases it is worth examining if the levels can be compressed, by for instance condensing it to a seed (a number or a short string) from which the level can be recreated. This might be possible if a random number generator is used to create the levels.

Playability

Assuming that the levels are solvable; can it also be assumed that the levels are playable? By this, we mean can they be solved by the player.

The computer might be able to solve a given level, but is the same information available to the player as the level generator or solver? In a maze type game for example, the game may hide part of the map from the player in a "fog of war". However, if the algorithm that generated or verified the level can see the area, it has an advantage to the user. Therefore, it is important for the verification process to consider the player's perspective.

Is the solution logical to the user? In a puzzle game where the player's actions are irreversible, a level generator that does not take playability into account, might produce a level that is only solvable by the player through chance, for instance if she throws 10 "lucky punches". Meaning she guesses the correct moves ten times in a row, without having any way of predicting whether or not it is the correct one. In this case, solving the level becomes either completely luck-based or it must be solved through trial-and-error by replaying the same level repeatedly. In any case, these kinds of solutions usually do not make a very fun game. Therefore, the generator should construct the levels in a way that they can be solved logically.

Figure 2 Example of fog of war, from Civilization IV

It is also important to consider the player's entry level. Different players have different skills, some are better than others, and if you want to cater to as many types of players as possible, the level generator should be able to create levels of varying difficulties. When designing levels manually, a given levels difficulty rating can be established by testing it on multiple users and analyzing their feedback.

This is not possible with a procedurally generated level, as the game generates different levels each time you play it, so other ways of measuring the difficulty should be established. This could be done by defining a set of measurement heuristics such as "the number of enemies", "the position of the enemies", "resource available to the user vs. resources available to the enemies" etc, and finally an algorithm that calculates some weighted average of the results. In the end, the only way to determine whether or not the calculated difficulty rating hits the mark, is to perform user tests, and this time have the users test whether a batch of levels fit the given difficulty rating.

Realism

Depending on the game type, realism can also be a factor that must be taken into account. Clearly, in games where the visuals are supposed to represent something world-like to the player, a forest of trees for instance, care must be taken that the level components are placed in a natural manner to not break the illusion for the player. If contents are scattered in a completely random fashion, the result may look artificial and unappealing. In this sense, the levels can be generated too randomly. Even for a puzzle game where the visuals are not supposed to be realistic, the flow between levels can seem "unrealistic" if it jumps or falls sharply in difficulty, because the progression does not seem natural.

Conversely, as a level can seem too random, the opposite can also hold true. If the levels look too much alike or a particular level has too many similar elements, the game can easily become boring and repetitive to the player. Therefore, it is important that a level generator can find a balance between a linear or logical progression between the levels and still have diverse and non-repetitive levels.

We will in the following chapters look at different methods to avoid these pitfalls.

3.3 Math Puzzles

Easy to understand math puzzle can be used to be able to understand some of the problems in creating levels based on randomness. In these games, the goal is to solve it, but it can be difficult to create a completely random level that is also solvable.

15 puzzle

The 15 Puzzle was invented in the late 19th century and is a small game where the player has to align the pieces by moving one piece to the empty square. This game is an excellent example of how randomly generated levels in a game can provide unsolvable levels. In the figure there is a 4x4 game field and all the numbers in correct order from 1 to 15, and the lower right corner being blank:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

If you scramble this by hand, it will always be solvable. However, if 14 and 15 are swapped the puzzle will no longer be solvable.

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Many other games have similar problems if the level is generated completely randomly. Therefore, a basic starting point for generating a level is by doing it in reverse. Meaning that you have the final solution and then working backwards to the starting position. In this way, it will always be solvable. Proving that the figure above is not solvable is done using group (4).

Being able to determine whether a specific level is mathematical solvable or not is a great help when creating random levels. It will then be possible to just randomly create levels and discard the levels that are not solvable.

Rubik's Cube

The Rubik's Cube is another well-known puzzle, which has the same properties as the 15 puzzle, since it is possible to create a cube that is not solvable and it can be determined through the same mathematical principles as the 15 puzzle.

To create an unsolvable Rubik's Cube, take a finished cube and switch around two corners. It will no longer be possible to solve the cube.

Another interesting point is the amount of moves it requires to solve the game, more precisely the least amount of moves needed. This problem is NP-hard and a lot of research has been done in finding this number for any given Rubik's Cube.

Scientists have dubbed this number "God's number", and it has recently been proven to be 20 moves (5). However, this type of solution is hard to implement in a game, due to the limited resources available. For example, there are 43,252,003,274,489,856,000 different positions of a Rubik's cube, and solving each of these on a normal computer would take an enormous amount of time.

3.4 Solvability in general

A level is solvable if it is possible for the player to get from the starting point of the level to the finishing point, by carrying out a sequence of actions. This sequence can be defined as the path through the level and the level generator has to ensure that the level contains at least one path.

It would be very difficult to find a model that ensures solvability for all game types, so this chapter will focus on solvability for games such as A Mazing Monk for which the following assertions are true:

1. A level is constructed from a set of building blocks that can be placed in a finite set of positions.
2. Different levels are different permutations of these building blocks, so the problem becomes how randomly picking a sub-set of building blocks and placing them in such a way that the level is solvable, given the actions available to the player.
3. An action is an alteration of the building blocks, such as creating a new building block in a specific position or changing/reordering existing ones. If the game has a playable character, it can also mean moving this from one position to another.
4. The actions can be governed by rules that determine if an action is legal in a given context. A rule could for example require that certain objectives must be accomplished before an action is legal.

The forward approach

The simplest way of generating a solvable level that has at least one path through it, is to simply construct the path incrementally and then add any additional misleading paths afterwards. This could be called the forward approach, as it builds the path from start to finish.

As an example, consider a maze set in a grid of cells, which can be either a pathway or an obstacle. These are the two building blocks of the game. A player starts in a given cell and can move to any adjacent cell, in an up, down, left and right direction, if it contains a pathway. The objective of the game is for the player to get to the end cell.

Creating a solvable maze using the forward approach can be accomplished by first marking all cells in the grid as obstacles. Then a random cell is picked as the starting point and marked as a pathway. From this cell, a random path is built by iteratively selecting and moving to a new cell selected randomly from the adjacent cells and marking them as pathways. This is done a certain number of times and the final cell is used as the end point. As a final touch up, additional misleading pathways are created in a similar fashion at random points along the path. This process can be visualized as seen below:

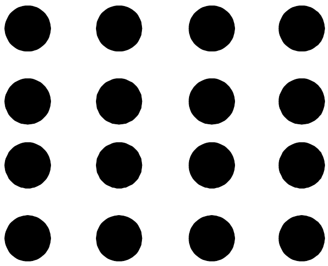


Figure 3 Step 1: Fill the maze with obstacles.

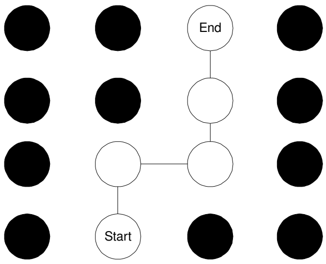


Figure 4 Step 2: Build random path and mark the first and last cell as start and end point.

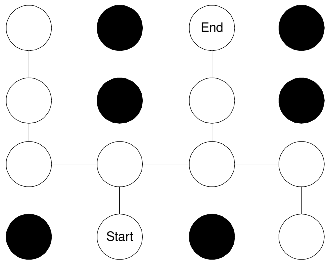


Figure 5 Step 3: Add misleading pathways.

The forward approach is useful for levels that have simple or no requirements for the end result, apart from the fact that a path from start to finish exists. It is also important that the next step in the iterative path building process is simple to calculate. In this case, selecting an adjacent cell is a simple calculation.

The backward approach

In some circumstances, it may be beneficial to work the other way around, starting with the end point and working backwards towards a starting point. This approach can be aptly named the backwards approach and is useful if a level must fulfill certain requirements. Using the forward approach a generator would build the level incrementally and in the process try to fulfill the requirements, which may or may not be possible. In contrast, the backward approach would first ensure that the requirements are met and then assemble the rest of the level.

As an example, requirements can be added to the maze described earlier. It must now contain three rooms of a certain size, which the player has to visit. Additionally, these rooms only have one entrance and they must be placed at a certain distance from each other. The forward approach would build the maze incrementally as described earlier and in the process try to fit in the rooms at random points along the path. If the grid is small enough it may lead to a situation, where there is not enough space to place all the rooms according to the requirements. The backwards approach would place the rooms in the grid first, ensuring they meet the requirements and then connect the maze to them.

Using the game 15 Puzzle as another example, it was described in 3.3 Math Puzzles that simply placing the numbers 1 to 15 in random cells may result in an unsolvable puzzle. A better approach is to start with a solved grid, i.e. the numbers placed in ascending order from the top left corner to the bottom right corner, and repeatedly swap the empty cell with a randomly picked adjacent cell for a certain number of iterations. The resulting scrambled grid is the starting point of the level, achieved by working backwards from the solved grid.

Adding Backtracking

The forward and backward approaches are both prone to dead ends, which are situations where the level generator cannot complete the level, because a series of unfortunate decisions have left it with no further options to choose from. A technique called Backtracking can help avoid this scenario.

Backtracking is essentially a way of allowing an algorithm to take a step back and try another approach if it runs out of options. It works by maintaining a decision tree, where the node stores the action taken to get to the current state, the decision taken immediately before it as the parent node, and a list of options now available in the form of its children.

When an algorithm picks a path that leads to a dead end, it is now able to traverse back through the decision tree and pick a new direction, by following another option from the parent node or other nodes in the tree containing unexplored options. The node containing the option leading to the dead end is removed from the tree and if the parent node has no more children, it is removed etc. until all dead branches have been pruned from the tree.

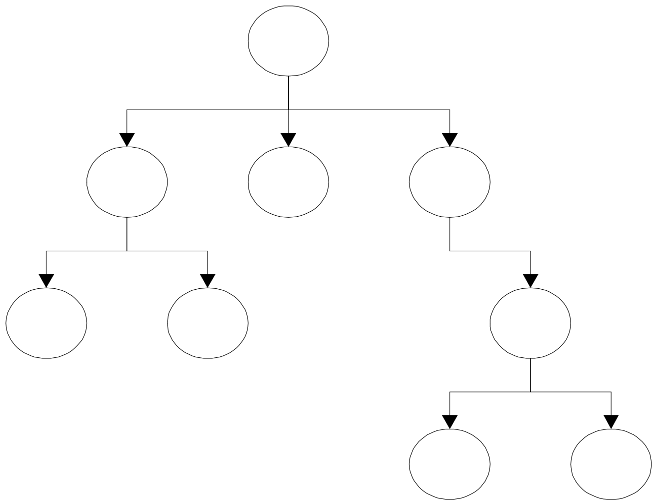


Figure 6 Decision tree used for backtracking

As an example, consider the game of Sudoku; here the goal of the level generator is to fill in a 9 x 9 grid with the numbers 1 to 9, so they conform to rules of Sudoku and finally hide a certain number of the cells. A naive solution to filling the grid using the forward approach would look like this:

1. Pick a random empty cell
2. Calculate the numbers that can legally be placed in the cell and place a random number from that group in the cell.
3. End the program if the grid is filled otherwise go back to 1.

Anyone who has ever tried to solve a game of Sudoku will know that this approach would probably lead to an unsolvable grid very quickly. To improve the algorithm a backtracking structure is added. Assuming the decision nodes stores the coordinates of a specific cell and a number to place in it, the improved formula looks like this:

1. Create a root node and set it as the current node.
2. From the current node, check to see if the grid is filled. If so, end the program.
3. Otherwise, create a child node for each legal number of each empty cell in the grid.
4. If the current node has no children, destroy it and set the parent node as the current node. Repeat until the current node has children.
5. From the current node, set a random child node as the current and go back to line 2.

The tree structure implemented in the above example ensures that the algorithm does not stop at dead ends it simply tries another approach. While an improvement to the first algorithm, it is still far from being perfect, since it is in a sense brute forcing its way through the Sudoku, trying every possible option until a solution is found. It would require a large amount of memory to hold the sizable tree structure; the root node alone has 9 x 9 x 9 = 729 children. It would also require many computations since it will most likely back track quite a few times before it converges on a solution. It can be optimized, however, as many of the branches will, if the decisions are resolved, contain repeating grids for example. The only difference between them is the order in which the numbers are placed on the grid, which is irrelevant. Therefore, child nodes should only be created if their resulting grid does not already exist somewhere in the tree. This information can be stored in a lookup table containing the grids produced so far and would reduce the number of options to try. The algorithm as described performs a depth-first search on the decision tree. However, other selection techniques can be used as well, such as picking the next node to explore using a best-fit heuristic, i.e. the node with the highest probability of success.

Using backtracking for level generation is not only applicable to Sudoku, but also any other combinatorial based games with complex rules. It can be a useful supplement as it allows algorithms to undo previous actions and avoid dead ends. It may be necessary to implement optimizations, however, which can differ depending on the game type, as it can easily grow to unpractical sizes in terms of memory usage and time consumption. In fact, Sudoku generators commonly combine backtracking with other graph algorithms such as graph coloring (17), an algorithm used to assign a fixed set of colors to each vertex in a graph so that two connected vertices never share the same color.

Solvability as a graph

By expanding on the backtracking example, solvability can be viewed as a state graph, consisting of a Start state, an End state and a set of intermediary states. The Start state represents a blank canvas, i.e. a level without any building blocks and the End state represents the final solvable level with all the randomly picked and placed building blocks. The intermediary states represent the building blocks available to the generator in any position possible.

In a game of Sudoku for example there would be 9 x 9 x 9 = 729 intermediary states each representing a number in a respective cell. Creating a solvable Sudoku grid corresponds to connecting a path through 81 intermediary states from the graph, so that placing the number of the path states in the cell they represent, results in a legal Sudoku grid.

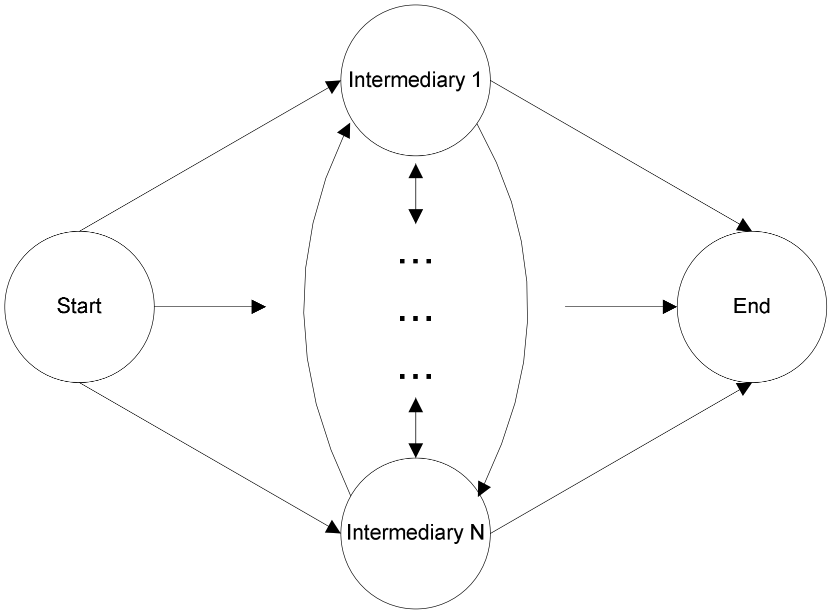


Figure 7 Level state diagram

A connection between two intermediary states corresponds to an action in the game that changes its state. Using Sudoku as an example, a transition to a state *s1* performs the action of placing the number of *s1* in the cell it represents. The transition can be conditioned by certain objectives that must be accomplished before the transition is allowed. In this example, placing a number in a cell has to follow the standard Sudoku rules.

To create a solvable level, the generator has to connect a path from the Start state through the intermediary states and to the End state. There may be more than one way of getting from start to finish, but there should always be at least one solution to the problem. Each time a connection is made between two states, the state of the level changes, so in a sense the method solves the level as it builds it.

Viewing solvability as a graph can be useful as it enables the use of existing graph algorithms to find a solution. For example, algorithms that can be used to limit the amount of candidates to try at each decision point or algorithms that can pick the best candidate to proceed with. However, it does require that the intermediary states can be calculated easily.

Adding a solver

Some games may be easier to solve than create. The requirements and rules for the levels may be so complex that creating levels and ensuring they are solvable as they are built, requires an unpractical amount of time and memory. In these cases, it can be useful to create a solver that works in conjunction with the generator. Then the levels can be built using best-guess heuristics that will for the most part generate solvable levels, and use the solver to discard the ones that fail.

In case the of Sudoku, a long list of solving techniques have been discovered (18) and a common technique to generate Sudoku grids is to first fill in a number of random cells, with random legal numbers, and then use the solving techniques to fill in the rest of the grid.

3.5 Realism in procedural level generation

Realism in levels encompasses how to place level components in a realistic manner and how to build levels that feel natural. The need for this, of course, varies greatly from game to game depending on the universe they are set in.

Commonly, the problem is solved by examining the nature of the components, deriving some simplified heuristic of how they behave in real life, and then implementing these into the level generating algorithm(s). The goal is not to make the perfect simulation of real-life, but to make the objects appear natural and recognizable to the player in the given context, which can often be accomplished by simple tricks.

The visuals in *A Mazing Monk* are not supposed to look realistic. At present, the game has two types of obstacles, trees and rocks, which does not allow for much variety. Future versions of the game could however expand the size of the cube and add a more diverse set of obstacles. This would make the visuals more complex and different ways to control the types of components used becomes more important, as realism becomes a factor.

Creating a forest

Mick West, co-founder of Neversoft, has written an article (4) on the matter in which he tries to solve the problem of scattering trees in a random but seemingly realistic manner.

The article makes a distinction between two methodologies in procedural content generation, teleological and ontogenetic, and gives examples of both. Teleological is the method of accurately modeling the physics of the environment. The ontogenetic method in contrast looks at the desired result and attempts to reproduce this through ad hoc algorithms.

Teleological

The first example is of the teleological approach. The article starts by concluding that positioning the trees in a purely random fashion results in an unrealistic looking forest as illustrated below:

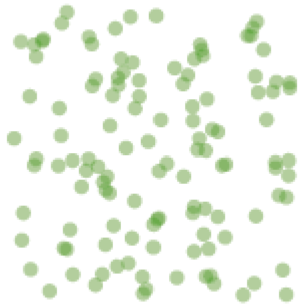


Figure 8 Trees placed randomly (from Gamasutra)

The trees are represented as green dots and are scattered randomly on a plane. The result does not look very realistic as some trees are almost standing on top of each other, while others are standing alone with a lot of white space around them. The article then goes on investigating how trees scatter in real life and derives two heuristics based on the findings. After a few iterations, the model is defined as the following:

- Trees are spawned from seeds from a parent tree, placing them close to that.
- The closer a tree is to other trees the less likely it is to succeed (grow) and if its probability of success falls beneath a certain threshold it is not planted.

Following these heuristics, the pseudo-algorithm becomes:

1. Pick a random position, determine if a tree can succeed at that point and if so "plant" the tree.
2. From that tree, spawn a random number of children around it, in random positions within a certain distance, and remove the unsuccessful ones.
3. Either spawn new trees from the children trees or go back to one. Repeat until enough trees have been planted.

The outcome looks something like this:

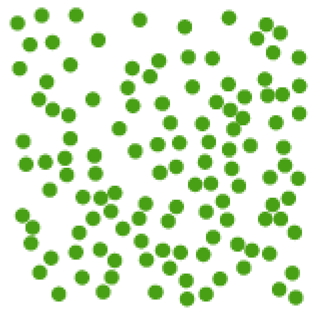


Figure 9 Trees scattered randomly but in a controlled manner (from Gamasutra)

Whether the above picture looks more like a forest than the first example is, of course, a matter of taste. However, a simple way was found to circumvent some of the most unrealistic traits of the purely random model. Further heuristics could be added to improve the outcome, such as trees like water and would tend to flourish better near lakes and rivers; they do not like to grow in shade or on rocky soil etc.

Ontogenetic

The disadvantage of the previous algorithm is that it can take an unknown amount of tries, before all the trees have been planted; meaning it could potentially be very expensive computationally. It is also harder to control the outcome and may take many retries and verifications before it fits the level. Therefore, the article tries another approach and redefines the problem; instead of modeling the behavior of trees, it determines what the resulting forest should look like. Now the model is defined as follows:

- The forest should contain x amount of trees that are distributed evenly to avoid clusters, but with random x-and-y offsets, so that they are not standing in a row.

Compared to the first algorithm this is clearly a relaxation of the "realistic" nature of forests, but it is more focused on achieving a forest that is suitable for the given level.

An example of an algorithm that achieves this is given, in which the desired amount of trees is placed in an evenly spaced grid, with all neighboring trees linked together. For a fixed number of iterations the algorithm then moves each tree individually in a random x-and-y direction, ensuring that they keep a minimum distance to each of their neighbors. The result is a forest of trees that look randomly scattered but without any clustering or overlapping of the trees.

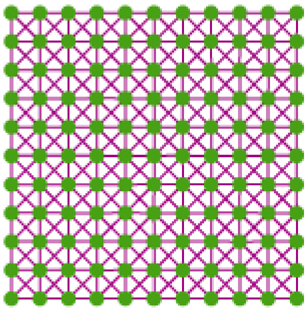


Figure 10 Trees evenly placed in a grid (from Gamasutra)

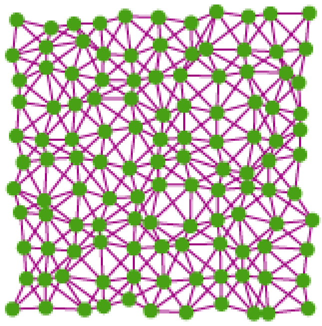


Figure 11 Same trees after a number of iterations (from Gamasutra)

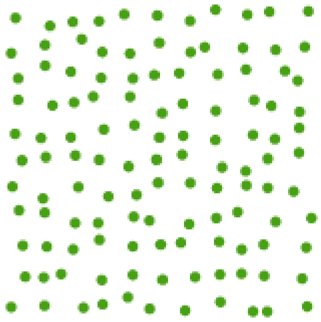


Figure 12 End result (from Gamasutra)

Again, realism is in the eye of the beholder and further testing and tweaking may be needed to produce a result that is realistic and suitable enough to the needs.

Common algorithms

Inspired by the website pcg.wikidot.com (5) a list of common algorithms used in teleological and ontogenetic methods will be presented here and a short description of each given:

Teleological methods

Artificial Life: Artificial Life (6) is not a single algorithm but an umbrella term for algorithms that attempt to reproduce the physical behavior of biological systems. It is related to agent-based systems (7), which perceive components as autonomous agents that interact in certain ways with their environment and other agents. In these types of algorithms, the life cycle of each object is simulated, meaning that they have an age and a time limit and can often reproduce (spawning new objects). In level generation, Artificial Life can be applied by simulating the "life" of certain level objects up to a point in time and using the current state of the objects for the result. Even terrain can be simulated (8) in this fashion as it erodes and changes over time.

Cellular automatons: A cellular automaton (9) is a collection of cells, usually organized in a grid, that are each connected to their neighbors. In a grid, this would mean the adjacent cells. Each cell has a state and a shared rule for updating this based on its current state and that of its neighbors. An update function applies the update rule to each of the cells in the collection, or a particular subset of them. This progression of cell states can be placed on a timeline, where the initial state of the cell equals time = 0, and each iteration equals time = 1 ... n. If the cells can expire after a certain time span, the method can be used as an

Artificial Life algorithm. Likewise, the algorithm can be applied to level generation by running it up to a specific or random point in time and using the cell states as part of the content generating algorithm.

Genetic Algorithm: The Genetic Algorithm (15) (16) works much as the first scattering trees example. An initial population of objects is either randomly created and through an iterative process each member of the population is eliminated, if it fails a fitness test. If not, it may potentially be selected for reproduction, creating new objects. In the scattered trees example, the fitness test was the distance to the nearest neighbor. Like the Cellular Automatons, this evolutionary algorithm can be run up to a certain point in time and the final objects used in the content generating algorithm.

Markov Chains: The Markov Chains (17) algorithm constructs a chain of states through iterations, where each state is dependent on the previous state, and only that. At each iteration step, a new state is appended to the chain, by selecting the most probable state to follow the current. If more than one state shares the same probability, a random state is picked from that group. This ensures a random but controlled generation of states. The transition probability from one state to another can be stored in a table. An example of such a table could be given for an algorithm that generates tiles for a map, with five different tile types to choose between. The table could look something like this:

From	To	Probability
Rock	Grass	0.8
Rock	Dirt	0.8
Rock	Tree	0.1
Rock	Bush	0.1

This table, of course, only covers the transition from one tile type (rock) to the other four (grass, dirt, tree, bush) but should illustrate the principle. Another example is the Drunkard's Walk algorithm that builds a Markov Chain using positions on a grid as the state. At each step, the algorithm jumps to a new position in one of four directions (up, down, left, right) with equal probability. The resulting chain after a given number of iterations could be used to carve out paths for a dungeon or construct the roads in a city for example.

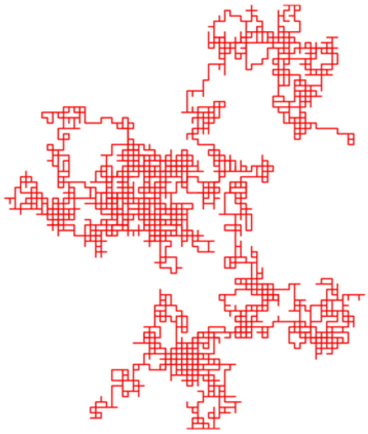


Figure 13 Example of Drunkard's Walk (from Wikipedia)

Ontogenetic methods

Fractals: Fractals (10) are functions, which produce fragmented geometrical structures, or images that have infinite complexity and exhibit a behavior of self-similarity. Infinite complexity means that you can zoom in at any point and get more details. Self-similarity means that the overall structure repeats itself to a varying degree. The Mandelbrot set (11) is a well-known example of this.

The interesting thing about fractals is that they can be found everywhere in nature. Looking at the outline of a mountain range would show that it has a pattern of big variations (peaks and valleys) and small variations (cliffs and rocks). Zooming in on the mountain range through a binocular would repeat this pattern. The same can be observed in trees. Looking at them from afar, you see the trunk that has branches protruding from it. Zoom in on a branch and it may have other smaller branches sticking out from it and so on. In a sense, we are used to looking at fractals in nature and mimicking this behavior can help creating realistically looking levels. For example by placing trees in a fractal pattern or by outlining a complex coastline and so on.

Perlin noise: Perlin noise (12) is a fractal technique used to generate a continuously random signal of an arbitrary dimension. It requires one or more noise functions and an interpolation function. A noise function is a seeded random number generator, which maps a point in space to a real number between -1 and 1. It will always return the same value for a give set of parameters, so the signal is reproducible. The parameters for the noise functions can be x and y coordinates, for a two dimensional signal, x, y and z for a three dimensional signal and so on. A time parameter can also be added to make the signal time dependant. The actual noise is then generated by summing and interpolating the noise functions for each discrete point in space.

By combining noise signals with different frequencies and amplitudes, a great variety of signals can be created. For example, a signal resembling a mountain range could be constructed by adding a noise signal with great variation (big amplitude, low frequency) with a signal containing small variation (small amplitude, high frequency) , giving it big peaks and valleys along with smaller changes in between.

Perlin noise is useful for many purposes in a procedural level generator. A two dimensional Perlin noise signal can be turned into a texture, for variety in the level components. The same texture may be used as a height map for a 3D terrain or to create realistically looking cloud textures and so on. A one-dimensional signal can control the movement of an object, making it appear to behave in a more natural and less mechanical way.

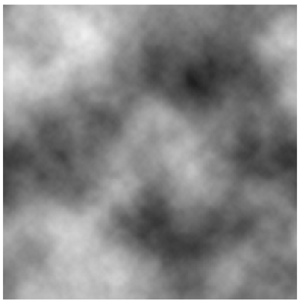


Figure 14 Example of 2D Perlin noise texture

L-Systems: Another fractal technique is L-Systems, mostly used for generating realistic looking plants and trees. To use it one defines a language containing a set of words and a set of rules for combining these. A word is chosen as the initial state and from that, a sequence of words is appended iteratively. As an example, consider the following language:

Words: A, B
Rules: A → AB, B →A

This defines two words “A” and “B”, and two rules which state that after the word “A” an “AB” string must follow and after the word “B” an “A” string must follow. If the initial state is “A”, running the algorithm through four iterations will result in:

Iteration 1: A
Iteration 2: AAB
Iteration 3: AABA
Iteration 4: AABAAB

Rather than concatenating a string, one could construct a tree from the language, revealing its use in plant generation. The above example results in the following after four iterations:

```
              A
          /       \
          A       B
      /       \       \
      A       B       A
   /     \       \   |   \
   A     B       A   A   B
```

The rules can further be expanded by including the angle from which a new node should be positioned relative to its parent node, to create more interesting structures. For example if “+” means 10 degrees and “-“ means 270 degrees, the rule A → A-B+ means: Create an A and a B and place A at a 10 degree angle from the parent and B at a 270 degrees from the parent. This technique can create a very diverse set of fractals as shown below:

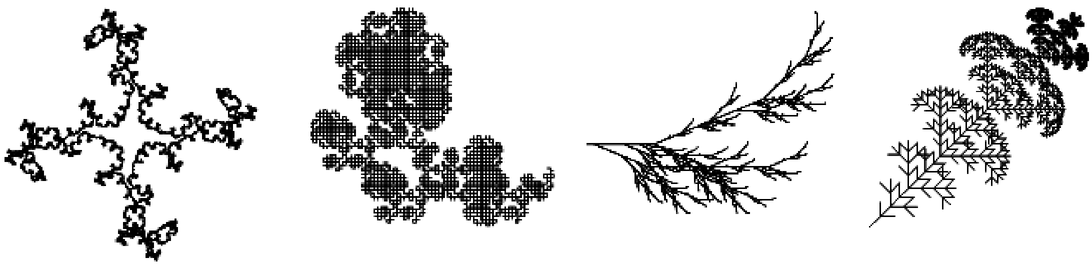


Figure 15 Example of L-System fractals.

The above fractals were all created using the L-System applet by Robert Jessop (13). To add randomness to the process, each node could be assigned a specific probability of spawning new children. This would also ensure that the resulting pattern is not completely symmetrical.

Finally, it should be noted that L-Systems are not limited to generating plants or connected graphs. Instead of viewing the pattern as a tree, one could simply use the positions of the nodes as locations for level objects and thereby scattering them in a fractal pattern.

While the above-mentioned algorithms were categorized as being either teleological or ontogenetic, it should be noted that many of them could be used in both approaches. The second algorithm for scattering trees bears a resemblance to Cellular Automatons for example. The maze created by the Markov Chain has a degree of self-similarity and it could be used to generate fractal patterns. Often it is the defined heuristics and rules that determine whether or not a bottom-up (teleological) or top-down (ontogenetic) approach is being used. We chose evolutionary algorithms for the teleological examples because they illustrate nicely how to “grow” a level over time and harvest the result. Conversely, fractal algorithms were chosen as ontogenetic examples since they produce naturally looking patterns.

When deciding which approach to choose, it should be considered whether it is possible to define the heuristics needed for the bottom-up approach in a sufficiently accurate yet simple way. This should be contrasted with how easy it would be to provide a coarse but adequate model for the top-down approach. In general, it seems that teleological methods produce the “prettiest” results and ontogenetic methods produce the fastest results.

3.6 Mapping a complex problem to a simpler model

The tree scattering examples also illustrate a third method applicable to simplifying the process of generating levels; namely that of mapping a complex issue to a simpler model. In this case, a forest is reduced to a collection of dots on a plane. The forest could be represented as three-dimensional objects in the final game, so by reducing them to dots the problem of placing them realistically becomes more comprehensible and potentially reduces the amount of computations needed as well.

Another example is the game Pick15 (14) where two players take turn in picking a number from 1 to 9. Each number can only be picked once and the object of the game is to be the first player that has three numbers that add up to 15. Picking the next number that helps you getting closer to 15 while ensuring the other player does not achieve the same in her next move, is not trivial task for many players. However, it can be shown that the numbers 1 to 9 can be placed in a 3x3 matrix in such a way that the numbers in each column and row and both diagonals add up to 15. This changes the scenario to a game of tic-tac-toe, which is probably easier to understand and solve for most people:

8	1	6
3	5	7
4	9	2

This illustrates, of course, also that mapping problems to simpler models is not only useful for enhancing realism in a game, but to simplify many programming tasks in general.

A third example is height maps used to generate 3D landscapes. Mapping the landscape to a 2D texture ensures that a continuous landscape is created and speeds up calculations such as determining bounding boxes and the height at a given position. It also makes it very simple to create the landscape, since an artist can simply draw it in her chosen graphics software.

The original game of A Mazing Monk takes places on a three dimensional cube, with three of its sides out of view. Mapping the cube to a 2D model makes it easier to visualize the cube, as all sides are visible.

3.7 Difficulty

What is

It is not always easy to create something truly difficult, and to understand what can be done when making the difficulty of a game, and what works when the games are supposed to have different difficulty levels.

Real

When the difficulty is real, it means that the challenge is real. This is different from the other ways of creating difficulty discussed. What the player sees as a *real* challenge is when there are just more enemies, less hiding places, and the player actually has to make the right decisions to win.

Another example is if it is a puzzle game, or some other game that can be solved purely by using math. To heighten the difficulty in such a game, the equations the player has to solve will be of greater difficulty e.g. withholding three of the variables in the equation and letting the players guess is not a part of real difficulty. This would be considered luck.

Cheating

There are different approaches to achieve a hard difficulty level; one is by letting the computer cheat, either by giving the computer massive bonuses or inhuman abilities. However, the player might get frustrated if he thinks the computer is playing straight up but is unable to beat it. In *Street Fighter IV,* the final boss encounter is Seth, who has been engineered in such a way, that he is immensely hard because he can do all special moves of all the characters, and does it in such an effective way.

When the computer is cheating, it can really frustrate the players, although some players might find it more interesting. Another example is the Civilization games, where on the lower difficulties you get more bonuses up until normal difficulty, hereafter the computer gets the bonuses and the player might even get a penalty.

Luck-based

A more relevant way of creating difficulty is by creating luck-based problems for the player. These might not be intentional and will most likely frustrate the player e.g. if the game was supposed to be purely mathematical or skill based.

An example of a game where this is an actual problem is the Mine Sweeper game that comes with MS Windows. When playing the game the player might end up with two squares left, but there is no way to figure out which one is the bomb.

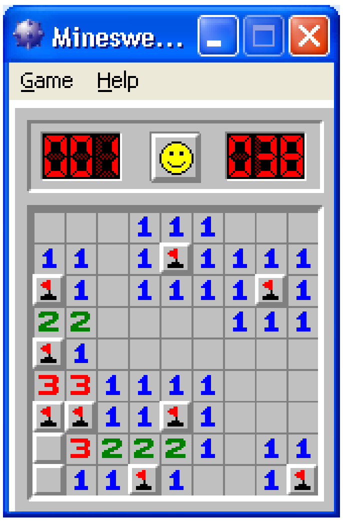


Figure 16 The problem in minesweeper that makes the player guess

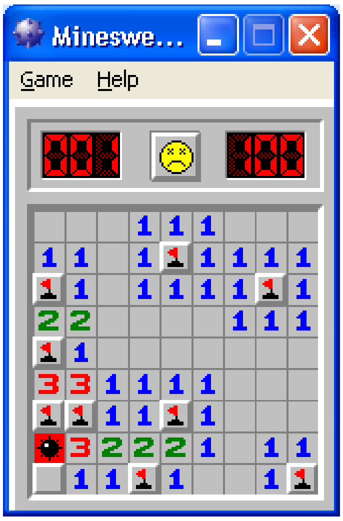


Figure 17 The player takes a guess and loses

Psychologically

A different way of creating a harder level is if it psychologically challenges the player, by intimidating or frightening him. A boss fight requires a big monster that will initially scare the player, and might make the player panic.

Another psychological aspect is time. A countdown can stress the player and forces the player to make rational decisions.

Difficulty used in A Mazing Monk

In A Mazing Monk, there are design decisions that increase the difficulty of the game. One is that only half of the sides are displayed and thereby limiting the moves only allowing turning a side in the direction so the player can see what happens on the top side. This design was to force the player to explore the small world of the cube. However, it can be rather frustrating that the player might be forced to guess what is on the hidden sides, but this is part of the gameplay.

New players are hesitant on rotating the sides with the soldiers on. However as the game progresses more and more soldiers are added, the player will be forced to rotate the sides with soldiers on and learn, when they can attack the Monk and when the Monk will be safe.

A timer is not used in A Mazing Monk as it is supposed to be relaxing and something you can easily put away for a couple of minutes and then return to it. However, implementing it and utilizing as both a feature of performance and difficulty is a possibility.

Adaptive Difficulty

Different difficulty levels are a very important part of games, but traditionally the difficulty is statically increased. Either it could be by increasing the sides of a Rubik's cube from 3x3x3 to 4x4x4; however, there are disadvantages of increasing the difficulty in this way. One is that it could scare away new players; they might not think they are able to beat the harder difficulty. The same happens for the easier difficulties, where a player might feel degraded if he has to choose *Very Easy*.

How these work is for instance by increasing the puzzle size as mentioned, increasing the amount of enemies, or decreasing the attributes of the player e.g. ammo. It mostly depends on the game how it is tweaked to raise or lower difficulty. However, in some cases, the steps might be too high, sometimes too small; they never really benefit the player or take into account that there should have been a difficulty between Normal and Hard for a certain player. That player feels that Normal is too easy but Hard is too hard, this might make the player stop playing the game.

In adaptive difficulty, the game evaluates the player's performance and then balances the game accordingly. An example is that the player is performing really badly, and then the game will make the enemies die faster. Using this technique the player no longer has to choose which difficulty level to play; the game will automatically adjust and make the game fit the player. There have been numerous different implementations of this technique, some only changes a small element of the game other the completely game play experience.

Another example of dynamically changing the game play is Left 4 Dead 2 (18) where the game changes pace, depending on what has happened, how much ammo the players have, what has happened and much more. This greatly increases the re-playability, and makes it fun for everyone to play.

The Lego Indiana Jones and Batman games both feature adaptive difficulty, where they change some basic variables based on how well you play. Not the entire pacing and game play are changed as with L4D2, but only how many studs you lose when you die.

The performance of the player can be hard to measure, and thereby adjusting the level of difficulty to fit the play style and skill level. Therefore, it is also very time consuming to implement this technique unless it only factors in very small game play elements.

In a game where the level is generated procedurally, it is more practical to implement adaptive difficulty since the level is generated with a specific difficulty in mind. The challenge here is to measure the performance of the player, so the next level generated fits.

3.8 Creating a solvable A Mazing Monk

Having looked at solvability, difficulty and different approaches to fill a level with content we will now investigate how to construct a level generator for *A Mazing Monk*.

Random

The focus of this project is to generate levels that are on par with level designer created ones. To start out we will look at generating a purely random level, to see some of the problems. This also gives a good idea of what is important when generating a new level.

Even a cube that is randomly created has to obey some rules. Firstly, the monk always has to start in the upper left corner on the top side of the cube, and soldiers should not attack the monk before the player has actually made a move. Using these rules a random level is created; meaning everything else is randomly placed.

So, what is learned from this, after playing through a couple of these levels, is that there will not always be enough Karma on the map to be able to finish it. As default, it requires seven Karma pieces to finish the level. Sometimes, however, there is so much Karma on the top side that only a few or no rotations are required to complete the level. The game can be either unbeatable or incredibly easy and boring. Obviously, there are other types of configurations uses this technique. To sum up the more problematic configurations:

- Too easy: All Karma is on the top side, or very close.
- Impossible: Not enough Karma.
- Aesthetically boring: There is a side consisting of only one type of objects, e.g. a side full of soldiers.
- Unsolvable: When everything is seemingly fine, but no matter how the level is played, it just cannot be solved.

This is valuable information on how to optimize the level generation. The first thing to analyze is the Karma, which are the amount and the placement. The level has to contain the required amount of Karma to beat it. More importantly is the placement of Karma, since this has a major impact on the difficulty of the levels and game play. This can be handled in a couple of different ways depending on what the result should be.

Placing the Karma evenly on all sides, together with placing all obstacles and soldiers evenly on the sides, would create a better-looking game and therefore a very good method for making it aesthetical, but the variation of each level can be a problem. Especially if the player is playing more than just ten different levels, the levels will become monotonous.

It is important that not too many of the Karma pieces are easily available. It may be acceptable on the first levels to have the Karma close and easy to collect, however, this is not something that should happen on the later levels of difficult. A level can have ten or more soldiers, but if all the Karma is on the top side of the cube, the difficulty of that particular level will be very low.

A strategic approach is to use the Karma pieces to create difficulty. Calculating the distance from the current position to the top of the cube gives a good idea of how difficult a level is. Karma that is placed on the back of the cube, so it takes three rotations of a specific side to get it to the top side, is much harder to get to and it is likely that a soldier will be able to attack the monk in the process. The placement can also be done in such a way that the player will learn how the cube functions. Having it on a side, so the player has to rotate to be able to finish or a configuration where the player has to do a few specific moves to be able to get the Karma safely. Much like how a player solving the Rubik's Cube knows of different algorithms of how to move a specific piece/color to another position through a specific set of moves.

It is possible to generate a cube with no soldiers and still have it somewhat challenging, but the real fun and challenging levels come when the player has to figure out how to deal with soldiers. The position of the soldiers is equally as important as the placement of Karma. By looking at a cube with only one soldier, there are in essence only three different positions in a corner, in the middle of a side, or in the center.

The soldiers in the corner position are the easiest to manage; the path can be blocked in multiple ways. There are different ways to do this, one of the more efficient ways are by blocking the monk in or blocking the soldiers to restrict the movement. If the player blocks in the soldiers (first picture below), the top row and first column can be rotated without worrying about the soldiers on the top side.

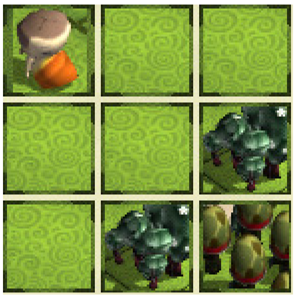


Figure 18 Blocking soldiers with only two obstacles

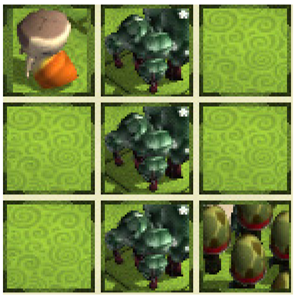


Figure 19 Blocking the soldiers with three obstacles

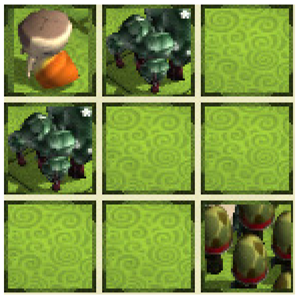


Figure 20 Blocking in the monk instead of the soldiers

If the soldiers are in the middle position of a side it is still possible to use some of the same configurations, but the player can no longer constrict the movement of the soldiers with only two obstacles, and therefore limit the available rotations, without worrying of the soldiers, to one. An example of this is shown below, and only the first column can be freely rotated.

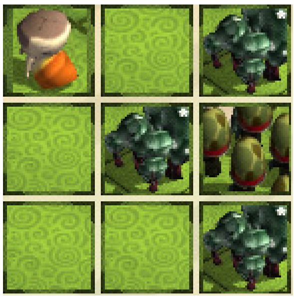


Figure 21 Blocking the soldiers if they are on the center square of a side

The soldiers in the center square are the most difficult to manage, mostly because there is only one way to effectively deal with them, by blocking the monk in.

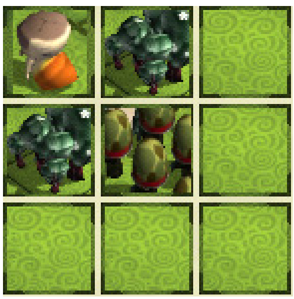


Figure 22 Blocking the movement of the monk to avoid a path to the soldiers

The best way to counter any of these soldiers is by blocking in the monk and rotating the third row or third column, but the player will be constricting the movement of the monk and cannot pick up any Karma.

The last object in the game is the obstacle. The obstacles are very important and their main role is to block the path between the monk and the soldiers. Depending on the placement of the soldiers and Karma, more or less obstacles are required. If there are not enough obstacles a level might become unsolvable, since no matter what side is rotated a soldier will be able to attack the monk.

Adjusting variables

An important factor when creating levels is to control the difficulty of the generated levels. So, using the same technique as before, but now controlling the amount of Karma, soldiers and obstacles. The level will still have the same problems as a completely random, but we will use this to study the effects of different variables.

The following set is used to determine the amount of each object. Difficulty is defined from 1 to 10, and there will at least seven required Karma pieces, which is the required amount to win the level.

Karma = (10 - difficulty) + 7
Soldiers = difficulty
Obstacles = 10

When increasing the difficulty of a level it got tougher to beat but there are still levels that become ridiculously easy. Most notably was from difficulty 8 to 10, where some of the levels were too easy, due to the Karma placement.

The most obvious rules to add to this formula are the rules about the placement of the Karma, making sure there are no levels that have all the Karma on the top side. Therefore, a level that has more than a percentage of the Karma on the top side will be discarded and a new level will be generated.

Instead of regenerating the level every time a flaw in its design has been spotted e.g. because of Karma placement or soldiers attacking the Monk on the start position, the generator would in some cases regenerate the level over 50 times. Measures to lower the regeneration rate are needed, or it will take too long to generate levels on smaller processing units e.g. on cell phones.

One way to create a work around for regenerating levels due to the monk being attacked on the starting position is to block the path between the soldiers and the monk with an obstacle. This can, however, only be done if the soldiers are not standing directly next to the monk, and if the obstacle is placed on top of a Karma, this needs to be placed somewhere else on the cube. This tweak will lower the regenerations by over half of the amount.

The obstacles are left at a constant ten, for testing purposes. This is not optimal and it should be varying but the exact formula for how many obstacles there are should be random in all difficulties. Where the first difficulties might have a lower range of how many obstacles there are e.g. 1-10 obstacles, and the harder ones would have e.g. 10-20 obstacles. The exact number of obstacles needed for a good level is hard to determine, and this factor is left out in the later iterations of the generator, simple because a better solution than a static number was discovered.

Optimizing the generator is essential for platforms with low processing power or the load times will be too high, also on a cellular unit the battery will suffer from the high load on the CPU.

Solver

To verify that the levels can actually be completed a solver is needed. Besides verifying solvability, a solver gives feedback on different metrics that can later be used for difficulty rating. Solving a level, however, requires a lot of processing power, so having it running after a generated level is not too bad on a PC, but on a mobile device, it can take quite some time.

Besides verifying the level, we can use the output of the solver for determining difficulty. A perfect solver will be able to solve all levels with the least amount of moves and the amount of different solutions. In addition to determining the difficulty, we can rate the performance of the player versus the optimal moves. However, finding the lowest amount of moves requires that all possible solutions have been explored; this can take a lot longer than just verifying solvability.

By using the solver, the generator will only create solvable levels, and regenerate all levels that cannot be solved. However, this is far from optimal; we need a way to create levels that are always solvable to minimize the regeneration rate of levels. The solver is still needed when designing such an algorithm to test whether the algorithm actually produces solvable levels.

Building the level backwards

Instead of building the level randomly, a much better approach is to generate it backwards, as discussed earlier where creating a dungeon or scrambling a completed Rubik's cube using backwards approach. We will not be using backtracking since it is more time consuming to implement, and it would increase the memory needed for storing the different states.

Using a cube consisting only of obstacles and the monk at the end position as starting points we can continue generating the level backwards.

Figure 23 A cube only consisting of obstacles.

When incorporating the rules from the other approaches and generating the level backwards by removing trees and placing paths and Karma, while moving the monk, should always give a solvable level. However, this might not always be the case.

Irreversible moves

The specific way the cube is created is by traversing the cube backwards. Some of the things to note when doing this is that it is possible to create a path and then make a move so it is no longer possible to go back through this path. Therefore, unlike moving backwards through a maze where it is always possible to turn around and go back through the path that was created, it is not always the case in A Mazing Monk.

The more specific reason for this is that the generator needs to take into account when an obstacle has been used to block the path between the monk and the soldiers. The obstacle needs to be protected from removal.

One last improvement on the generation of levels is to use backtracking to make sure the steps are all reversible. Simply using the moves of the monk and rotations of the cube when it was generated in reverse order and obeying the rules of the game i.e. the solders will attack if a path to the monk is presented.

Difficulty

Many parameters determine the difficulty of the level, both mathematically and psychologically. The psychological difficulty in A Mazing Monk is the soldiers, the more that are visible the more intimidating. Mathematically this is not the hardest of the levels, instead the distance the monk has to travel to pick up all the Karma is.

Levels with many soldiers will make the player have to figure out what is a safe move. For a computer it is easy to analyze the available moves, and it will quickly discard moves that will create a path between the soldiers and the monk. However, a person that is playing the game will have to analyze each move to make sure he does not lose the level. The difficulty rating of each level should consider the levels from a player perspective. There is no reason in this game to create a rating based on how hard it is for a solver to solve it.

When creating the difficulty some of the important things to take into consideration are:

- Distance of the Karma to the top side and the amount that are visible from the start position.

- Placement of the soldiers, if are they in the middle squares, and if they are visible from the start.

- The amount of paths on the cube, the more paths the monk can traverse the easier the level.

To measure the performance of the player there are some easily available metrics, from just analyzing how the level was played.

- How many times did the player die and restart?
- How many moves did he use versus the minimum needed to beat the level?
- The time between each move and overall.

Using the difficulty rating and the performance of the player the game will generate a new level that would fit the player. Not all the parameters should be weighted equally; a player could be playing the game a little on and off, while doing other things at the same time. However, it should give a good approximation on the skill level of the player.

4. Design

We wish to expand *A Mazing Monk* with a level generator and a solver. The level generator should ideally be able to create levels that are always solvable, and the solver is created to be able to verify that this is true. These are the basic goals of what we wish to create.

4.1 The framework

The framework of the original game is not suitable for creating new algorithms or for performing tests on a large batch of levels, so we will develop a new framework that simulates the cube. This framework will constitute the base classes that the solver and generator can built upon.

We will also construct an editor that will use the new framework to simplify level creation and experimentation. We will not discuss the editor in detail, but its code is located on the DVD accompanying this report.

Dissecting the cube

As described, the base framework consists of the classes needed to simulate a cube and the operations performed on it. The following chapters will break the cube down into its core components.

Basic elements

A cube has six faces, one for each side of the cube, i.e. a top, bottom, left, right, front and back face. The top, left and front faces are the visible faces in the game:

Figure 24 3D model of the cube, with face labels.

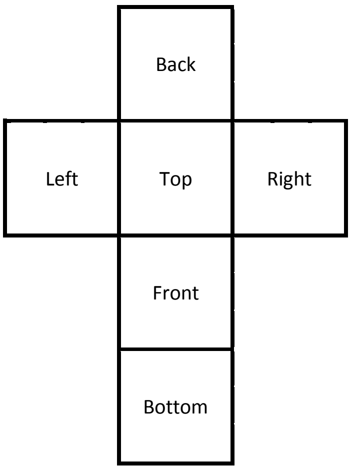


Figure 25 2D model of the cube, with face labels.

In the 3D model depicted above the right, back and bottom face are opposite to the left, front and top face respectively.

Each face consists of nine squares, referred to as cubies from now on, arranged in a 3 x 3 grid. The upper left cubie on a face has position (0, 0) and the bottom right cubie has position (2, 2). A cubie represents a tile type from the game and can either be a monk, an obstacle, a clear path, a group of soldiers or a piece of karma.

The cube has three columns and three rows with respect to the top face, indexed from zero to two, so the arrangement of the cubies, columns and rows looks like this:

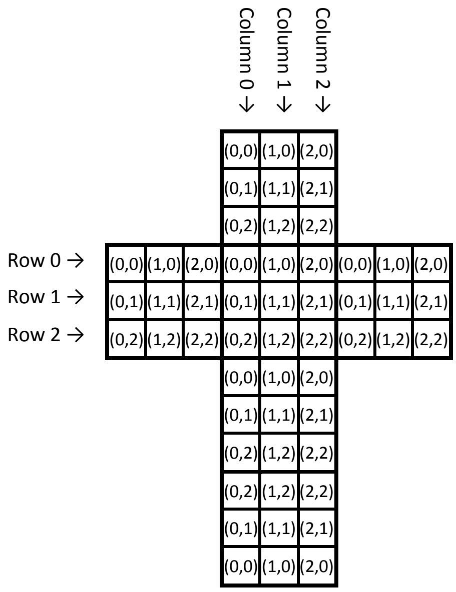


Figure 26 Cells, columns and rows of the cube.

Note that the bottom face has reversed y-coordinates for its cubies, because it is upside down compared to the top face.

Basic operations

The generator and solver will also need to perform specific operations on the cube. These are rotations and movement of the monk or soldiers.

Rotations

There are two types of rotations available in the game, row rotations and column rotations. Row rotations are performed left-to-right or right-to-left and column rotations are performed front-to-back or back-to-front. The game only allows the player to rotate left-to-right and front-to-back in the game, but the generator and solver may need the opposite rotations to undo moves, so they will be implemented as well.

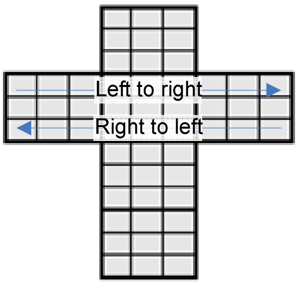


Figure 27 Row rotations

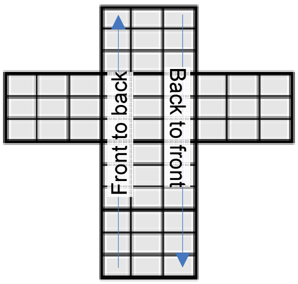


Figure 28 Column rotations

Rotating a row left-to-right shifts the content of the cells in the row across the adjacent faces in the following order: Right, bottom, left and top. The order is reversed for a right-to-left rotation.

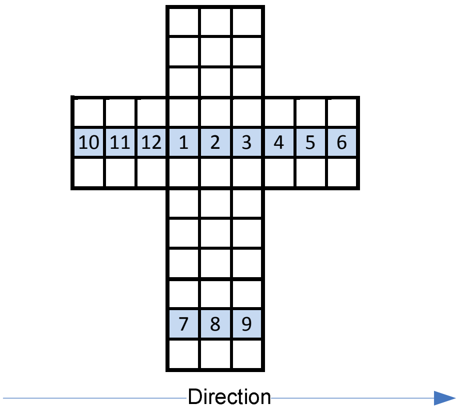


Figure 29 Rotating the numbers 1 - 12 from left to right

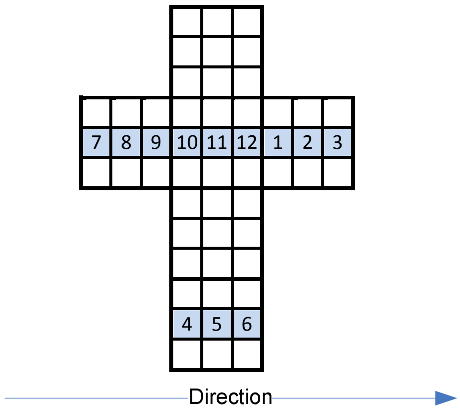


Figure 30 The cube after the rotation

Column rotations are performed in the order back, bottom, front and top for a front-to-back rotation and in the reversed order for back-to-front rotations.

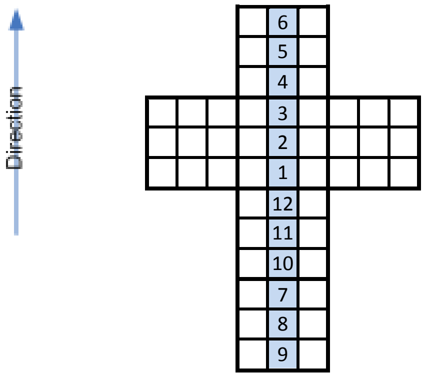


Figure 31 Rotating middle column front-to-back

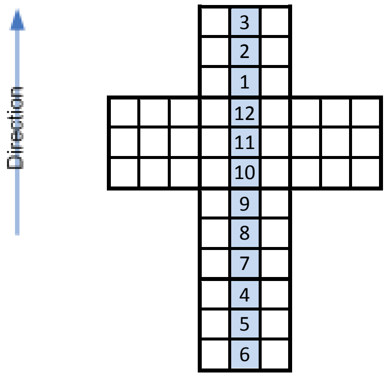


Figure 32 The cube after the rotation

When rotating a column or row that is not in the middle, i.e. row 0, row 2, column 0 and column 2, the cells of the face sharing the side with the top face, has to be rotated as well, in the same direction. Rotating row 0 for example means the cells of the back face have to be rotated counter-clockwise for a left-to-right row rotation and clockwise for a right-to-left rotation.

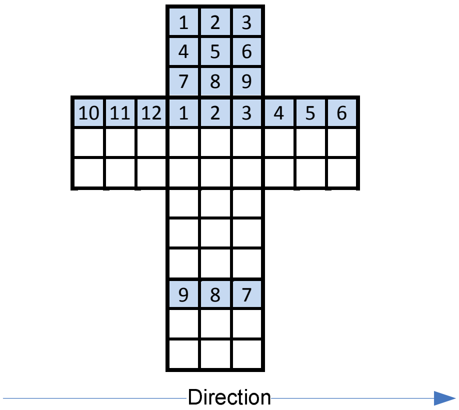


Figure 33 Rotating row 0 left to right, remembering to rotate the back face as well.

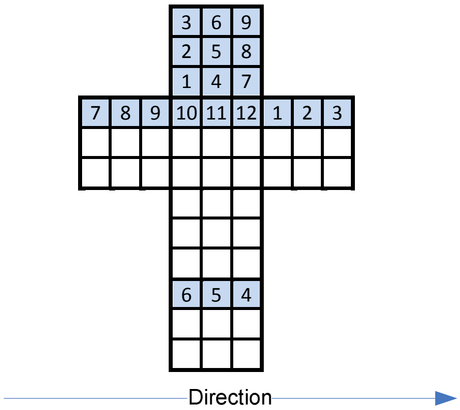


Figure 34 The cube after the rotation

Movement

To move the monk in the game, the player clicks on a square on the top face. The game then determines if there is a path from the monk to the selected square and, if so, moves the monk along the path. Similarly, when the player has performed a rotation the game determines if there exists a path from any of the soldiers on the top face to the monk and if so moves the soldiers to the monk along the path. Dijkstra's algorithm (19) was used for path finding in the original game so the same algorithm has to be incorporated into the framework. The algorithm takes the coordinates of the start cubie and the end cubie as input and returns a list of the cells the path consists of.

For clarification, the monk and the soldiers can only move up, down, left and right and only through cells with cubies representing either a path or a piece of karma.

Class diagram

To summarize, the basic elements appearing in the game are:

- The Cube containing six faces.
- A face containing a 3 x 3 grid of cubies.
- A cubie, representing the tile on a square.

Each of these elements will be encapsulated in their own separate class except for the cubie, which can be represented as an enumeration. This leads to the following class diagram:

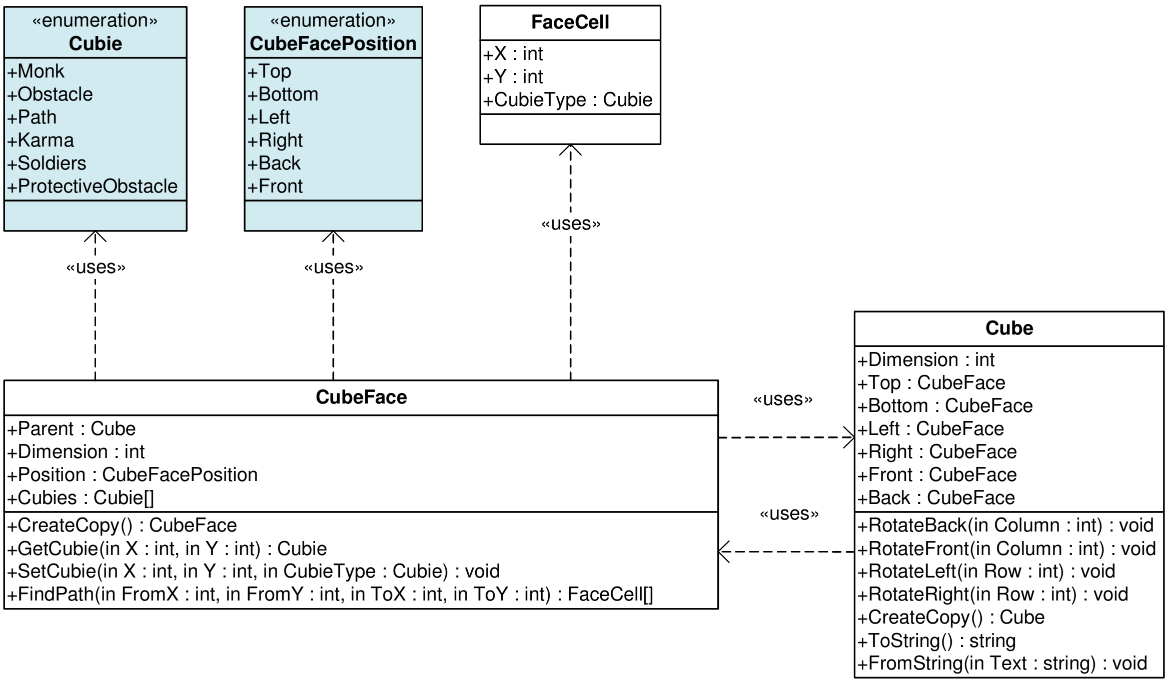


Figure 35 Class diagram of the base classes

Cube class

The Cube class contains all the rotation methods as a rotation affects the whole cube. The *RotateBack* and *RotateFront* methods perform a front-to-back and a back-to-front rotation respectively on a given column. Similarly, the *RotateLeft* and *RotateRight* methods perform a right-to-left and left-to-right rotation on a given row. The class also contains methods to make copies of it, which the level generator and solver can use to store the state of the cube at a given point in time.

Cubeface class

The CubeFace class represents a face on a cube and therefore has methods to get and set the individual cubies on it. It will also contain the path finding method *FindPath*, since path finding is specific only to a given face. The output of the method is an array of the *FaceCell* class, which contains the cubie and its (x, y)-coordinates.

Cubie

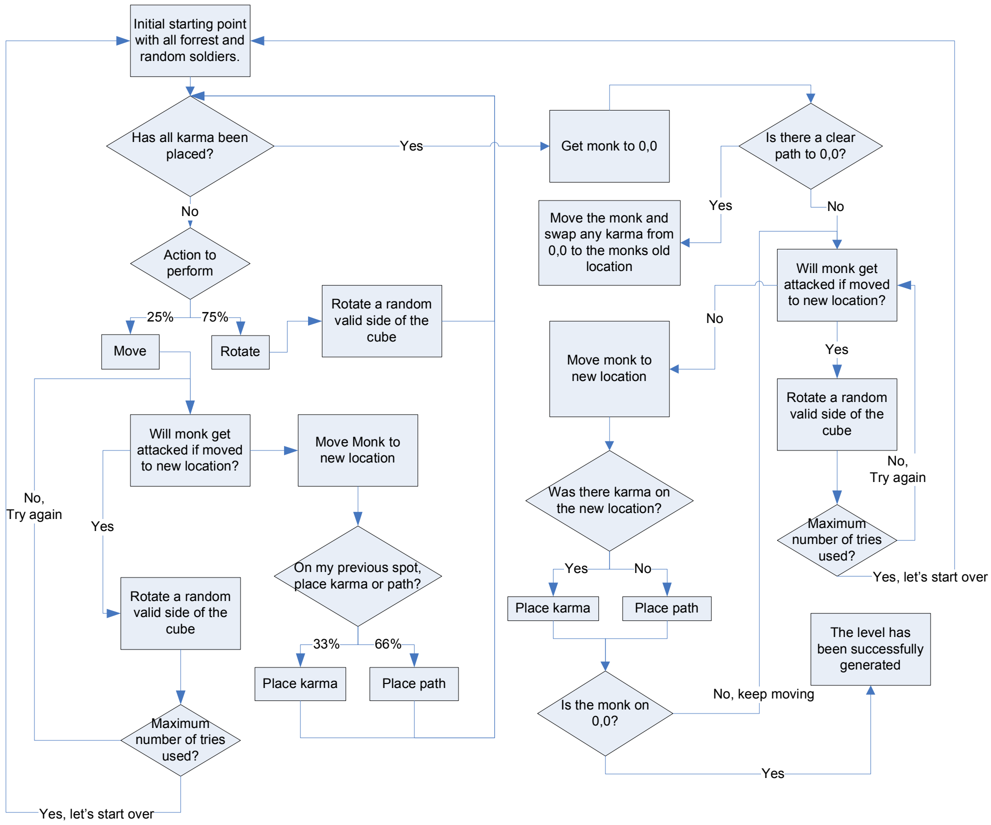
The cubie is an enumeration containing one of the following values: Monk, Obstacle, Path, Soldiers, and ProtectiveObstacle. ProtectiveObstacle is used by the level generator will be explained in the following chapters.

4.2 The generator

Creating the starting state begins with a function that creates an empty cube with the monk in the lower right corner of the cube (2,2), which is then populated with obstacles for each of the cells. The specific difficulty determines the number of soldiers needed and places these in the first cells and the cube is randomized without regard to any of the game rules. After the randomization, the generator checks if the monk is being attacked, if it is the cube is randomized until this check passes. The generated cube in Figure 39 illustrates the starting state.

The level generator will use a backwards approach following a set of steps in a loop:

1. A move; it is either a rotation of a side or a relocation of the monk to an adjacent square, removing any obstacle in the way.
2. After each move, the generator checks the current top face to see if the monk is under attack and reverses the move if this is true.
3. When the monk moves from one square to another, the previous square will become either a path or a path with Karma on it.



After each move or rotation the generator creates protective obstacles, if applicable, the generator cannot remove these obstacles since they are needed when reversing the moves. The selection of what obstacles should be marked as protective is done by checking whether an obstacle blocks the path between the monk and the soldiers. This is done by removing one obstacle at a time and checking for a path, if a path is available the obstacle will be marked as protective, if not, it will be put back.

Figure 36 No obstacles are protective

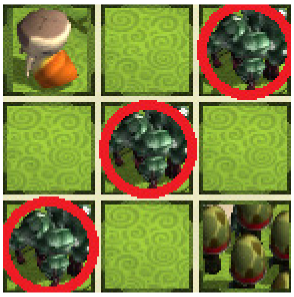


Figure 37 Every obstacle is protective

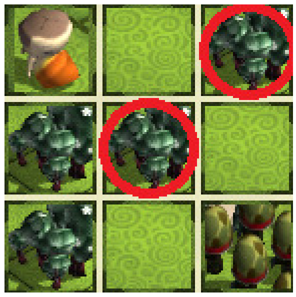


Figure 38 Only the marked obstacles are protective

When the generator has placed all the Karma, it moves the monk to the start square of the level in position (0,0). The generator will not remove any obstacles when trying to move the monk to this position, so a random row or column rotation is done until a path to (0,0) is found.

For all random events, which loop in the generation e.g. rotating a random side, there is a maximum on how many times this can occur. This ensures there are no infinite-loops. If a maximum is reached, the level is regenerated.

After the level generation loop ends, the level is scrambled by rotating the rows and columns randomly. The generator then analyzes the amount of Karma on the top side, to make sure the level is not winnable in a very few amount of moves. If a higher amount than half of the Karma is on the top side the level is re-generated. Adding checks for setups only consisting of obstacles, soldiers or karma can ensure a good variation and aesthetical look; however, this has no impact on the gameplay and it is left out of this iteration.

Visualization of the generation

The figures below show the generation process. The obstacles (trees in this case) will be tinted red when they are protective and cannot be removed by the monk later on. The first four steps are shown, then a snapshot in the middle of the generation and the outcome.

Figure 39 The start of the generation

Figure 40 Rotation of the top row

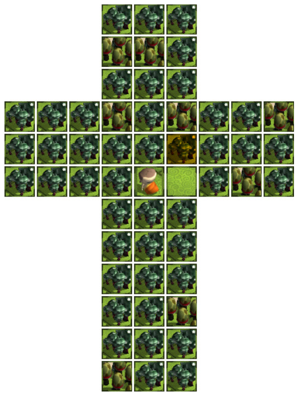


Figure 41 The monk is moved and a protective obstacle is set

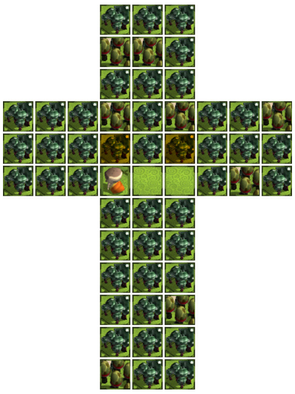


Figure 42 The monk moves again and another protective obstacle is set

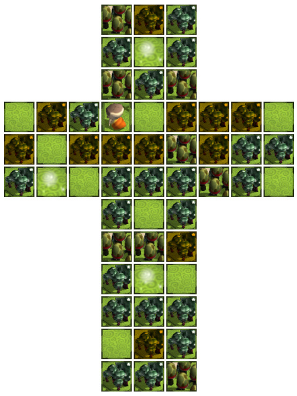


Figure 43 Snapshot in the middle of the process

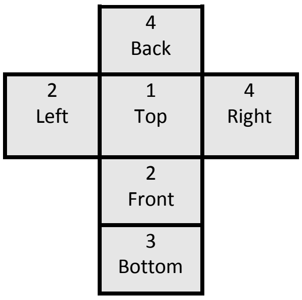
Figure 44 The final level

Difficulty

The first parameters used when generating the level is the amount of karma needed and amount of soldiers. The formula described in the theory section is used to generate ten steps of difficulty levels and each level is relatively rated in each of the ten categories.

The difficulty rating of each level uses the following factors: Karma distance, soldier placement, visible soldiers, and amount of movable paths.

Karma Distance = Distance of the seven closest Karma / Max Distance, where Max Distance is 28 which would be by measuring the distance of each karma according to the following scheme:,



Karma pieces on the top side count as 1, the two visible sides 2, the bottom 3 and the two sides on the back of the cube as 4. This means that the further away the Karma is the higher the amount is, so all Karma on the two non-visible sides will generate 7 times 4 which is why the max distance is 28. Essentially a level where three rotations are needed for each Karma piece will be harder than one where only one is needed.

Soldier placement = Amount of soldiers on the mid square / Max Amount, where Max Amount is six, one for each side. The hardest level will have a soldier on each side in the center position. As described in the theory this is the hardest of the soldier positions to deal with. Actually, a level with soldiers on all 6 center squares will be unsolvable, since the monk will not be able to move away from the starting position. However, it is still the maximum, even if it is not obtainable.

Visible soldiers = Amount of soldiers visible / Max Amount of Soldiers, the percentage of the soldiers that is visible on the given cube. This is the soldiers on the top, left, and front faces. This rating is relative to the cube, and is one of the reasons why the difficulty rating cannot be compared between each of the ten difficulty steps.

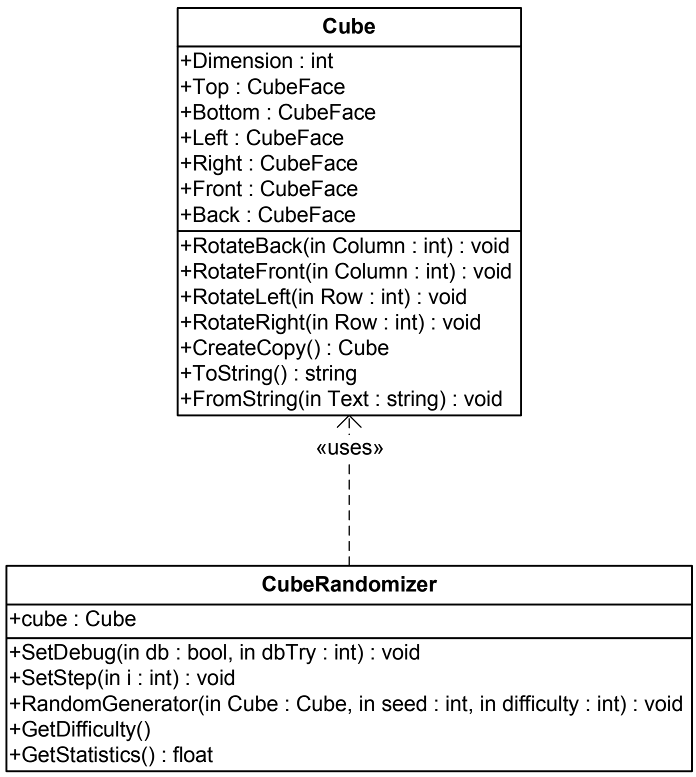
Amount of paths = 1 - Amount of Paths / Max Squares on the cube, where the max amount is 3 x 3 x 6 which is 54. A low count of available paths means that the level is harder than if there is a higher amount. An easy level of difficulty e.g. 1 to 5 will generally have a very high number of paths whereas the higher tiers of difficulties tend to have a lot fewer.

Adding these four parameters together gives a difficulty rating for the level ranging from 0 to 4. It is relative to the amount of soldiers and Karma; therefore, comparing this rating across different setups is not applicable.

Class diagram

The randomizer class takes a blank cube and molds it through the RandomGenerator-function. Using the SetDebug and SetStep-functions a systematic method can be used to visualize each step of the generation of the level for debugging.

GetStatistics-function is mostly used for statistics on the generation and is only used in the editor, whereas the GetDifficulty-function is used in both versions and returns the relative difficulty of the current generated cube, and must be run after RandomGenerator.



4.3 The solver

A solver will verify that the levels produced by the generator are indeed solvable as described earlier. If it is able to find all possible solutions to a given cube, it will also provide information about the shortest path possible and the number of unique solutions, which can improve the difficulty-rating calculations. If this is not practically possible, which the internal testing of the algorithm will reveal, the solver should, as a bare minimum be able to determine if a level is solvable within a reasonable amount of time.

Inspired by the backtracking algorithm, the solver will try every possible combination of the cube using a finite state machine and heuristics to pick the next state to investigate. This requires a partitioning of the solving process into states.

States

A state constitutes all the parameters needed to recreate a point in time in an A Mazing Monk level. These are:

- The cube as it looks at the given point in time, i.e. its configuration.
- The monk's position (x, y) on the top face.
- The amount of spheres remaining, zero meaning all have been collected.

In addition, some extra parameters can help the solver pick the next best candidate. These are:

- The amount of moves performed to get to this state.
- The parent state that spawned this state.
- The children states created from this.

Finally, to recreate the path that led to the given state, the moves performed on the top face are also stored in a list. A move in this case is either a rotation or a relocation of the monk. The state finds the path by traversing back through the tree, appending the moves of each visited state to a list until it reaches the root. This can be used to playback the solution in the editor for example as a useful debugging tool.

The parent and children links organize the states in a tree structure, which the solver will use to find the optimal path through the level as explained later on in this chapter. The following diagram sums up the elements needed:

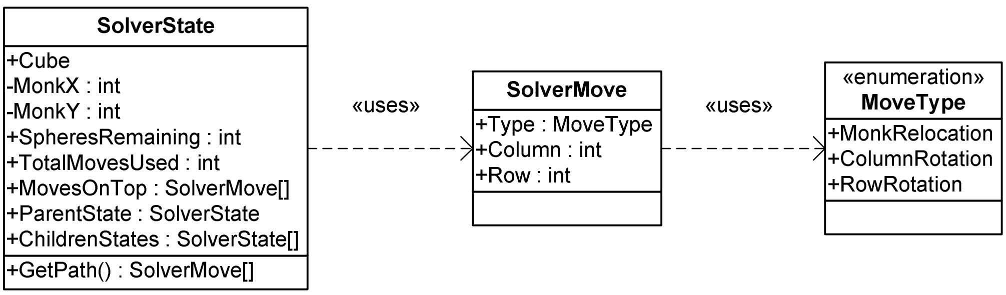


Figure 45 Diagram of the state class

Best candidate heuristic

Newly created states are stored in a list, named the unprocessed list, from which the solver iteratively picks the next best candidate to try. As an initial heuristic, the solver picks the state that has collected the most spheres using the least amount of moves first. This may be subject to further refinements when the algorithm is tested on the generated levels.

Avoiding duplicates and optimizing the path

The solver will also keep a list of explored cube configurations, named the history list, to prevent it from repeating previous actions. A hash-table can implement this, using the cube formatted to a string and the state it belongs to as a key/value pair.

The solver adds a new state to the history and unprocessed lists, if its cube configuration is not already in the history list. If an old state already exists, the solver compares the amount of moves they have used so far and replaces the old state with the new one, if it has used the fewest. Otherwise, it discards the new state.

When replacing an old state, the solver transfers its children to the new state and updates their move count, so it reflects the new path. This optimizes the state tree continuously as the solver progresses through the level. The tree shown in Figure 46 illustrates the process. A circle represents a state and the upper number inside the circle is the amount of moves the state has performed on the top face. The bottom number is the total sum of moves (parents + own).

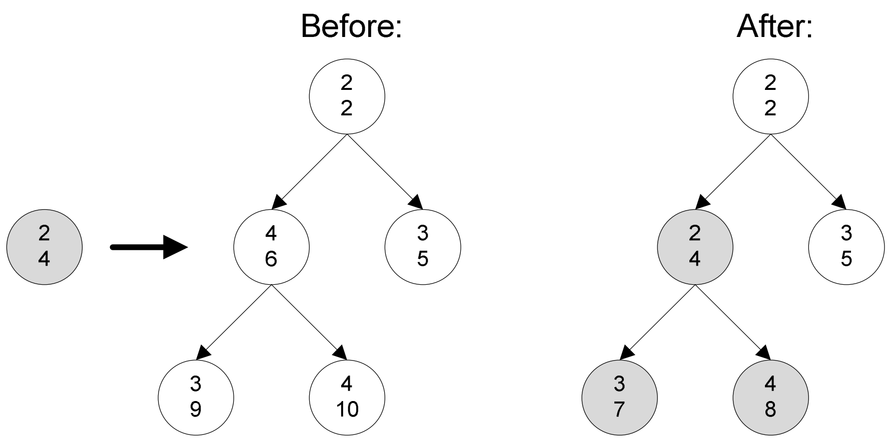


Figure 46 State tree before and after a replacement

The state machine

The following flowchart illustrates the basic structure of the solver algorithm:

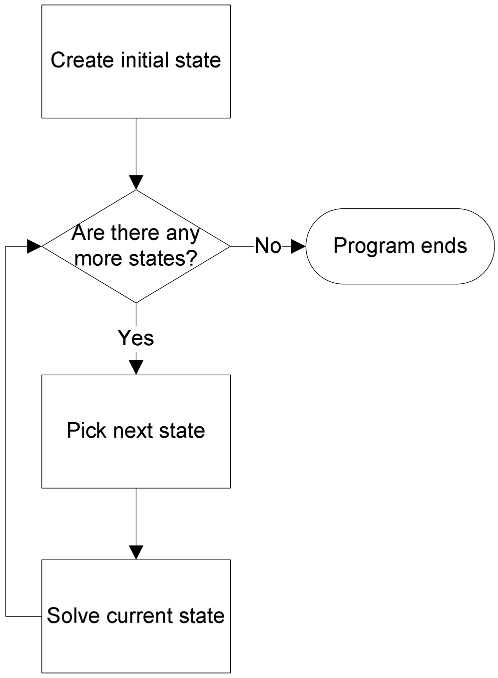


Figure 47 Basic stack based solver.

The algorithm consists of three stages and a loop.

Create initial state

The first stage simply creates a state from the unsolved cube, setting the monks position to (0, 0) and the spheres remaining to 7, and adds it to the list of unprocessed states.

Pick next state

This stage performs the search heuristic on the unprocessed states and picks the best candidate.

Solve current state

This stage examines the current state and determines if there is a solution. If not new states are generated from this one. The stage itself can be split into two, solving the top face and performing rotations. The following describes how to solve the top:

1. If spheres remain to be collected, pick every sphere on the top face reachable from the monk's position.

2. If no spheres remain, is there a path to the end square? If yes, move the monk to it and mark the state as a solution. End the stage.

3. Otherwise, perform rotations.

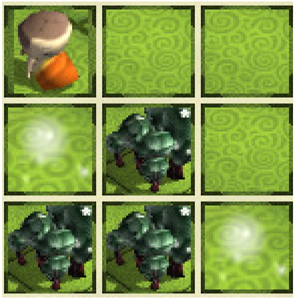


Figure 48 A top face with two spheres reachable for the monk.

Figure 49 A top face with the end square (red) in reach. The path is marked in yellow.

The algorithm performs rotations as follows:

1. For each path-cubie on the top face that the monk can reach:
    a. Calculate all legal column and row rotations from the path square.
    b. For each legal rotation:
        i. Create a new state from the current state.
        ii. Move the monk to the path.
        iii. Perform the rotation.
        iv. Add the new state to the unprocessed list.

A rotation is legal if:

- It does not rotate the row or column the monk is standing on.
- It does not create a path between the monk and a group of soldiers.

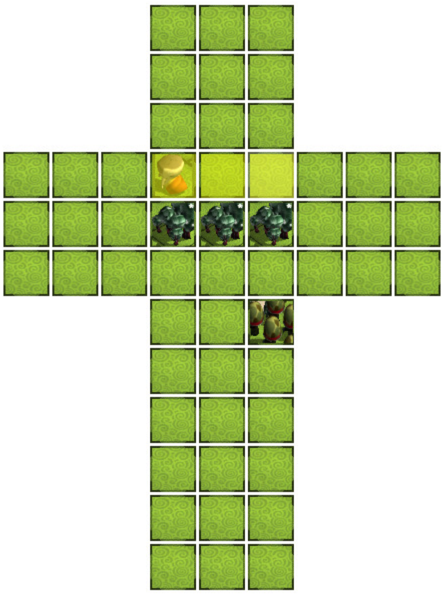


Figure 50 The paths (marked in yellow) from which the rotations are performed.

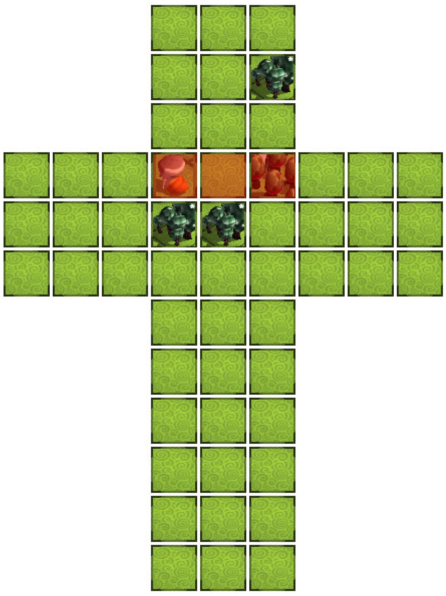


Figure 51 An illegal rotation has been performed, clearing a path between the monk and a group of soldiers.

Loop

The loop repeats the three previous stages as long as there are states left in the unprocessed list. The loop can terminate after the first solution, if the solver only has to determine whether a level is solvable or not.

Flowchart

A final flowchart of the solver is given in Figure 52 below:

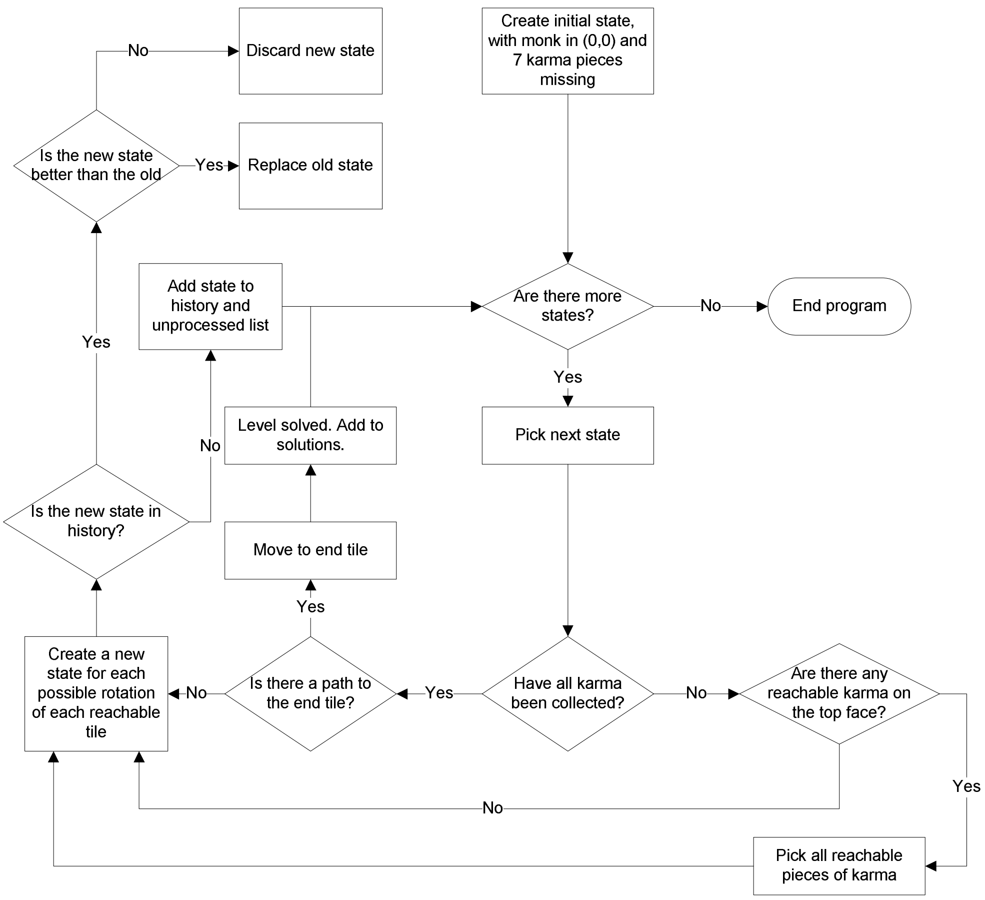


Figure 52 Flowchart of the solver algorithm

Class diagram

The class diagram (Figure 53) introduces the Solver and State class to the framework as well as a SolverMove class and a MoveType enumeration. The Solver class encapsulates the solving algorithm as described in the previous chapters. The Solver.Solve method starts the solving process and returns true if the cube is solvable and false if not. The Solver.Solutions property contains the discovered solutions as an array of State objects. The State.GetPath method returns the moves taken by the solver to reach the given state.

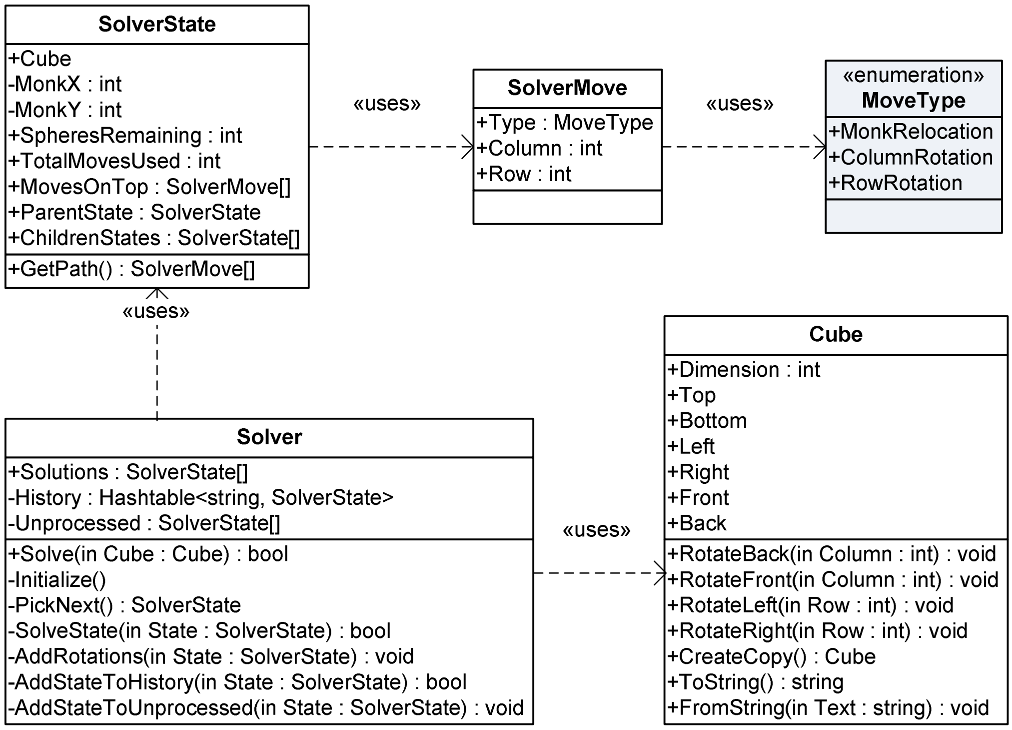


Figure 53 Class diagram of the solver

5. Implementation

5.1 Implementing the Generator

The level generator is created as described in the design section. It uses a random approach to generate the levels, with nine different checks to ensure the generator does not end up in an infinite-loop and always generates solvable and satisfactory levels.

One thing we noticed when optimizing the implemented generator was that the biggest performance gain was achieved by finding ways to cut down on the amount of times it regenerates a level, for example by revising the strategies that changes objects on the cube or reverts moves that lead to infinite-loops. However, care must be taken to ensure that this type of optimization does not come at the cost of added computations and therefore added time.

5.2 Implementing the solver

The implementation of the solver follows the design specifications, meaning the parent state generates a child state for each legal rotation it can perform. The algorithm then picks the state that has collected the most spheres using the least amount of moves as the next candidate. Early test results showed that this implementation was able to find the first solution within a couple of seconds for the easier levels and 10 – 20 seconds for the harder levels. On some of the harder levels, it would take a several minutes to find the first solution.

By examining the unprocessed states, sorted by the karma collected and moves used, we discovered that the monk had stalled in many of the cases where the solver took more than 20 seconds to find the first solution. Stalled means he was standing in the same square unable to reach other squares but was able to rotate the sides and generate new states. The monk had essentially reached a dead end that the rotations could not bring him out of, but since the generated child states had collected more karma than the other unprocessed states they would always be picked first.

We then modified the algorithm, so whenever it adds a state to the unprocessed list, it counts the number of times the monk has not moved and stores it in a stalled counter in the given state. The selection process was then altered, so it first sorts the unprocessed list based on the selection heuristic, then runs through the list and picks the first state, which has stalled for less than four moves. We chose the number through testing and four yielded the best results. This tweak dramatically increased the performance of the solver, and cut down on the number of levels that takes minutes to solve.

5.3 Implementation of the final game

This chapter describes the changes made to the original game, i.e. procedural level generation and two game modes to display the new abilities of the game.

Procedural level generation

The original game featured three levels created by a level designer; the new version is capable of generating levels procedurally. The current implementation generates a level from a given seed and a desired overall difficulty and then calculates a difficulty relative to all levels with the same overall difficulty, allowing them to be ordered. There are multiple ways to include this in the final game; however, each has some advantages and disadvantages.

Generating at the player's device

A lot of processing needed client-side.

Generating the levels on the client's device requires plenty of processing power, since producing a specific difficulty may require more than one try. This method will however enable the game to create all possible level combinations and make it truly "infinite" as it generates the levels on the fly.

Pre-generating and storing to a file

Little processing needed, just read the level from a file.

Another option is to pre-generate a large amount of levels, sort them by difficulty and store them to a file. The client then only has to read this data from the file to recreate a level with a specific difficulty. In addition, if the game keeps an index of the current file entry, it only has to increase this index to get a level that is more difficult than the previous one.

Obviously, this method adds to the disk space requirements, depending on the amount of levels created. Pre-generating too many levels could cause the level-file to exceed the disk space available at a given platform; on the other hand creating too few would break the illusion of an "infinite" game.

There are two ways to store the levels in the file. The first method converts a generated cube to a string of a known size and writes it to the file. The second method only writes the seed and desired overall difficulty to the file. The advantage of the first method is that the client only has to parse the text string to recreate the cube, whereas the second method uses the generator to recreate the cube and therefore requires more processing power. The advantage of the second method is that it requires less disk space to store a given level. Storing a cube as a string, using one char per cubie, requires 3 x 3 x 6 x 1 byte = 54 bytes per cube. Storing a cube as seed + diff requires 4 bytes + 1 byte = 5 bytes per cube.

Decision

Testing revealed that generating cubes on the fly was too time consuming using the current implementation. The game should get progressively harder as the player completes the levels and since the generator is not currently able to generate levels with specific relative difficulty, achieving this with a generator required too many retries. Instead, 10,000 levels are pre-generated for each overall difficulty rating available (one to ten) and stored. The number was arbitrarily chosen and it can be adjusted for future versions of the game if it is found to be too big or small.

A cube entry is stored in the file using only its seed to minimize the space needed. Creating a cube using the generator takes on average less than a second, so it was decided that the file size was more decisive than the computations required. If it was possible to store each entry as bytes, they would take up 5 bytes each as described earlier, and getting a specific entry could be accomplished using the following formula: entry = index * 5. However due to a bug in Unity, the entries have to be stored in text format, meaning they have a variable size, so each entry is separated with an end-of-line character. This means that the entire file has to be read and parsed before specific entries can be picked. It was therefore decided that the levels should be stored in ten separate files, each corresponding to an overall difficulty rating. This way the game does not have to retrieve all 100,000 entries at once and since the overall difficulty can be inferred from the file name, it does not have to be stored, resulting in smaller level files.

Game modes

Inspired by other casual games, such as Bejeweled , we implemented two game modes that use the level files in different ways.

Campaign mode

The first mode is the Campaign mode, which resembles the original game the most. In this mode, the player has to complete 10 levels of increasing difficulty. When the tenth level is completed, the monk becomes Buddha and the reward screen from the original game is displayed to the player.

The generated levels are sorted by difficulty and divided into N amount of groups of equal size, where N is the numbers of levels for the game mode, which again means N = 10 in this case.

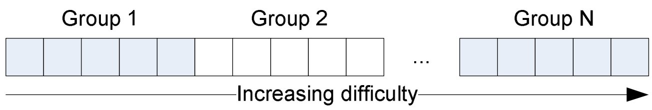


Figure 54 Sorting and grouping the generated levels.

The game then picks the next level randomly from its corresponding group. The game picks level one the first group, level two from the second and so on.

Infinite mode

The second mode is the Infinite mode. In this mode the game never stops, instead the game picks a new level based on the player's performance on the previous level. Therefore, while the campaign mode had a fixed difficulty increase, this mode has a variable difficulty increase or decrease.

The players performance is based on the number of attempts needed to solve the previous level, i.e. the number of times the monk died. If the monk did not die, the game takes a big jump forward in the difficulty rating. If the monk died once, the game takes a smaller jump forward in the difficulty rating. If the monk died more than once, it decreases the difficulty a little bit.

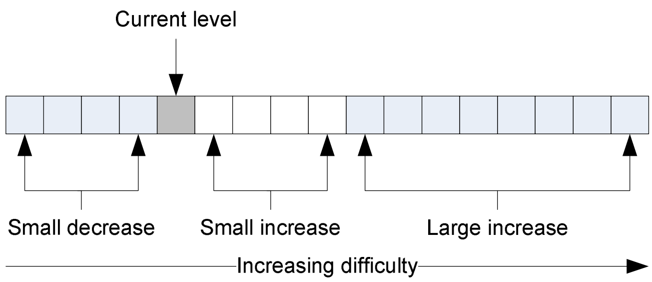


Figure 55 Picking the next level based on the player's performance

In this implementation with 100,000 pre-generated levels, a big jump is defined as skipping 10,000 levels and a small jump is defined as skipping 2,000 levels. After a certain amount of levels the pre-generated list will be exceeded, so it needs to either circle around in the upper tier of levels in the list, or generate new levels. Instead of just using the pre-generated levels we decided that it would generate levels from difficulty 8 to 10 and with a rating above 2.0, which is the median of the levels in this difficulty range. This way the performance of the player will affect whether the next level generated is difficulty eight, nine or ten. The levels generated will be a challenge for every player and it is possible to play endless amounts of levels that are of roughly the same difficulty.

5.1 Version incompatibility

The 2D editor where our tests are conducted is written using XNA on the 3.5 .Net framework. The 3D version is created in Unity3D 2.6, which runs on Mono 1.2.5, which is an open source equivalent of .Net. Even though the code is written in C# there are some differences in what libraries are available, most notably are LINQ and the *var*-keyword not available in Unity3D.

The more specific challenges that arose when implementing the solution in both engines are discussed below.

Unity3D TextAsset

Pre-generating 100,000 levels into a text file for each difficulty from 1 to 10 and storing each seed and difficulty and then ordering them by seed requires around 700 KB per file, this could be compressed by using a binary format instead of plain text, however, importing a binary file into Unity is not possible. The only possibility is to use the TextAsset class, which will read the file as a text file, and this breaks the binary file, so this solution was discarded and a pure text based file was used.

However using an external database would decrease the space required for the application, but a connection to this database must be present every time the game is played. This would work for a browser-based game, but porting it to a mobile device would not be possible. Therefore, for now, sticking to the TextAsset class is the only viable option.

Random number generator

When all levels are pre-generated on a computer with plenty of processing power, the level will be only be stored as the minimum needed for regenerating this level, which are the seed and the difficulty rating. So in order to actually regenerate the level, the random number generator needs to generate the same numbers from a certain seed in both our 2D XNA version and the 3D Unity version. One might assume the two frameworks have the same implementation of *System.Random*, but this is not the case.

The levels are generated using the 2D version due to this being faster than the 3D version, but there was a problem. The level generated in the 3D version was not the same as the one calculated in the 2D version. This is a problem and the pre-generation method does not work unless the same seed generates the same level in both versions.

Therefore, instead of using the standard implementation of *System.Random* in both frameworks, we implement another random number generator that will generate the same numbers independent of the framework. The algorithm used is the Mersenne Twister (6), which generates numbers fast and with an equal distribution of the numbers in the given range. The algorithm is not suitable for cryptography since the numbers it generates can be guessed after a sequence of only 624 numbers in the 32-bit version. However this is not a concern for our use, the player will never notice any reoccurrences in the levels generated.

5.2 Test and results

Sample

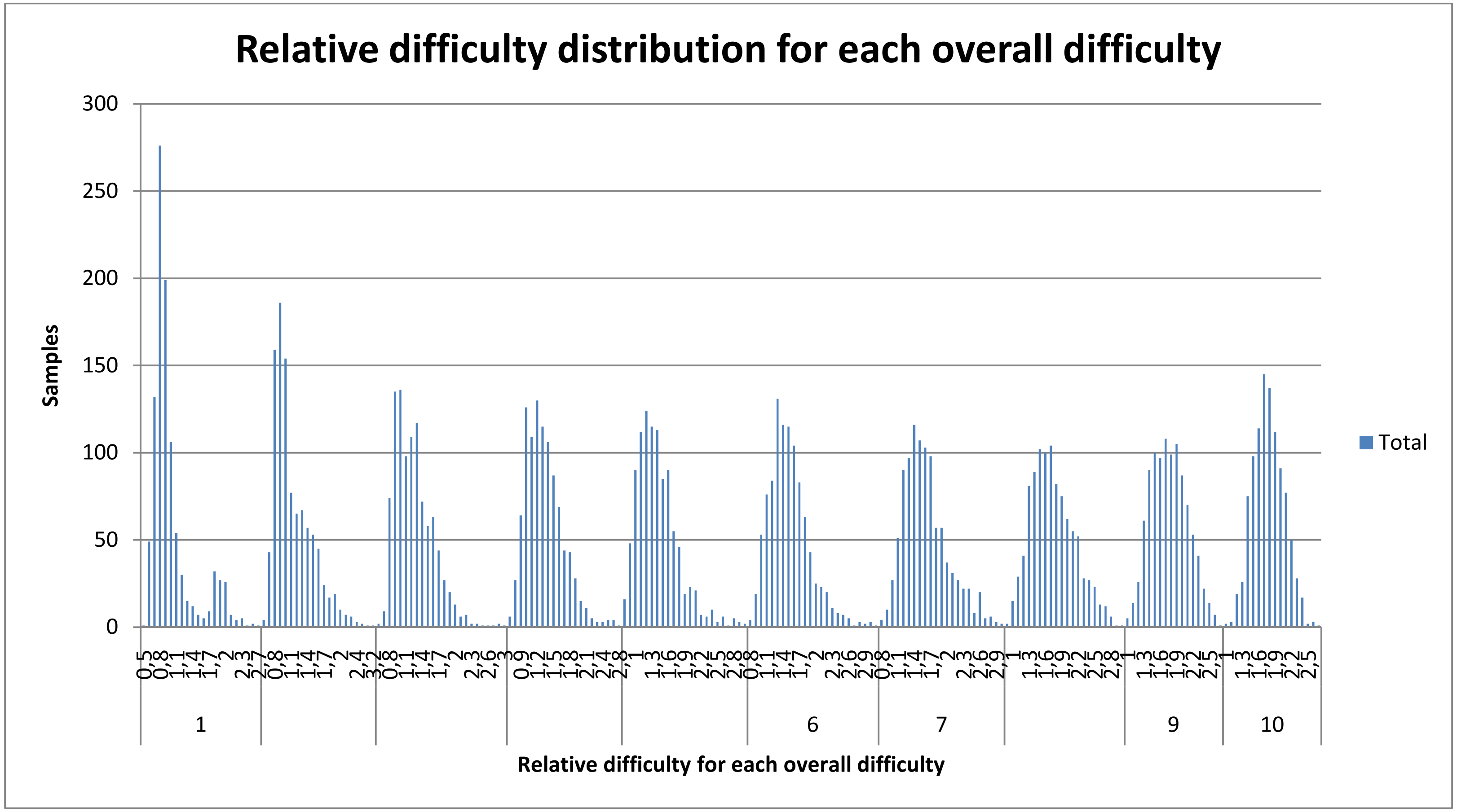
The following graphs are created from a sample of 1000 levels for each overall difficulty setting, one to ten. A computer with an Intel i7 930 @ 2.80 GHz processor generated all the levels and extracted the statistics from the process.

To ensure that every test sample got the same amount of CPU time, the test program ran in a single thread. Obviously, making the generation process multi-threaded would enhance the performance on multi-core CPU's, but for testing purposes single threading is best. This also reflects the capabilities of mobile devices better, as they usually contain a single core processor with limited processing power

Difficulty

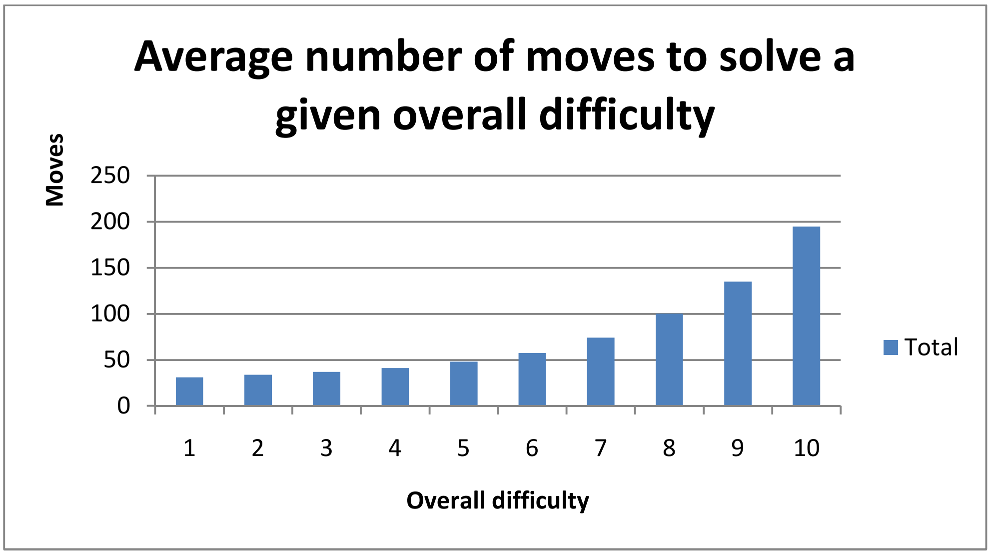
Graph 1 show the distribution of the generated levels' relative difficulty ratings for each overall difficulty setting. The x-axis displays the relative difficulty ratings for each overall difficulty and the y-axis displays the amount of samples generated for each relative difficulty.

For the first six overall difficulties, the generator mostly produces levels that fall within the range of 0.7 to 1.5 in their relative difficulty rating. For the last four overall difficulties, it mostly produces levels skewed towards a relative difficulty rating of 2.0. This graph shows that for each overall difficulty, the levels produced by the level generator tend to have the same or nearly the same relative difficulty rating. This illustrates the problem of generating levels on the fly, where a specific difficulty rating is needed.



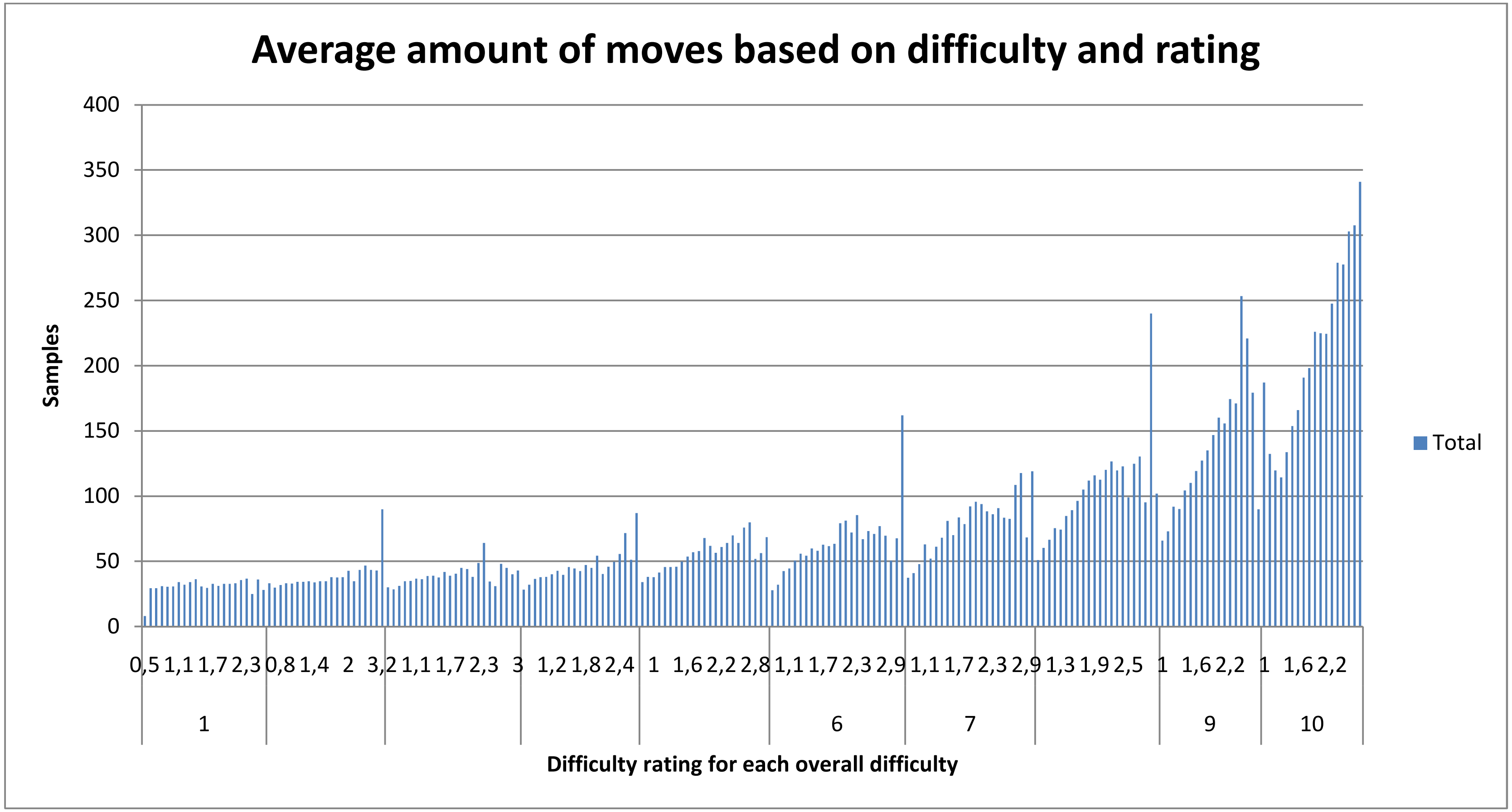
Graph 1 Difficulty rating for each difficulty

The solver used by the generator returns only the first solution it finds for a given level, along with the amount of moves it takes. The move count can give an indication of whether the levels actually increase in difficulty, at least from the solvers point of view. Graph 2 shows the average amount of moves needed to solve a level for each overall difficulty. The amount of moves are growing with the increased difficulty and more rapidly after difficulty five, which could indicate that the earlier difficulty levels are too similar in terms of challenge.



Graph 2 Average number of moves to solve a given difficulty

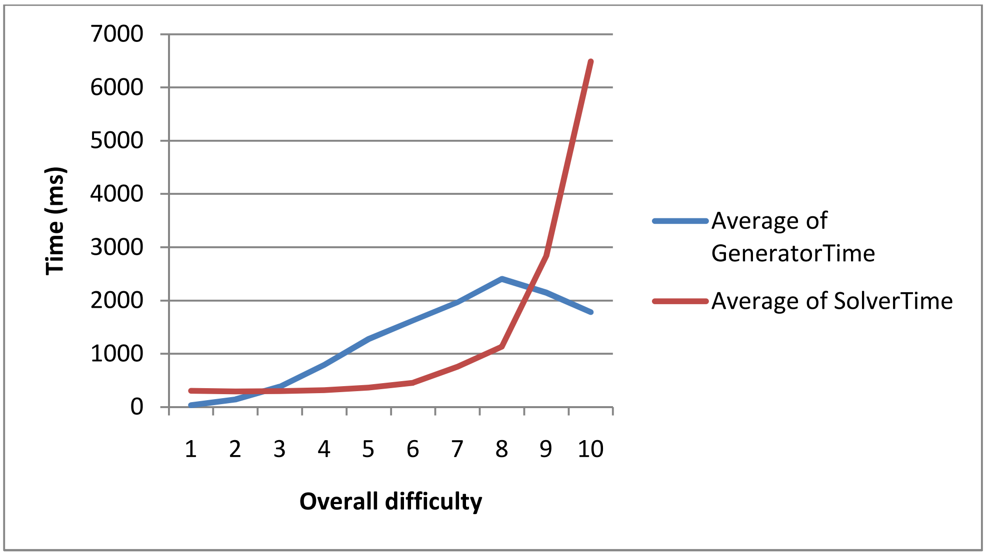
Graph 3 expands on this by showing the average amount of moves need per relative difficulty. Looking at this graph in detail shows that a higher overall difficulty rating will increase the amount of moves needed for the solver to complete a level. The exact relative difficulty rating is not as important as the variance and spread of the moves and the important thing to notice is that there can be a big difference between the easiest levels and the hardest levels. However, the solver only return the first solution, which might not be the shortest.



Graph 3 Average amount of moves based on difficulty and rating

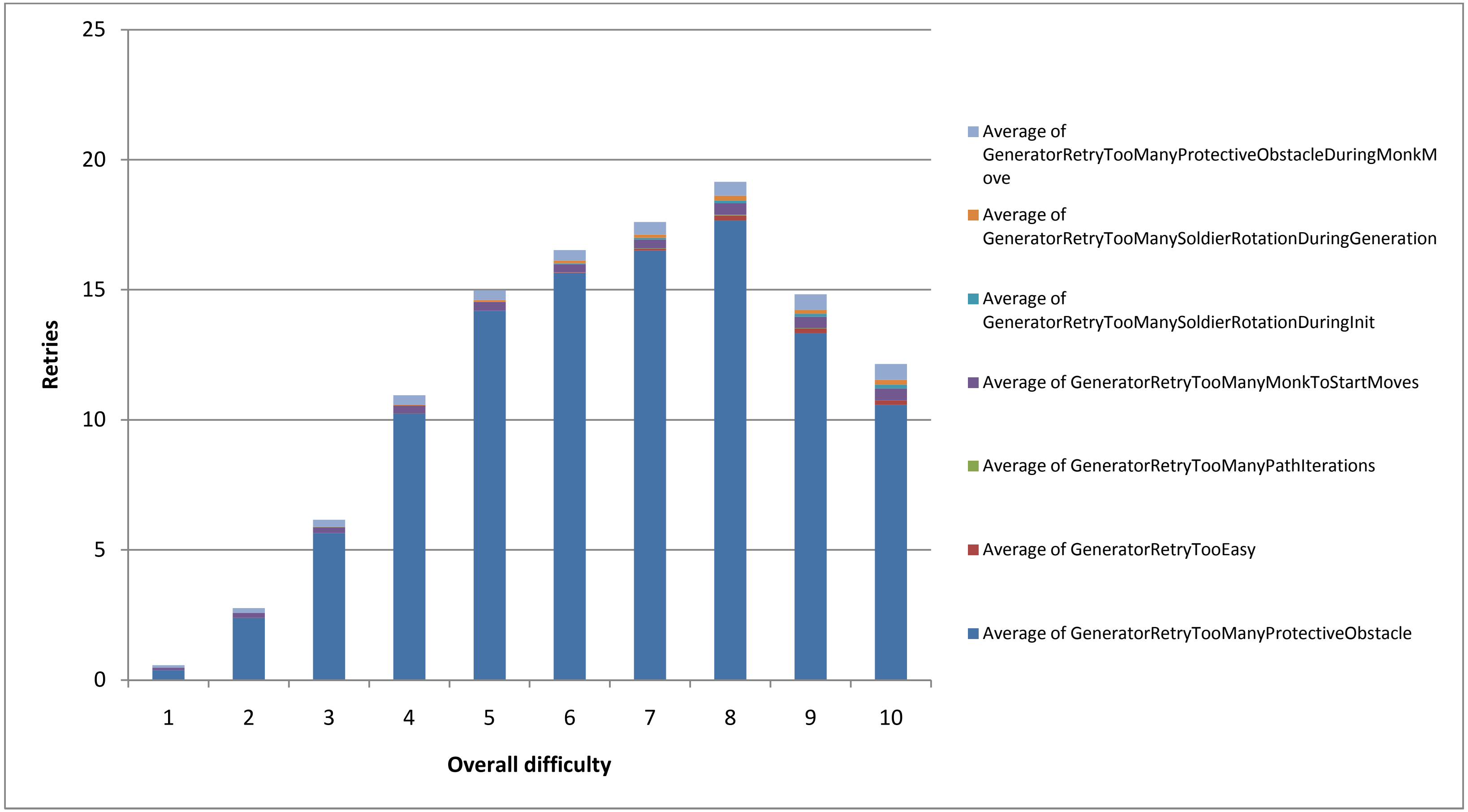
Timing

It takes time to generate a level; it also takes time to solve it. Is it possible to create a generator that also solves the level generated, before showing it to the player? Graph 4 shows the average amount of time in milliseconds for generating (*GeneratorTime*) and solving (*SolverTime*) each level based on the difficulty. If the solver has to verify each level after the generation on a harder difficulty, it can take up to eight seconds in total on our testing computer.



Graph 4 Time it takes to generate and solve levels based on difficulty

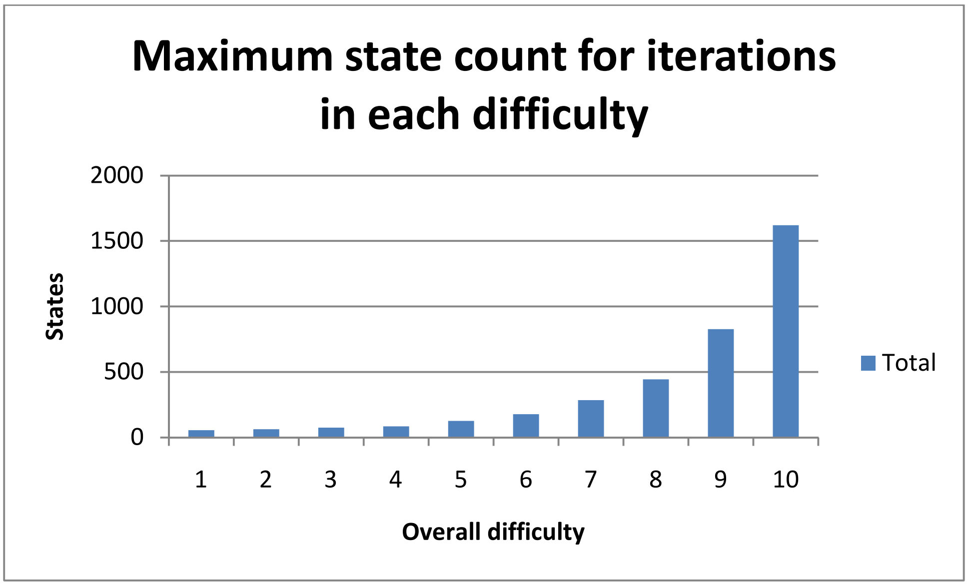
Taking a deeper look at the generator, one of the parameters that really increases the time is the amount of re-generations that it has to do. In Graph 5 it is shown that the average amount of retries peaks on difficulty eight and follows the same graph as the average time it takes to generate a level. The graph also shows that the main contributor, to the total amount of retries, is when the generator restarts due to an infinite-loop with protective obstacles. The details of this re-generation is when the monk has done more than 100 tries to move to a location but is blocked by protective obstacles. This is where our main attention should be focused when optimizing the generator, as the rest of the reasons for the generator to restart are all acceptably low.



Graph 5 Average amount of retries for each difficulty

The solver drastically increases the amount of time used when the difficulty rises above five (Graph 4). However, the lower tier of difficulty seems to take around the same amount of time to solve, just as when looking at Graph 2 where the moves needed first increases after difficulty five.

The same is shown in the total amount of states the solver has per iteration step based on the difficulty in Graph 6.

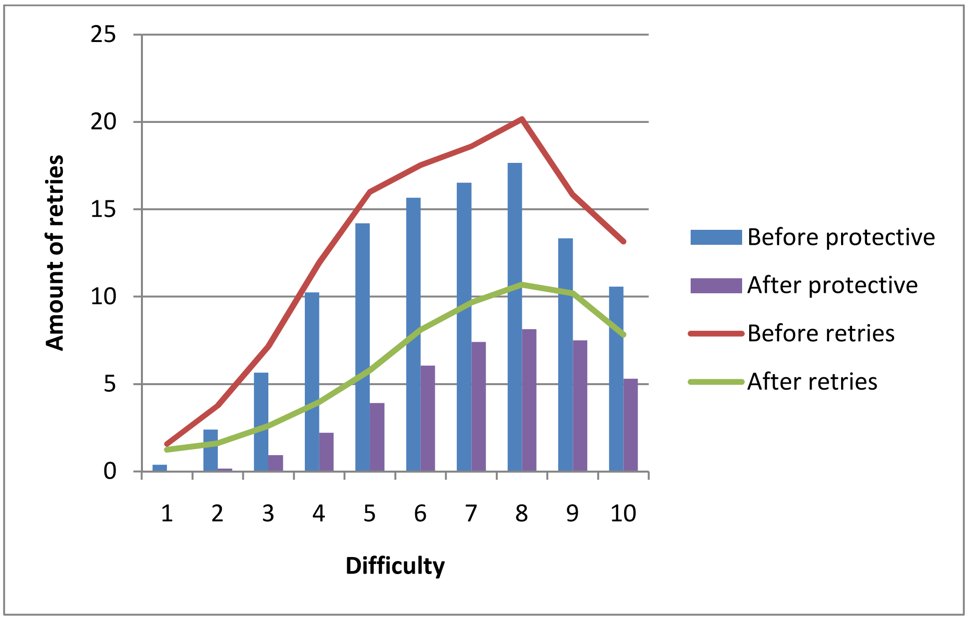


Graph 6 Maximum state count for iterations in each difficulty

Optimizations

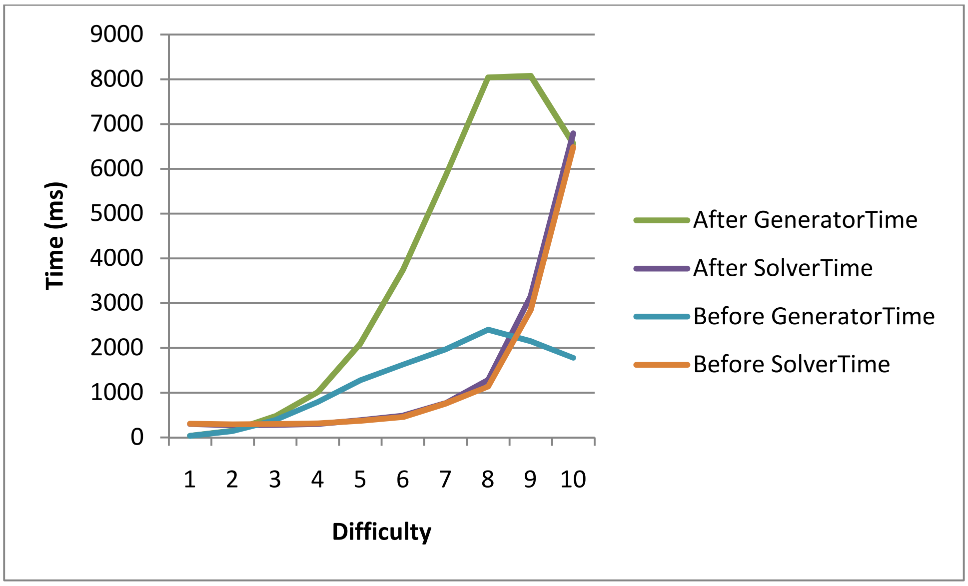
After studying the graphs, we decided to tweak the generator so it would regenerate the level fewer times. This was done by simply changing the criteria for when it would regenerate due to too many protective obstacles from 100 to 1000. This is obviously a drastic change, but also an experiment on how this affects the generator.

As seen in Graph 7 the total amount of retries are reduced to about half the amount, and the main contributor to the total is still the protective obstacle check.



Graph 7 The amount of retries in total and specifically due to protective obstacles

A more important factor is the amount of time it takes and if this has been reduced the same way as the amount of retries. Graph 8 shows the generator and solving times for each difficulty before and after the change was made, and the change clearly did not optimize the generator as the time it takes to generate a level has been increased by a large amount. However, if we tweak this change a performance increase can be gained.



Graph 8 The average amount of time it takes to generate and solve a level before and after the change

5.3 Interview with game designer on the final product

The game designer of the original game was invited to try the new version of *A Mazing Monk*, which introduced procedurally generated levels and adaptive difficulty. He noted that the Infinite Mode provide a nice new challenge for the player, but the levels and game play felt monotonous after a while. He suggested that the game could benefit from more random elements in the models to keep it fresh, such as procedurally generated grass textures. He imagined a rain forest setting with lots of life and energy that the player would want to explore and continue playing. In general, he thought the new levels were harder and more challenging the original ones.

6. Discussion

In the current implementation, the solver is able to find the first solution for most levels in a reasonable amount of time (less than 3 seconds). Cases exist where an unreasonable amount of time (more than 20 seconds) is needed to find the first solution and finding all solutions is not possible without spending considerable time. Due to these circumstances, the solver is not fit for runtime verification of levels generated on the client machine or for providing exact data usable for additional metrics for determining the difficulty.

During the level generation process, the solver verified each level with a time limit enforced upon it to keep the process from stalling. If a given level could not be verified within 100 seconds, it would be scrapped and a new one generated. In the process of generating 100,000 levels, approximately 100 levels in the difficulty levels of seven or higher were scrapped due to timeouts and zero was scrapped due to the solver finding them unsolvable. Since the solver is currently our only way of proving a level is solvable, we find it necessary to include the solver in the level generation process. Given that the solver requires excessive time to verify certain levels, the only viable option for this implementation is to generate the levels on a dedicated machine and storing them to a file for use on the client machine. While this adds to the disk space needed, enough levels can be created to make a game seem infinite to the player. 100,000 levels demands less than 1 MB of disk space and regenerating a level from a seed without verification (which has already been done by the dedicated machine) takes around 1 to 4 seconds on a modest machine. Another advantage to this approach is that the levels can be examined afterwards and sorted based on different heuristics. The current levels are sorted by their overall and relative difficulty. In addition, the levels could also be sorted based on their variety, ensuring that two successive levels are not too similar.

Due to time and money constraints the main question of game developers is, is it worth it and should we use it in our game? The advantages of using procedurally generated levels are plenty and it shifts the focus of the game and level designers to other tasks, than just creating levels. Such as tweaking and introducing new elements. In addition, it opens up for new gameplay and reward schemes. There are some disadvantages as well. The aesthetics of the game can be ruined by and the game only contains dull and boring levels without a human touch. So a level might be mathematically challenging but if the presentation is bad, no one will buy the game anyway. Therefore, it is important to use a lot of effort on tweaking these aspects of the generator. So a game where aesthetics are a vital part of the experience a random level is not always wanted or appreciated, the decision should be made on whether it adds to the experience of the game or not. The reason should not be solely to increase the longevity of the game, which was mostly the reason why we wanted to do it. We discovered though the project that it opens up a completely new aspect of gameplay when implementing it, so it is strongly recommended to add this to any puzzle game, but the gameplay needs to be able to support it and any new features procedural generation brings.

7. Conclusion

The goal of the project was to examine the advantages and disadvantages of procedurally generated levels and to see if it was possible to expand the game *A Mazing Monk* with a procedural level generator. Based on our research we conclude that solvability, variety and difficulty ratings are key requirements that should be met when generating levels for puzzle games. Some of the disadvantages that needed to be catered for were the aesthetics of the game and the extra load on the programmers, which is the real reason why this is not used that heavily in the industry, especially in AAA games. However, in smaller casual games with distinctive and simple gameplay it is relatively easy to implement.

To meet the goals we implemented a solver to verify our generated levels and a level generator capable of producing levels with an approximated difficulty. The generated levels were then stored in level files and used by the new version of *A Mazing Monk* in two new game modes: Campaign mode with a fixed difficulty increase and Infinite mode with variable difficulty. The success of the implementation hinges on performance and acceptance among the test subjects.

The game designer of the original game performed the test on game play and aesthetics of the new version and his comments constitutes the user testing. Time did not allow us to perform additional user tests but this should be done in order to ensure that the difficulty ratings are accurate and that the levels are varied enough.

The performance was measured by analyzing the results of the generator and solver. From the tests, it was decided that a dedicated machine should be used for the level generation and verification using the current implementation. Generating and verifying a level of the hardest difficulty can take up to 20 seconds on a powerful machine by standards of today; on a mobile device with limited processing power this would take even longer. The levels can however be stored to a file and regenerated on the client machine at reasonable costs.

We therefore feel that the goals of the project have been met, but with room for improvements. Additional tests should also be performed on a larger group of test subjects and machines of different specifications, to fine tune the current implementation.

7.1 Future Work

Here is a list of possible extensions and improvements to the implementation.

Performing additional tests

Additional testing is needed to fine tune the difficulty rating system and the hardware specifications needed to run the game.

Improving the solver

In its current implementation, the solver generates new states from each rotation that can legally be performed on a given cube. This approach ensures that every possible combination of the cube is examined, but its performance is unsatisfactory in some instances. Another approach that could be explored in future versions is a more objective based solver. Rather than looking at rotations from the top face only, the solver could examine how the current objective of a given state can be achieved based on the entire cube.

If a state is in the process of collecting karma for example, finding every karma piece on the cube the solver could generate new states from every course of action that makes a karma piece reachable for the monk. Similarly, when a state has collected all karma and is looking for a path to the end square, finding every corner square on the cube with a path cubie, the solver could generate new states from every course of action that places it in position (2, 2) on the top face and is reachable for the monk.

The current implementation can also be further optimized. We tried several different search heuristics during the test phase and picked the one that on average solved a cube fastest. Other heuristics yielded faster results in some cases, but slower in others. If we could determine when a heuristic works better, the algorithm could analyze the unprocessed states and based on that pick the best heuristic dynamically.

Improving the generator

When the current level generator is finished placing karma and path cubies, it rotates the cube and moves the monk randomly until a path to the start square is available. If this takes more than a fixed amount of tries, the cube is discarded, so it is luck-based whether this will succeed. To improve this, a reversed version of the solver could be implemented, as it ensures that every possible combination is tried and only once.

In combination with adding, a solver to some of the elements in the generation the ultimate goal would be to create a generator that always generates solvable levels, making the solver unnecessary in the generation process and only used when the player needs help at solving a level.

8. Bibliography

1. A Mazing Monk. [Online] June 2009. http://www.amazingmonk.com.

2. **Parish, Jeremy.** Essential 50 - 12. Rogue. [Online] 1UP, 2007. [Cited: 10 15, 2010.] http://www.1up.com/do/feature?cId=3123537.

3. Speed Tree. [Online] http://www.speedtree.com.

4. .kkrieger. *.theprodukkt.* [Online] [Cited: October 14, 2010.] http://www.theprodukkt.com/kkrieger.

5. **Archer, Aaron F.** A Modern Treatment of the 15 Puzzle. *http://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/15-puzzle.pdf.* [Online] November 1999.

6. **Rokicki, Thomas, et al.** God's Number is 20. *http://cube20.org/.* [Online] July 2010.

7. Graph Coloring. *Wikipedia.* [Online] [Cited: October 14, 2010.] http://en.wikipedia.org/wiki/Graph_coloring .

8. Solving Techniques. *Sadman Software.* [Online] [Cited: October 14, 2010.] http://www.sadmansoftware.com/sudoku/solvingtechniques.htm.

9. **West, Mick.** Random scattering. *Gamasutra.* [Online] August 2008. http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php.

10. List of teleological and ontogenetic methods. *PCG Wiki.* [Online] http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic .

11. **Langton, Christopher G.** Artificial Life. [Online] 2009. http://www.probelog.com/texts/Langton_al.pdf .

12. **Macal, Charles M. and North, Michael J.** TUTORIAL ON AGENT-BASED MODELING AND SIMULATION PART 2: HOW TO MODEL WITH AGENTS. [Online] 2006. http://www.informs-sim.org/wsc06papers/008.pdf.

13. **Olsen, Jacob.** Realtime Procedural Terrain Generation. [Online] October 31, 2004. http://oddlabs.com/download/terrain_generation.pdf.

14. **Wolfram, Stephen.** Cellular Automata as Models of Complexity. [Online] 1984. http://www.stephenwolfram.com/publications/articles/ca/84-cellular/3/text.html.

15. Introduction to genetic algorithms. *obitko.com.* [Online] [Cited: october 14, 2010.] http://www.obitko.com/tutorials/genetic-algorithms/introduction.php.

16. **Watcharasatharpornpong, Nirach and Kotrajaras, Vishnu.** Automatic Level Difficulty Adjustment in Platform Games Using Genetic Algorithm Based Methodology. *Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.* [Online] [Cited: October 14, 2010.]

17. Introduction to Markov Chains. *Dartmouth College.* [Online] [Cited: october 14, 2010.] http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/Chapter11.pdf.

18. Naturally Occurring Fractals. *MIQEL.com.* [Online] [Cited: October 14, 2010.]

19. Mandelbrot Set. *MIQEL.com.* [Online] [Cited: October 14, 2010.] http://www.miqel.com/fractals_math_patterns/visual-math-mandelbrot-magic.html.

20. **Perlin, Ken.** Making Noise. [Online] December 9, 1999. http://www.noisemachine.com/talk1/index.html.

21. **Jessop, Robert.** L-Systems Applet. [Online] [Cited: October 14, 2010.] http://home.clara.net/niknak/fractal/.

22. **Moler, Cleve.** TicTacToe Magic. [Online] 2009. http://www.mathworks.com/moler/exm/chapters/tictactoe.pdf.

23. **Booth, Michael.** The AI Systems of Left 4 Dead. [Online] 2009. [Cited: October 14, 2010.] http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf.

24. Dijkstra's algorithm. *Wikipedia.* [Online] [Cited: October 14, 2010.] http://en.wikipedia.org/wiki/Dijkstra's_algorithm .

25. **Matsumoto, M and Nishimura, T.** Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation 8.* 1998, pp. 3-30.