Procedurally Generated Content for Mobile Games
Segment-based online level generation

Author: Bartomiej Siemieniuk
Supervisor: Dr. Ke Chen

Third Year Project
School of Computer Science
The University of Manchester
May 9, 2016
Abstract

The aim of this project is to explore online procedural content generation (PCG) techniques based on the latest research in the field. The project consists of two main parts, the creation of the level generator and the development of a 2-D video game which demonstrates the capabilities of the generator on a mobile device.

Based on a review of current state-of-art level generators and techniques, this project focuses on an implementation of a segment-based online generator with rule-based methods to address quality assurance and provide controllable game segments. It discusses the performance and evaluation of the final products and explores approaches for improvement of both, the performance and the quality of generated content.
## Contents

1 Introduction ................................. 4  
  1.1 Overview & Motivation .................... 4  
  1.2 Project Goals ............................. 5  

2 Background ................................. 6  
  2.1 Video Games ............................... 6  
  2.2 Procedural content generation .......... 8  
    2.2.1 Classification ....................... 8  
    2.2.2 Verification ........................... 9  

3 Methodology ................................ 10  
  3.1 Approach .................................. 10  
  3.2 Content Space Representation .......... 10  
  3.3 Evaluation Rules .......................... 11  

4 Project Design ............................. 13  
  4.1 System Overview ........................... 13  
  4.2 The Game ................................. 13  
    4.2.1 User Interface ....................... 14  
    4.2.2 Game Engine .......................... 14  
    4.2.3 Game Loop ............................ 14  
    4.2.4 Component-Entity Pattern .......... 15  
    4.2.5 Scene Graph .......................... 16  
    4.2.6 Segments Generation ................. 16  
    4.2.7 Optimisation .......................... 17  
  4.3 The Level Generator ........................ 17  
  4.4 Supportive Tools .......................... 17  

5 Results .................................. 18  
  5.1 The Game ................................. 18  
  5.2 Level Generator ........................... 18
List of Figures

2.1 Screenshot of Wolfenstein 3D, one of the first 3D games, released for MS-DOS in 1992 ................................. 7
2.2 Screenshot of The Witcher 3 demonstrating the amount of details and complexity of the game. .................. 7
3.1 Part of a level in Broforce showing a grid representation of the design elements. ............................................. 11
4.1 Steps in the SpriteKit game loop.[7] ................................. 14
4.2 "Problems Evolving An Inheritance-Based Design loop."[9] ........................................... 15
4.3 "Entity-Component Design Composes Functionality",[9] ........................................... 15
4.4 Parent-child node relation. [8] ........................................... 16
5.1 Generated levels with different leniency: A represents low leniency and B great leniency. ......................... 20
5.2 The level A represents low linearity and the level B high linearity . 20
Chapter 1

Introduction

One could perceive the meaning of randomness in two ways. This chapter introduces the problem domain, proposed solutions, and deliverables of the project.

1.1 Overview & Motivation

The growing number of titles on the market and the rising capabilities of modern hardware drive increasing customer expectations. Traditional approaches to game content construction and creation results in high production costs due to the required effort of many professionals over many years.

A recently released title “The Witcher 3” required 240 experts and about 3.5 years to be created. The production consumed around 306 million Polish Zoty (approximately 81 million USD).[14] Grand Theft Auto 5, another AAA production, made by a formerly Scottish studio, cost 170m million USD.[12] Present production costs make it difficult for new studios to enter the market.

The increase in smartphone performance now allows for the development of complex and impressive mobile games; often the same game is released on both personal computers and mobile. [5][4] However, the amount of content created by various artists results in a growth in the size of the applications. Modern personal computers have no issues with storage limitation, however in the world of mobile devices, storage capacity is still highly limited. One of the most popular smartphone on the market, the iPhone[3], is mainly (91%) sold in its 16GB or smaller edition. [11] Moreover, operating systems share the same storage resulting in even lower available space. The previously mentioned game Witcher requires 40GB of space, “Diablo III”, one the most successful titles developed by Blizzard requires 25GB of storage without an expansion pack. Available space is a substantial limitation for the development of complex games targeted for mobile devices.

A widely used solution to these problems is Procedural Content Generation. “Procedural content generation (PCG) in games refers to the creation of game content automatically using algorithms.”[19]. Initially used to overcome mem-
ory limitation of former computers, today PCG aids development by rapidly generating a large amount of content. Over the years, various techniques and approach towards procedural generation were created, and this project tries to explore them.

1.2 Project Goals

This project sets out two goals: first to create a game designed for iOS smartphones; second to explore PCG techniques and create an online algorithmic level generator which will produce high-quality content. The following deliverables are required to make this project successful:

- 2-D side scrolling game for iOS smartphones
- online segment-based level generator

The game should be simple enough so that the development can be carried out during the time of the project. However, it needs to provide enough variety in design elements which would be attractive to the development of the level generation. This project will create a simple 2-D platformer, in which levels have infinite length. The player’s goal will be to guide an avatar through a diverse set of obstacles, including pits, platforms and enemies. As the avatar advances further into the game, the player receives a higher score. Collecting coins or defeat enemies could gather additional points.

The level generator needs to have sufficient performance to create high-quality short segments during gameplay using a modern smartphone CPU. Produced segments will contain design elements as presented in the game description. Evaluation rules verify the quality of generated building blocks; together, multiple high-quality segments compose the final level.
Chapter 2

Background

This chapter presents background information on the two distinct parts of the project:

1. Creation of the simple video game
2. Development of the level generator

Although these are not directly relevant, they are important for understanding the project’s context.

2.1 Video Games

In 2013, the game industry made twice as much money as the film industry. Games have become an important part of the entertainment and world economy. However, it all started very simply, with basic text-based application nothing like today’s productions.

On May 5, 1992, Wolfenstein 3D, one of the early 3D games was released. Its great success resulted in establishing the model of subsequent first-person shooters (FPS).

Today, 24 years later, video games come in all shapes and forms. From real-time strategies (RTS), through massively multiplayer online role-playing (MMORPG), where millions of players connect to experience exciting stories, create organizations and conquer virtual worlds, to ridiculous productions like "Goat Simulator".

Modern productions feature large, open worlds with non-linear stories. Games are not only programs, but they connect various other fields like art, music, and artificial intelligence. They are the form of art themselves, allowing people to create and experience stories in a way books or films never could, and at the same time pushing engineering to its limits.

Modern computers owe their current computational power and capabilities to the video game industry. A good example of this is video cards and 3D graphic
 accelerators, which were made to aid games and UI rendering. Nowadays, they are not only used for rendering but to support expensive computation like deep learning.
2.2 Procedural content generation

Procedural content generation (PCG) refers to creating game content automatically, through algorithmic means[18]. The term content is widely understood and could relate to such aspects as animations, levels, quests, characters or textures and sounds. [18]

The important concept to understand is that PCG does not mean random. This could be perceived in two ways. One could understand PCG as a deterministic process - given model, as an example a mathematical function, it always produces the same output based on the particular input. For an example in upcoming production, No Man’s Sky is a game that contains over 18 quintillion \((1.8 \times 10^{19})\) planets, and none of them are stored permanently. Instead they are generated on demand with the use of layers of mathematical formulas. [YC] Visiting the same planet always yields the same results. Deterministic generation is of particular importance in multiplayer online games where all the players need to have the same content. However, not all of the generator must be deterministic. It is more important to view PCG as a process which outputs a high-quality content in well-defined and structured manner. The generated content must make sense in some way. That could be defined by whether there is a path from the starting point to the end or whether created content fits the rest of the game in visual ways.

This section presents several different ways of classifying PCG algorithms and discusses possible approaches towards verification of the output.

2.2.1 Classification

PCG Algorithms could be distinct in various ways. This section highlights the ones required to understand this project entirely.

Online vs. Offline

Online refers to the generation which occurs during the gameplay, and offline to the generation taking place during the development of the game. [18]. No Man’s Sky provides an excellent example of the online generation, the space available in the game is simply too large to persist, and the player’s space coordinates expand into the actual game content during the runtime. For comparison offline could be used if all the planets would generate during the development, possibly adjusted by designers, and would be distributed along with the game.

Constructive vs. Generate-and-Test

Generate-and-test is an approach in which content is generated and then evaluated against a set of defined criteria; content which violates the rules could be discarded or partially regenerated. Algorithms loop until an acceptable solution is found. In contrast, constructive rules create the content “[...] only by performing operations, or sequences of operations, that are guaranteed never to produce broken content.”[18]
2.2.2 Verification

Required content has to be of the highest quality. Otherwise, the created content could be frustrating for the player or lose realism, in the worst case, the content would be non-playable. There exist different approaches used to verify generated content.

Evaluation Functions

Togelius [18] presented evaluation functions as a deductive method of content verification. Evaluation functions could be boolean or numerical; the boolean function would approve or discard content where the latter returns how much tested content violets it. Presented functions are:

1. Direct Evaluation in which the function directly evaluates features extracted from the generated content (e.g., number of obstacles, the size of the map, etc.)

2. Simulation-based Evaluation in which agent (bot) attempts to complete the game and various statics are collected then evaluated (e.g., number of moves, time or tries required to complete the level).

3. Interactive Evaluation The function evaluates content based on iteration with a player during the runtime by collecting various data, e.g., by measuring the time player interacts with the particular piece of content. [18]
Chapter 3

Methodology

This section describes methods and algorithms used for content creation.

3.1 Approach

The methodology used in this project was inspired by recent work in the field [16], and focuses on the exploration of short-game segments instead of the complete level.

The method used here is a mixed approach of Generate-and-Test, and Constructive techniques. The content is first generated and then verified against a set of rules. If the rules are violated, the content is discarded, and the process is repeated until a fitting piece of content is found. The complete level is composed of multiple segments that survived verification process; those blocks are called Constructive Primitives. As a result, this technique allows for an efficient online generation. It is possible to control generated segments by adjusting different properties of the generator like the number of obstacles, number and size of gaps and height of platforms.

3.2 Content Space Representation

Usually, in 2-D games, each level is represented as a 2-d grid, e.g., 200x100 units. Direct encoding - an attempt to generate content to for each of cells individually - is not desirable as it produces high dimensional content space vector (genotype) which is not suitable for many algorithms. Additionally, this results in high locality meaning that a small change to the genotype would result in a minor change to the generated level (phenotype) [18].

Motivated by previous work[16], the level is described using a list of element designs, where each of design elements is represented as an atomic unit. The design element is defined using several values, e.g., x and y position, type or width. This led to a very concise representation of the content space, keeping the number of features small - 100. This approach has the advantage that
increasing the size of the level does not affect of the dimension of the content space vector. As well as changes to the genotype result in significant variation of the phenotype.

3.3 Evaluation Rules

Many of the generated segments contain conflicting design elements or an undesirable combination of them. For example, a crate could be only placed on the part of the building block which does not contain a gap. To overcome this issue, a version of the constraints presented in [17] were adopted. This method of quality assurance is deductive, which introduces several issues. Most importantly, developers of the game have to be aware all the features present in the content space and their relations to be able to encode constraints into the evaluation functions. Inductive methods such as a recently developed Learning-Based PCG framework [16] which learns from data could support quality assurance. The following constraints were adopted:

1. \texttt{doTwoColide} This constraint checks whether two elements collide with each other.

2. \texttt{doOverlapVertically} This function checks whether two elements overlap vertically.

3. \texttt{spaceOverlap} This function check if all supplied design elements can fit in given space without colliding with each other or design elements that have already been listed in the space.
4. **doOverlapItself** Discard segment if any of the array of element overlap with another element.
Chapter 4

Project Design

This chapter presents details on the game; its requirements and design of the level generator and other supporting tools required to develop a successful project and the reasoning behind particular design decisions.

4.1 System Overview

The project contains three different subsystems:

1. iOS game
2. The level generator
3. Supporting tools

Supportive tools assist the development of the level generator, allowing for visual inspection of the content produced by the level generator. The level generator itself was a standalone library and connected to the game with the output of the generator being translated into the actual game content using a dedicated realizator.

4.2 The Game

This section describes the choice of a game engine and the construction of its main loop. This is followed by reasoning about the use of a component-based architecture to model games entities. It is explained how game content is structured and how the generation of segments is handled with some optimisations applied.
4.2.1 User Interface

The game’s interface was designed to be minimalistic, not to have any explicit UI elements which controls the character following the design language from many popular titles.

4.2.2 Game Engine

Several game engines are available that allow for development on iOS. The most features-rich is Unity3D supporting cross-platform development. However, it’s too complex for a project of this size. It does not give direct control of the device nor does it expose the native API. Unity is also a 3D engine which is not suitable for this project. A smaller and more low-level engine fits the requirements of this project better. However, as the game levels will have to be able to be previewed on the computer as well as during the game, a cross-platform solution is needed. The best fit for these requirements is provided by Apple’s SpriteKit 2-D [7] engine, which runs on both Macintosh computers and iOS devices. This allows for reusable component and reduced development effort. SpriteKit provides an animation and rendering engine, physics simulation, and collisions system.

4.2.3 Game Loop

The fundamental element of the game is the game loop. A game loop is a critical point, divided into multiple actions such as receiving user input, updating entities position, updating physics and checking for collisions.

![Figure 4.1: Steps in the SpriteKit game loop.][7]
4.2.4 Component-Entity Pattern

Entities often share particular characteristics which suggest that inheritance is an excellent choice to provide reusable code. However, when complexity increases, entities features often conflict with each other making an inheritance-based approach over complicated. Component-based architecture is a widely used solution to the presented issue. [13]

Figure 4.2: "Problems Evolving An Inheritance-Based Design loop."[9]

Component based architecture favours composition over inheritance. The entity is a base object which stores all the components assigned to it. Previous functionality is decoupled from the object into separate reusable components classes like rendering, movement, and attack for an example.

Figure 4.3: "Entity-Component Design Composes Functionality".[9]
4.2.5 Scene Graph

A scene in the game is represented as a tree; scene is the root of the tree having entities attached to it as child nodes. This kind of representation is particular useful when dealing with complex objects. Nodes help in organising the scene and make applying affine transformations easier, as well as deleting or running actions on nodes.

All segments produced by the segment generator are represented as a node. All the elements were children of the segment node except enemies, which have freedom of movement and can travel between building blocks. Thus, there was a need to create additional children in the scene graph to hold enemies only. This allows for easy deletion of segments and all elements that are part of them.

Figure 4.4: Parent-child node relation. [8]

4.2.6 Segments Generation

The requirement of extending levels as the player progresses required thoughtful design. The game starts with 10 initially generated segments. As the user approaches the last available piece of the level, another constructive primitive is generated and added it to the game scene.

All of this was handled by the Realizator class which abstract all of the details to the GameScene instance; it was also responsible for the communication with
the PCG engine.

**Content Vector Translation**

The generator outputs a numerical vector, which is meaningless information to the game engine. To be able to create a scene, the game engine needed to translate the vector to in-game entities. ContentVector was a class wrapping the numerical vector and providing abstracted access to the elements of the segment. Element was a basic class representing entities such as enemy, coin, chest or a platform. Element had information about height, x,y position in the segment and a type of the element it describes. Element and Vector were designed to be cross-engine. Each "Element" instance was converted to an appropriate instance of a game entity by the engine. In this case the factory pattern [6] was used to provide an instance of a node with appropriate position and its required components.

**4.2.7 Optimisation**

Due to limited resources available on smartphones, several optimisation techniques were implemented. There were only a few constructive primitive in memory at the same time. Due to the game requirements, there was no need to preserve constructive primitives already passed by the player. When a new CP was generated, the left-most building block from the game scene was removed. Additionally, all entities from the enemy layer which at that point were behind the current viewport were also deleted. This resulted in the minimum memory footprint and kept the CPU usage low, as only required elements have to be updated and rendered. Moreover, only a few nodes participated in the collision detection.

**4.3 The Level Generator**

The level generator was designed to be standalone library included into the project. Implementation details of placed components were unknown. This makes use of already presented ContentVector and Element classes. The level generator parameters could be adjusted, such as a minimum number of particular entities, minimum gap width, or minimum height of platforms.

**4.4 Supportive Tools**

The supportive tool uses the same components for rendering and integration with the level generator as the game. The graphic user interface was created with Apple Cocoa framework.
Chapter 5

Results

This chapter presents and discusses the final products of the project, including the mobile game, its interface, and gameplay, examples of the generated content, and capabilities of the supportive tool. The performance and evaluation of the created applications are discussed in the next chapter.

5.1 The Game

The game features simple gameplay. In addition to the player’s avatar running to the right, touching the left side of the screen makes the avatar jump. Jumping aids a player in avoiding obstacles and gaps as well as navigating between platforms. Touching the right side of the display launches a missile that moves towards the touched position. The collision of the bullet with the monsters results in their removal from the game scene and earns the player additional points. The player can score further points by collecting the coins located in various places on the level.

The bottom left side contains a pause button; the top left corner shows collected coins. The upper-right corner displays the amount of points awarded to the player.

5.2 Level Generator

The final version of the level generator can produce content with use of six different design elements in the same build block: a gap, a platform, a store, a coin, a crate and a monster. Adjusting the parameters of the level generator results in indirect controllability of the produced level. Levels could be classified based on two different properties: leniency and linearity. [16]

1. **Leniency** relates to difficulty of the produced content and it is adjusted by controlling the number of enemies, crates, as well as the size and the number of gaps.
2. **Linearity** describes how much the level resembles the straight line, it could be adjusted by specifying the number and height of the platforms.
Figure 5.1: Generated levels with different leniency: A represents low leniency and B great leniency.

Figure 5.2: The level A represents low linearity and the level B high linearity.
Chapter 6

Evaluation

This chapter discusses the techniques used for the testing and evaluation of the finished project, regarding both the quality of the generated content and the performance of the applications.

6.1 Unit Testing

Unit testing is a technique to test source code in software engineering. A unit-test is a short program which assesses one or more units at the same time against a set of expectations. Unit could refer to an entire interface, classes or individual methods.

Comprehensive unit testing was used during this project. This allowed for automated testing during development. Previously written test cases help to assure that written and refactored code still behaves as intended. This successfully led to discovering mistakes and errors which saved time and effort. Tests were created with the use of Apple’s XCTest framework. [10]

6.2 Performance

The performance of the game and content generator was tested. This section discusses the tests and devices used, and outcome of the evaluation.

6.2.1 Tests

Several tests were carried out to verify the overall performance of the application and generation algorithms. Both tests aimed to generate 1000 Constructive Primitives, twenty units wide, five units tall. The first test was to produce CPs without a minimum number of any particular design elements which had to appear in the scene. The second required at least two enemies, two gaps and one crate.
6.2.2 Testing devices

There were two distinct devices used for the evaluation:

1. MacBook Pro (Retina, 13-inch, Mid 2014) with Intel Core i5 x86-64 2.6 GHz CPU and 8 GB 1600 MHz DDR3 ram

2. iPhone 5S with 1.4 GHz ARMv8-A and 1 GB of LPDDR3 DRAM.

6.2.3 The Game Performance

On each of the devices, the two tests were run. The first test performed well, resulting in constant 60FPS, low energy impact and CPU usage between 40% and 50%. However in the second approach, the game was very often unplayable, resulting in CPU usage of around 100%. This is unacceptable for mobile devices, as such high CPU usage results in instant battery drain.

6.2.4 The Level Generation Performance

A similar approach was taken to evaluate the level generator performance - the same two tests were used.

the MacBook Pro required 0.9s in the first test, and the second took 69.96s, resulting in a mean time of 0.07s per CP which is acceptable. The iPhone in the first test required 13.18s with an average of 0.01s and 917.876s (around 15 minutes) and a mean of 0.92s in the later. However, there was a high standard deviation - 3.22s, and up to 16.37s to generate single CP maximum. Around 3,489,013 segments had to be evaluated to produce 1000 CPs. This method has too much variation in the time required to generate a single building block making this technique unsuitable. An alternative approach to this problem is presented in 7.1.

6.2.5 Mobile Game Size

Thanks to the fact that there are no levels stored on the device, the final game size was 19.8mb, which is lower than the 2012 average in the game category. [1]

6.3 Content Quality Verification

During the manual inspection, it was noticed that some of the generated game segments could be unplayable. This was usually due to there being no path between grounds or platforms reachable by the player. Often the presented building blocks had unbalanced design elements e.g. having too many enemies or platforms in comparison to the rest of the elements. To address this issue, a significant effort was carried out to adopt learning-based techniques proposed by Shi and Chen[16]. These further details are in the 7.2
Chapter 7

Discussion

This chapter discusses the issues that occurred while developing this project, states its achievements and suggests how this project can be taken further. It also provides self-reflection on the knowledge which was gained and areas for improvement.

7.1 Performance of the generation process

The evaluation identified that the Generate-and-Test approach can lead to performance issues. Rules presented earlier work well if the design elements are independent of each other; otherwise, there is a small probability that dependent elements described by random numbers will satisfy the constraints.

Multiple gameplay sessions and performed tests showed that generation of the single constructive primitive on a background thread might take up to 16s on the iPhone 5s; production of multiple CPs which take more than 10s in a row leads to catastrophic results, as the player moves through the content faster than it can be supplied. This makes the game unplayable.

The discussed issues could be resolved by removing a constraint dictating a minimum number of elements which are dependent on each other which results on average 0.05s per CP. However, motivated by subjective, visual inspection of the generated CPs, this makes them less engaging. Therefore, there is a need for an alternative approach.

The proposed solution would change the generation process to a constructive one. Instead of using evaluation functions to apply rules after the candidate for the CP was generated, the rules would be directly encoded into the generation process.

Firstly place all independent elements and based on their description create an appropriate domain for placement of the dependent elements.

For an example, one of the constraints is that the crate might only appear on solid surfaces like gaps or ground. To do that, first, generate the gaps and based on their position create a vector which would hold available coordinates
on which crates could be placed. This way it is not required to validate the coordinate chosen from this vector as all entries are already valid.

This approach would result in discarding less of the generated CPs as this way, the performed sequence of operations are guaranteed to produce valid content. This results in a lower waste of created content, reducing the time needed to generate the constructive primitive and what is most important reducing the energy impact.

7.2 Learning-Based PCG

By the end of the project, a high number of constructive primitives were mis-classified as valid despite being unappealing or untraversable.

To overcome this issue, motivated by recent research\cite{16}, a great effort was put into developing a learning-based approach towards classifying the constructive primitives.

Recent work showed that the quality of the content can be highly improved by adopting learning techniques such as Weighted Random Forest classifier which, as they are weighted, prevents a misclassification of invalid segments, in the worst case reducing the available content space.

To do so, an iterative approach was taken to develop various decision tree classifiers including Random Forest and Weighted Random Forest\cite{2}. Forests are an example of ensemble learning, connecting a number of classifiers together, usually trained on different parts of a dataset allowing for more robust classification. This is achieved by the fact that final class assigned is the majority vote of all classifiers.

Content vector space representations were used as the data points. It was required to manually annotate a large sample of the data for the training process. This was obtained by first randomly generating a large number of example levels which were then clustered using K-Means algorithm. Each cluster is then sampled with the number of samples selected being proportional to the cluster size. Clustered result were then annotated by visual inspection using the supportive tool.

As the final training set was never annotated, the classifiers were regularly evaluated with an external data set. The dataset was obtained from UCI Machine Learning Repository - HIV-1 protease cleavage data set\cite{15}. The problem of classifying this data has many similarities to the original problem in the project as this is a binary classification problem in which all the features are discrete.

Due to time constraints, many components of the system were created separately but never connected. Therefore, it is difficult to discuss the performance and evaluation of the system.
7.3 Thrid-party evaluation

Generated CPs were evaluated using visual inspection by the developers, only relying on their judgement leading to a subjective opinion. In computer games, the metric of fun is critical\[19\] . However, it is a non-trivial problem and was outside of the scope of this project. Such testing could be carried out with an appropriate survey framework on a group of test players.

7.3.1 Self-Reflection

Carrying out this project turned out to be a harder task than anticipated. Many of the things which planned were not finished or were done briefly.

However its very important for myself that I explored different parts of the technology or problems I face everyday. This was a great chance to test out some of the latest research and work with Ph.D. students interested in similar research. I am remarkably proud of the effort put into the development of this project, although I am aware of many issues or weaknesses related to it.

The project called for the development of both a game and a procedural content generator in a short period. This project should have only focused on one of them. That would have allowed for deeper research and a more finished product.

There are already many games available with source code which could be used to focus on research in Procedural Content Generation instead attempting to create a custom solution.

Great effort went into the development of assets, which were supposed to make the game appear unique as it was planned to be released later on the App Store. These should have been made after the project was successfully delivered and placeholder images used during the development.

Only one approach to content generation was considered. Ideally, more methods of generation should have been explored. This would have allowed testing and discovering more appropriate solutions, for example, rhythm-based generation of segments, which there was not enough time to explore.

7.3.2 Future Work

In addition to the simple game mechanics explained, more complex mechanics such as moving platforms, stores, health points for the avatar and explosion on contact with boxes could be implemented. Also, dynamic difficulty adjustment could be carried out to create a more personalised experience.

7.3.3 Conclusion

The project successfully delivered the mobile game, as well as a simple segment online level generator using a combination of both generate-and-test and
constructive approaches. Various classifiers were created as well as several supportive tools such as a clustering tool and an annotation application, which also allowed for the preview of the generated content.

All of the achievements resulted in a fully-working and interesting project.
Bibliography


[2] Chao Chen and Leo Breiman. “Using Random Forest to Learn Imbalanced Data”. In: ().


