POLITECNICO DI MILANO
Corso di Laurea MAGISTRALE in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

CASTLE:
Map Generation and Navigation

Relatore: Prof. Pier Luca Lanzi

Tesi di Laurea di:
Faverio Luca, matricola 771462

Anno Accademico 2011-2012

Ringraziamenti

Ringraziamenti

Vorrei ringraziare il prof. Pier Luca Lanzi per la disponibilitá e l'aiuto fornito durante lo svolgimento del lavoro di tesi. Desidero inoltre ringraziare la prof.ssa Manuela Cantoia per averci dato la possibilitá di sviluppare il video game CASTLE. Grazie inoltre ad Alessandro e Valentina per averci fornito gli indovinelli e i quiz presenti nel gioco. Vorrei inoltre ringraziare Davide e Roberto per i 5 anni passati assieme e per i pomeriggi passati a lavorare a questa tesi. Grazie anche alla mia famiglia e a tutti i miei amici per avermi sempre sostenuto in questi anni.

Contents

1 Introduction **1**

1.1 Procedural Content Generation and Video Games . 2

1.2 Path-finding and A* Search 3

1.3 CASTLE 4

2 CASTLE **5**

2.1 Gameplay 6

2.2 Quiz Types 8

3 State of the Art **11**

3.1 Procedural Content Generation 11

3.1.1 Binary Space Partitioning for Dungeon Generation 13

3.1.2 Cellular Automata for Cave-Like Structure 14

3.1.3 Lindenmayer System for Plant Generation . 16

3.1.4 Perlin Noise for Texture Creation 17

3.1.5 PCG in Commercial Application 18

3.2 The A* algorithm 20

3.2.1 Tree Search Strategy 20

3.2.2 Breadth First Algorithm 22

3.2.3 Depth First Algorithm 22
3.2.4 A* Algorithm 25

4 Implementation 27
4.1 The Cocos2d-x Framework 27
4.1.1 Framework Description and Features 28
4.2 CASTLE Implementation 32
4.2.1 Map Generation 32
4.2.2 Furniture and Quiz 34
4.2.3 Player Movements with A* 35

5 Conclusion 40
5.1 PCG: Evaluation 40
5.2 A* Search: Evaluation 41
5.3 Further Works 41

Bibliography 42

List of Figures

2.1 *World Map* 6
2.2 Reward Screen and Treasure Chest 7
2.3 Different Types of Quizzes 9
2.4 Different Types of Quizzes 10

3.1 *BSP Iteration* 13
3.2 *BSP Final result* 14
3.3 *Cellular Automata: initial state and final result* . . 15
3.4 *L-System example:* 16
3.5 *Perlin noise: Effects generation* 17
3.6 *Perlin noise: Height map generation* 17

4.1 *Cocos2d-x Logo* 27
4.2 *Cocos2d-x Architecture* 28
4.3 *An example of scene work-flow* 30
4.4 *Two Different Maps generated by the Algorithm* . 33
4.5 Different Room Furniture 34
4.6 *Triangle inequality* 37

Abstract

Procedural Content Generation (PCG) is a promising field for the design, art and production of video games, allowing real-time generation of game contents instead of relying on pre-generated ones. In this thesis, we will discuss our implementation of a procedural generation method for 2D maps and a navigation algorithm based on A*. The map generator is structured in two phases: the first one deals with the map layout while the second one with object placement. Both algorithms have been implemented in the video game CASTLE, a game developed in collaboration with Universitá Cattolica del Sacro Cuore di Milano to support research about lateral thinking.

Sommario

La Generazione Procedurale dei Contenuti (PCG) é un campo promettente per il design, la grafica e la produzione dei videogiochi, permettendo la generazione in tempo reale di contenuti, al posto di utilizzarne di pre-generati. In questa tesi, discuteremo l'implementazione di un metodo per la generazione procedurale di mappe 2D e di un algoritmo di navigazione basato sul metodo di ricerca A*. Il generatore di mappe é strutturato in due fasi: la prima si occupa del layout della mappa mentre la seconda della disposizione degli oggetti al suo interno. Entrambi gli algoritmi son stati implementati nel videogioco CASTLE, un gioco sviluppato in collaborazione con l'Universitá Cattolica del Sacro Cuore di Milano con l'obbiettivo di supportare la ricerca sul pensiero laterale.

Chapter 1

Introduction

After the release of iOS in 2007 and Android a year later, the mobile game market is continuously growing in the last few years. 8 billion dollars of the total 56 billion dollars earned from mobile application came from the games sector. In both AppStore and Google Play, in 2011 [2] game is the most successfully category. The low barrier to enter the market and limited production costs that a new company has to face are probably the principal causes of success in the mobile gaming market.CASTLE started from a proposal of Universitá Cattolica del Sacro Cuore di Milano to create a game to develop lateral thinking in players through different types of puzzles. The game has been developed in about six month using the Cocos2d-x framework and then play-tested with students. This thesis discuss the procedural generation of maps and the navigation of the player in the video game CASTLE.

1.1 Procedural Content Generation and Video Games

Procedural Content Generation[1] (PCG) consists of ad-hoc algorithms to create different types of content. PCG can create every kind of video game content in a faster way, automatically generating it in real-time. Procedural generation methods have been applied in every field, from graphics [3] and level design [6][9], to music[4] or models and character animation[13]. PCG is a promising field for video games industry. In the '80s, PCG was used for the distribution of large quantity of content in a very little space, because it was necessary to save on disk only the algorithm parameters. For example *Elite*[2] (dated 1984) offers the player over 2000 different planets to explore in a floppy-disk size. Today video games use PCG to produce large amount of content which can't be simply pre-generated by a designer, saving time during the development phase. It is the case of the creatures in *Spore* [3] or the weapons in *Borderlands* [4]. Our goal in CASTLE is to offer different content to the player at each new game. Being an exploration game, different level means an higher replay value. Since was practically impossible for us to provide such a large amount of different maps, we have implemented an algorithm composed by two different methods. In CASTLE a map is composed by rooms arranged in different layouts. The first method deals with

[1] pcg.wikidot.com
[2] www.iancgbell.clara.net/elite/
[3] www.spore.com
[4] www.borderlandsthegame.com

the generation of the layout of the level while the second with the placement of objects inside rooms.

1.2 Path-finding and A* Search

Path-finding is a critical problem for many application, including video games. Given a map, path-finding objective is to calculate the best cost to travel from one point to another. To achieve this,a grid is superimposed over a region and graph search algorithms are used. One of the most used is A*[14]. First described in 1968 by Hart, Nilsson and Raphael it is an algorithm used to navigate graph in a more efficient way using a heuristic function. Firstly used for mobile robot navigation[10], it has been used for unit path-finding in video games, especially in strategy games, only in recent years[16]. In our game we needed a way to navigate rooms, from a starting point to a goal point, in an efficient way as well as taking into account obstacles. A* is the most suited algorithm for this task. The heuristic function makes the A* performs better than a standard tree search, while ensuring the optimality of the solution. In our case the Manhattan distance has been chosen considering the matrix representation of the room.

1.3 CASTLE

CASTLE is a 2D multi-platform, dungeon crawling video game developed with Cocos2d-x in collaboration with Universitá Cattolica del Sacro Cuore di Milano. Maps are generated through procedural methods and navigation through A*. The goal of the project is to support research about lateral thinking. A mechanism for sessions recording has been implemented to provide information about how video game can train lateral thinking and how different platform can effect players performance.

Chapter 2

CASTLE

Creative Activities Strengthening Thoughtful Lateral Experiences (CASTLE) is a 2D multi-platform video game developed for a mid-school audience in association with Universitá Cattolica del Sacro Cuore. It aims to provide a tool to research lateral thinking, i.e. the ability to solve problems through an indirect and creative approach. CASTLE is a dungeon-crawling video game, but instead of battling monsters the player is called to solve different types of quizzes to obtain access to new areas and challenges. Our hero is a student who is called by the king of a realm, cursed by an evil sorcerer, which has left all of its inhabitants without creativity. Our task is to defeat the sorcerer solving the quizzes on our path.

2.1 Gameplay

When the game start the player is taken to the main menu, where can choose to start a new game, load a saved one or change the game options. When starting a new game the player has to provide his name and sex, after that the world map is shown, as in Figure 2.1. From here we can decide from which castle our adventure will start. There are four castles: Three of them consist of twenty-five rooms and are available from the beginning while the forth castle consists of five rooms, is available only after the player collects all the letters needed to compose the key for it. During our adventure we will encounter different types of quizzes,

Figure 2.1: *World Map*

divided in three main categories: verbal,visual and spatial quizzes. In verbal quizzes the layer is asked to answer a question, like in a riddle. Visual quizzes require interaction with an image to find a

(a) Reward Screen (b) Treasure Chest

Figure 2.2: Reward Screen and Treasure Chest

solution. In the last case, the player has to rearrange objects in a target structure. Reward for solving quizzes is a certain amount of "creativity" based on how many hints have been used for solving it. Quizzes may give keys as reward, which can open closed areas or treasure chest scattered around the castles. Figure 2.2a shows the reward screen when a key is obtained, while Figure 2.2b one of the treasure chest. It is not necessary to solve all the quizzes, only a part of them are required to proceed with the story.

2.2 Quiz Types

There are seven types of quizzes in CASTLE: *riddle, associations, sequence, detail ,ambiguous image, find it and matchstick.*

RIDDLE

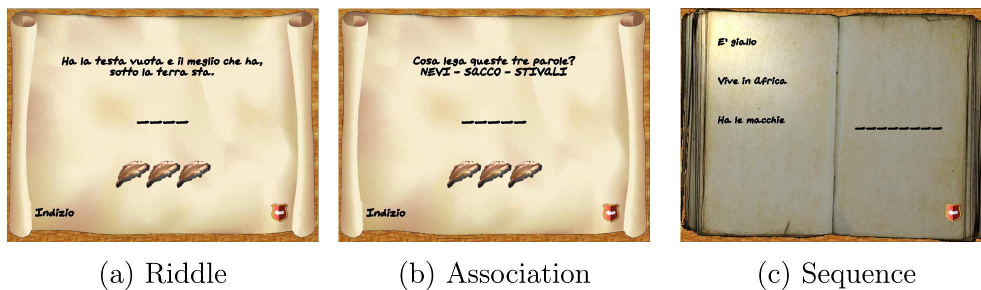
The player has to provide an unique answer to it. For each riddle the player has three attempts. Once finished the quiz cannot be tried again. The player can also receiving hints using the hint button. The number of hints depend on the length of the solution.

ASSOCIATION

Similar to the riddle, instead of a sentence we have three words and we have to guess what word connect them like in Figure 2.3b. Hint and attempts rule are the same as the riddle.

SEQUENCE

In this type of quiz the player has to guess the solution starting from a short description of it (Figure 2.3c). If the player fails another sentence will appear, but the creativity gained is lower. If the player fails five times the quiz cannot be tried again.



(a) Riddle (b) Association (c) Sequence

Figure 2.3: Different Types of Quizzes

DETAIL

A detail of an object is show and the player has to guess what it is (Figure 2.4a). If he fails, another one will appear and creativity is scaled down. After three attempts the quiz cannot be tried again.

AMBIGUOUS IMAGE

Ambiguous image are optical illusion which exploit graphical similarities to generate an image being able to provide different, but stable perception, as in Figure 2.4b. One of the most famous is Rubin's Vase. An ambiguous image is shown and the player has to find the two possible interpretation of it. There are no limit of attempts in this quiz.

FIND IT

The player has to find all occurrences of a specific object in an image full of other objects. Figure 2.4c is an example of it. There are no hint and the player has unlimited attempts, but after a certain amount of tries the quiz start over.

(a) Detail (b) Ambiguous Image

(c) Find It (d) Matchstick

Figure 2.4: Different Types of Quizzes

MATCHSTICK

Matchstick puzzles implies the re-organization of the initial configuration in one of a different type, moving a fixed number of sticks. Tries are unlimited but there are no hint. It is also possible to reset the quiz to the initial configuration. Figure 2.4d show a possible starting position.

Chapter 3

State of the Art

In this chapter, we overview Procedural Content Generation and A*.

3.1 Procedural Content Generation

In this chapter we will talk about Procedural content generation (PCG). Because it can be achieved in different ways and considering the amplitude of the topic, we will only see some example of algorithm actually used for generation of different contents. Procedural content generation refers to the programmatic generation of content using random or pseudo-random (random function initialized with a seed) algorithm rather than create them manually. This results in an unpredictable range of content automatically generated directly by the software. PCG is often used in video game industry: with it different types of game assets (maps, textures, sounds, models, animations, etc.) can be created in a quicker way. PCG has also two more benefits for game developer:

it allows the game to react to player choices in real-time in ways otherwise impossible and allows the developer to reduce the space on disk for contents, generating them real-time.

3.1.1 Binary Space Partitioning for Dungeon Generation

This algorithm generates dungeons (maps composed by room connected by corridors) using the Binary Space Partitioning method[17]. Binary Space Partitioning (BSP) is a method studied for recursively subdividing a space into convex sets using hyperplanes. This subdivision is usually represented by a tree structure known as BSP tree. For our dungeon generation we start with a maps filled with only wall cells. First we choose a random starting point and a random direction for splitting, then, using BSP, the map is split in several different sub-rooms (still filled with wall cells). Our dungeon is saved in a BSP tree and each room is in a leaf of the tree. Figure 3.1 shows the first two iterations of a BSP algorithm: For each iteration the generated dungeon is on the left and his tree representation on the right.

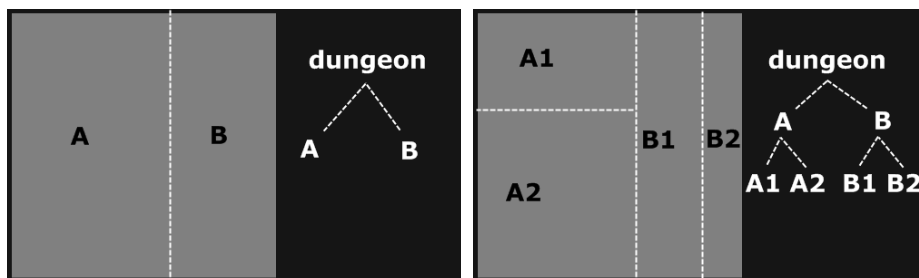


Figure 3.1: *BSP Iteration*

When the split is complete we start to carve a room in each leaf of the tree. For corridor generation we simply connect each leaf of the tree (our room) to her sisters, then, we do the same recursively for each parent node. When the root is reached, all

rooms will be connected to each other.

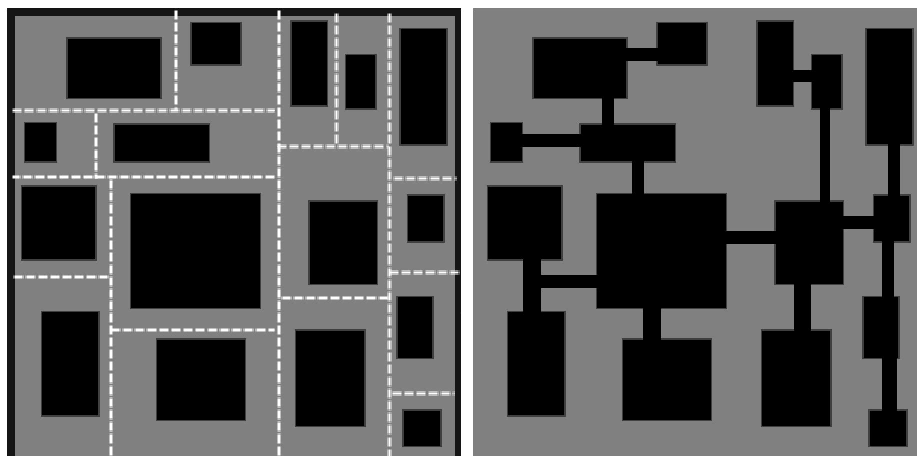


Figure 3.2: *BSP Final result*

Figure 3.2 shows the final result: on the left, all rooms have been created from the tree leafs. On the right, rooms have been connected and the dungeon is finally completed.

3.1.2 Cellular Automata for Cave-Like Structure

Another example of PCG algorithm used for generating maps is Cellular Automata[5] which can be used for generating cave-like structure[7]. Cellular Automata consists of a grid of cells each in one of a finite number of states (e.g. on and off). For each cell we define his cell's neighbourhood (usually it includes the cell itself) and a function for change its state based on the state of the neighbourhood. In the beginning the grid is initialized assigning a state to every cell. For each iteration, the state of cells is calculated based on the function and on his neighbourhood. Usually the update function is unique and is applied to all cells at the same time. Common 2D cellular automata examples are

Conway's Game of Life[1] and *Wireworld*[2]. Both are Zero-player games, meaning that the evolution of the system requires no other input other than the initial state. Cellular automata can be used in PCG for generating cave-like structure. Starting from a grid in which cells can either be a wall or empty space the algorithm evolves using the following rule: for each cell consider a 3x3 matrix centred on it. If the region contains at least 5 walls the cell will become a wall itself, otherwise it will be empty space. Each iteration makes the cells more like their neighbours and reduce the overall noise. Figure 3.3 shows an example of a random starting state and the final result. The rule can be tuned and changed for generate more realistic and appealing maps.

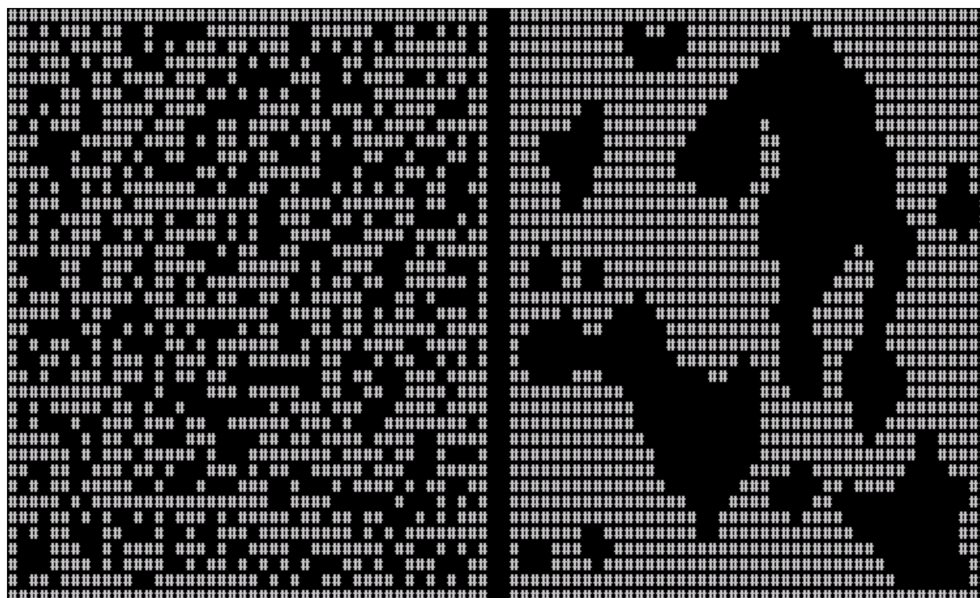


Figure 3.3: *Cellular Automata: initial state and final result*

[1]en.wikipedia.org/wiki/Conway's_Game_of_Life
[2]en.wikipedia.org/wiki/Wireworld

Figure 3.4: *L-System example:*
Variables: X F
Costants: $+ -$ []
Start: X
Rules:
$(X \rightarrow F-[[X]+X]+F[+FX]-X)$,
$(F \rightarrow FF)$
Where + means turn right 25°, - turn left 25°, F draw forward, X control the evolution of the system and [] respectively save and load a state.

3.1.3 Lindenmayer System for Plant Generation

L-System (Lindenmayer system, from the name of its inventor, the biologist Aristid Lindenmayer)[11] is a variant of a formal grammar composed by an alphabet, a collection of production rules, used to expand symbols into strings or other symbols, a starting axioms and a function to translate the generate string into a geometrical structure. Starting from the axioms the grammar rules are applied recursively, applying the maximum numbers of rules in the same iteration. This is the main difference between an L-system and a formal grammar: applying only one rule at iteration generate a language, not an L-System. L-Systems are widely used for generating plants and tree in a procedural way or to simulate their growth in a more realistic way. Thanks to its recursive nature which leads to self-similarity, fractal can also easily be described with an L-system. Figure 3.4 shows a plant generated by an L-System and its mathematical description

3.1.4 Perlin Noise for Texture Creation

Perlin noise[12] gets his name from his creator Dr. Ken Perlin. The basic idea behind the Perlin noise is very simple: first we generate a number of arrays, called octave, each one containing a coherent noise (a noise without discontinuities). The noise is usually obtained from the linear interpolation of the points derived by a seeded random number generator. Then all octaves are blended together, generating the final image. The main application of Perlin noise is textures for natural effects like smoke, fire, water or clouds. Figure 3.5 shows an example of it. It is also used for different task like generation of height maps for mesh generation(e.g. Figure 3.6) or objects placement on a grid.

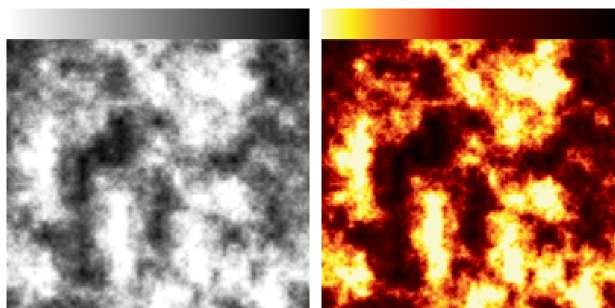


Figure 3.5: *Perlin noise: Effects generation*

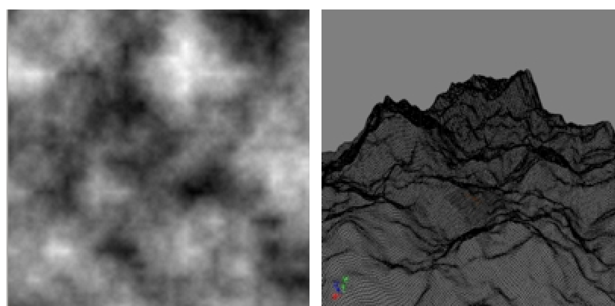


Figure 3.6: *Perlin noise: Height map generation*

3.1.5 PCG in Commercial Application

PCG have a lot of success in the video game industry allowing developers to produce a big amount of contents in a very short time. One of the first video game using PCG is *Rougue!* [3] dated 1980. This ASCII video game dynamically generated dungeon later populated with pre-created potions, weapons and enemies. During the years many other games followed the approach of *Rougue!*, probably the most famous is Blizzard's *Diablo* [4] series which have pseudo-random generated dungeons. Other examples are: *Dungeon Siege I*[5] and its sequel [6], *Angband*[7], *Moria*[8] and all the games which fell onto the classification of rougue-like. More recently *Edmund Mcmillen's The Binding of Isaac*[9], a 2D- dungeon crawling game, shows use of PCG for almost every aspect of the game, from the generation of maps to the placement of items, enemies and bonuses. PCG is not only limited at the generation of dungeons but can be used for generating a whole world or even planets like *Electronics Arts Starfligth*[10] (1986) or *Ian Bell and David Braben Elite* [11], two space-exploring game where each space system and planet is generated through PCG.

[3] wikipedia.org/wiki/Rogue_(video_game)
[4] us.blizzard.com/en-us/games/d2/
[5] www.microsoft.com/games/dungeonsiege/
[6] www.gaspowered.com/ds2/
[7] en.wikipedia.org/wiki/Angband_(video_game)
[8] en.wikipedia.org/wiki/Moria_(video_game)
[9] en.wikipedia.org/wiki/The_Binding_of_Isaac_(video_game)
[10] en.wikipedia.org/wiki/Starflight
[11] www.iancgbell.clara.net/elite/

More recent games like *Just Cause*[12] and *Borderlands*[13] uses PCG to generate the whole map and, in *Borderlands*, every weapons in the game. Not only games use PCG but exist also frameworks based on it. *SpeedTree*[14] is one of the most famous for procedural generation and placement of plants. It is currently being used in more than 70 AAA games.

[12]www.justcause.com
[13]www.borderlandsthegame.com
[14]www.speedtree.com

3.2 The A* algorithm

In this chapter we describe the A* algorithm. A* is an algorithm widely used for path-finding or graph/tree search. Unlike similar algorithm, A* reaches better performance using a heuristic function.In this way is able to calculate a path faster than other uninformed strategy approaches like Breadth first or Depth first. Before considering A* we have to understand how search strategies, like Breadth First or Depth First, work and then the improvement made by A* to this algorithms.

3.2.1 Tree Search Strategy

To create a search strategy first we need a model to define our problem. In Artificial Intelligent field one of the most common is the state space meta-model. In a more formal way, the state space meta-model is defined by:

- $S = \langle s', s'', \ldots, s_n \rangle$ the non-empty set of states.
- $A = \langle ', a'', \ldots, a_n \rangle$ the non-empty set of actions.
- `is_applicable(s,a)` a binary function stating if action a is applicable in state s.
- `result(s,a)`$= s'$ a function which return the result state s' given a state s and an applicable action a.
- `cost(s,a,s')`a non-negative function stating the cost of taking action a in state s to go in state s'.

to formalize a problem we also need to define:

- $s_0 \in S$ The initial state
- $G \subseteq S$ the subset of goal states.

Starting from these definitions, every search strategy to work need to build a search tree where every node represent a state $s \in S$. The root of the tree is the starting state s_0 while the goal states G are in the leaves of the tree. Connections between nodes represent the actions A and are generated using the `is_applicable` and `result` functions.
A Search strategy can be a tree or a graph search, can be parallel or sequential and can be informed or uninformed. In a Tree search the algorithms does not check if the state represented by the node as already been processed instead, in a graph search, this check is done and appropriate actions are made. A parallel strategy carry on the search in all possible directions, while a sequential one only follows one direction and resume the search along other directions only when it is required. Every strategy must have access at least at the description of the problem. If it does not rely on other information then is an uninformed strategy otherwise is called informed. The informations available to an informed strategy are not used to optimize the solution itself but for optimizing the process of finding a solution. Breadth First and Depth First Algorithm where both used for tree/graph search but, unlike A*, they both lack of an Heuristic function

3.2.2 Breadth First Algorithm

A Breadth first strategy (BF Strategy) generates all node at level K before generates node at level $K + 1$. It is a parallel uninformed strategy and can be implemented for both graph and tree search.The breadth first algorithm is complete because any solution has some finite length d thus, the solution will be discovered when the algorithm expands the level d. Breadth first is also optimal for uniform cost problems. Optimal solution is the shortest one and breadth first return always the shortest solution. This is not true if the actions cost are not identical. The uniform cost strategy (UC strategy) is a modification of the BF which is optimal even if the costs are not identical. To obtain this result UC expand node not based on the level (like BF) but based on the sum of cost of the action from the root to the node, in a non-decreasing order. In this way the first solution returned by the algorithm is the optimal one. If the costs are identical the UC produces the same tree of the BF.

3.2.3 Depth First Algorithm

A Depth First strategy (DF Strategy) expands the most recently generated node first. This strategy is uninformed, sequential and, like BF, can be implemented for both graph and tree search. This strategy is complete because if the tree has no infinite branches it has a maximum length m and so the solution will always be discovered. Unlike BF, DF is not optimal. There are no guarantees that the first solution found is the optimal one, even if the costs

are uniform.

Algorithm 1 Tree Search Algorithm

```
1:  function treeSearch(s_0, A, G)
2:      frontier                                  ▷ Initially empty queue
3:      node                                      ▷ Node representing the current state
4:      solution                                  ▷ Set of Actions representing the solution
5:      frontier.push(s_0)
6:      loop
7:          if isEmpty(frontier) then return Failure
8:          end if
9:          node ← frontier.pop()
10:         if node ∈ G then return solution
11:         end if
12:         frontier.push(expandNode(node, A))
13:     end loop
14: end function
15: function ExpandNode(node, A)
16:     for all a ∈ A do
17:         if a ⇒ isApplicable(node) then
18:             newNode ← newnode()
19:             succesorList.push(newNode)
20:         end if
21:     end for
22:     return successorList
23: end function
```

Algorithm 2 Graph Search Algorithm

```
1: function graphSearch(s_0, A, G)
2:     frontier                                    ▷ Initially empty queue
3:     node                                        ▷ Node representing the current state
4:     solution                                    ▷ Set of Actions representing the solution
5:     explored                                    ▷ Set of states alredy visited
6:     frontier.push(s_0)
7:     loop
8:         if isEmpty(frontier) then return Failure
9:         end if
10:        node ← frontier.pop()
11:        explored.add(node)
12:        if node ∈ G then return solution
13:        end if
14:        frontier.push(expandNode(node, A, explored))
15:    end loop
16: end function
17: function ExpandNode(node, A, explored)
18:    for all a ∈ A do
19:        if a ⇒ isApplicable(node) then
20:            newNode ← newnode()
21:            if newNode ∉ explored then
22:                succesorList.push(newNode)
23:            end if
24:        end if
25:    end for
26:    return successorList
27: end function
```

Algorithm 1 and algorithm 2 show a possible implementation of a tree search and a graph search strategy. They can be used for both BF and DF algorithm, changing how the *frontier* queue works: `FIFO` for BF and `LIFO` for DF.

3.2.4 A* Algorithm

A* is a variant of UC where the total cost is not only the sum from the root to the current node, but also the estimation of the cost from the current node to the goal. Defining the function $g(n)$ as the total cost from the root node to the current node, we can express the UC total cost of current node as:
UC Total Cost $= g(n)$
For A* instead, we consider a bit different cost function:
A* Total cost $= g(n) + h(n)$
where $h(n)$ is an heuristic function, representing the total cost from current node to goal node and must be defined based on the current problem. Because is practically impossible to obtain such function $h(n)$, an estimation $h(n)'$ of it is usually used. The new total cost is:
A* Total cost $= g(n) + h(n)'$
The algorithm execution is identical to the UC (if $h(n)'$ is equal to 0 we obtain the UC algorithm) excepts the values are ordinate for non-decreasing costs.

A* Optimality

For A* to be optimal we have to consider an admissible heuristic, namely that never overestimate the cost to reach the goal. So, the following condition must be satisfied: $h(n)' \leq h(n)$. Another condition needed for graph search is monotonicity. The cost $h(n)'$ must not be greater than the cost from n to n' (where n' is a successor of n) plus $h(n')'$:

$h(n)' \leq cost(n, n') + h(n')'$
This two conditions guarantee the optimality of A*.

A* in Commercial Application

A* is first described in 1968[10] when N. Nilsson suggest an heuristic approach for the path-finding algorithm for Shakey the Robot. Successive improvement made by B. Raphael and P.E. Hart lead to the current version of A*. Today A* and its dynamic version D*[15](which behaves like A*, but the cost on the arc can change during the executions of the algorithm) are still widely used for path-finding, especially with grid-based maps. Some relevant example of A* applications came from video games industry, especially from strategy games, where it is used to calculate the path of units around the battlefield. *Age of Empires* [15] uses a square grid to represent the map currently played and an implementation of A* for units movement. Despite theoretically perfect the path generated was not always the best one and sometimes units get stuck in obstacles, like forest, during their journey. Even more recent implementation suffers the same problems like in *Civilization V* [16] (which uses a Hexagonal grid instead of a square one). Another relevant application is path-finding for mobile robotic unit which can uses A* or its modification for path-finding in discovered maps or partially known environment[8].

[15] www.ageofempires.com
[16] www.civilization5.com/

Chapter 4

Implementation

In this chapter, we describe the detail of the implementation of A* and Procedural generation of contents algorithms.

4.1 The Cocos2d-x Framework

Figure 4.1: *Cocos2d-x Logo*

Cocos2d-x is a cross-platform open source 2D game engine. It is written in C++ and based on the Cocos2d framework for iPhone. Cocos family includes a lot of different branches, supporting a large number of platforms.

4.1.1 Framework Description and Features

We decided to develop our game using Cocos2d-x mainly for the great number of platform which supports, from Windows to iOS, allowing us to develop the game on a platform (e.g. Windows with Visual Studio) and with little or no modification have it running on a portable device (e.g. an iPad passing through MacOSX and Xcode) second, Cocos2d-x is specifically designed for creating 2D video games with specific and useful features for this job. Figure 4.2 shows the Cocos2d-x architecture. Cocos2d-x framework is a middle-ware which uses a set of module, each one with a specific function, to provide the developer with a single API on different operating systems.

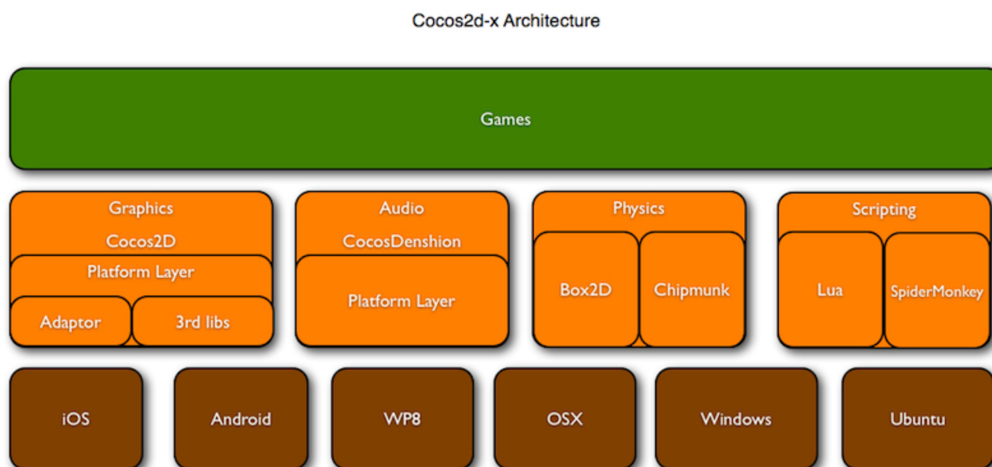


Figure 4.2: *Cocos2d-x Architecture*

Two-Phases Constructor and Memory management

Cocos2d-x uses two phases constructor in a way similar to Objective-C, Symbian and Bada SDK. The process consist in two phases:

in the first phase the standard C++ constructor is called: in it there are no programming logic but only variable initialization. In the second phase, a function containing the programming logic is called returning a Boolean value stating the outcome of the operation. In Cocos2d-x this two function are wrapped in the static `::create()` method which is used for every non singleton class. For memory management Cocos2d-x uses a reference count method. Every object contains a counter indicates the number of references of other objects to it, when this counter reach 0 the object is destroyed. Cocos2d-x wrap this method in an auto-release pool object similar to iOS `NSAutorelesePool`, so all objects instantiated by Cocos2d-x are auto-released in an automatically way unless using the `retain()` function, in this case the object is not released till a `release()`function is called.

Scenes and Layers

Cocos2d-x uses a scene approach for managing the program flow. A Scene is an independent part of the program work-flow, even if usually a program is composed by different scenes, at a given time only one can be active as shown in Figure 4.3. A scene is composed by different layers of the size of the drawable area one on top of each other; they can be semitransparent allow seeing the layers behind it. A layer contains sprites and other drawable objects, furthermore it handles all input requests from the user and know how to draw itself, so there is no need to call a `draw()` function for each layer. Transitions between scenes are managed

by a `CCDirector` a singleton object which knows the current scene and manage the scene call stack.

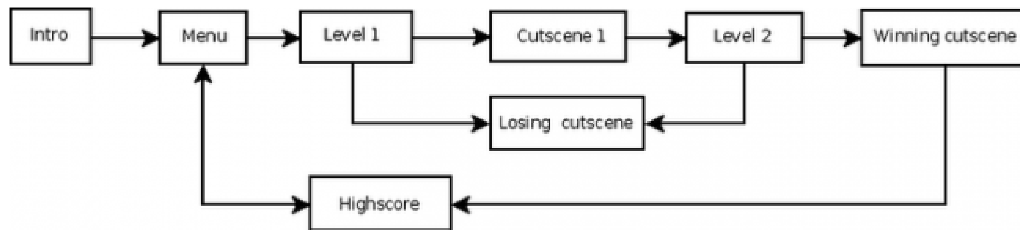


Figure 4.3: *An example of scene work-flow*

Actions

Actions are transformation applied to an Object derived from the class `CCNode`. Actions can change scale, rotation, opacity, colour, etc. of the objects and can be divided in two main groups: *IntervalAction* if the attribute is modified over a time interval or *InstantAction* if the object attributes are modified in a single action. In Cocos even scenes are subclasses of `CCNode` so they can be transformed with actions.

Title Maps

In certain type of games, like the one we have developed, the playing area is composed by small rectangular image called Tile. This procedure is useful to generate large maps with little art resources. Cocos2d-x support the use of tile for creating Orthogonal/Isometric/Hexagonal maps; the tilesets (the sets of all image needed to draw a map) are created with an external program (Tiled) and saved in `.tmx` format. Then the framework provides all the function needed for an easy use of tiled maps inside our application.

4.2 CASTLE Implementation

4.2.1 Map Generation

All maps in our application are procedurally generated. Figure 4.4 shows two different maps from the same castle. Each level of our game is created by the software in an automatic way, except for some fixed object needed for the plot's development. In our game castles can be generated by one of four pre-generated seed or by a random one. A seed is an alphanumeric string which is used to initialize a random function. If a random function is initialized with a seed, then every time we use that function with that seed the output result is the same. This is particular useful if we want to reproduce the same map, quiz disposition and furniture across different game sessions. From a logical point of view the map is an acyclic graph, limited by a maximum number of rooms. Each node of the graph represent a room which can have a maximum of 4 connections with other nodes, except for the root node which can have a maximum of 3 connections. One side of the root node is always used for the entrance door which brings the player to the world map. The map is enclosed in a boolean matrix structure which limits his expansion in the vertical and horizontal directions as well as helps us to identify the room position on the screen when drawing the map. Starting from a root node with a pseudo-random choice we create a new room in one of the four possible cardinals directions (north, south, east, west), next we move in the newly created room and repeat the room creation step. When

a room has no expansion slots we move back to the previous room and call the room creation function, if this has no expansion slots we move to the previous one and so on. A room has no possible expansion slots if each new possible rooms falls under one of these conditions:

(i) Exceed the boundaries of the matrix

(ii) It is adjacent to a room which is not connected to

The algorithm terminates when the maximum number of rooms has been reached.

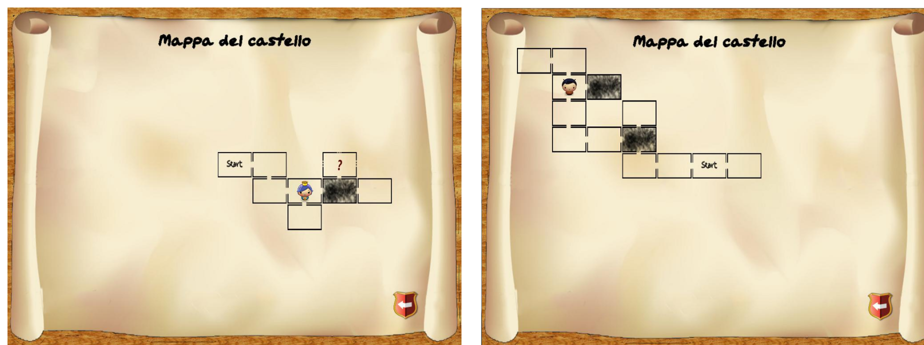


Figure 4.4: *Two Different Maps generated by the Algorithm*

Graph Search

Since the map is represented by a graph we have implemented a depth-first tree search algorithm to scan it (see Chapter 3.2.3). It is a recursive implementation, starting from the root node the algorithm scans each room expanding first the leftmost node if it is possible. If the room is a leaf then it came back to the previous room and expands the second node in the list.

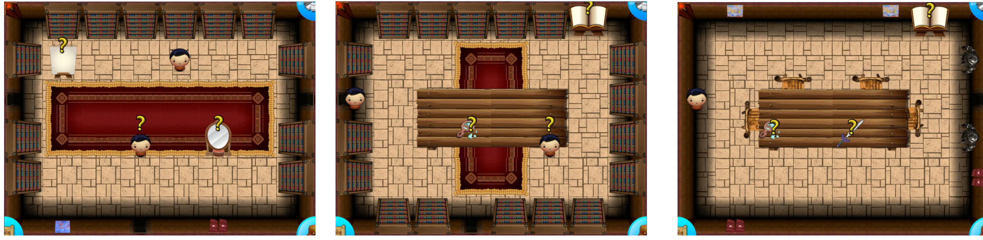


Figure 4.5: Different Room Furniture

4.2.2 Furniture and Quiz

Furniture and quiz disposition in CASTLE are procedurally generated too. Once the map is created, we scan it and add furniture to each room. There is a finite number of furniture divided in different categories (floor furniture, wall furniture, decorations) for each one, we choose one or more objects and add it to the room. In Figure 4.5 the different types of furniture can be seen in three different rooms. The choice is based on the random function, initialized with a seed. Between different categories some constrains may applied, e.g. we cannot add a library in the middle of the room if a table was already added. Once the room is furnished the following step is quiz placement. Based on the furniture disposition the quizzes are placed in such a way that they can be always reachable by the player and they do not block the access to the other rooms. Then, quizzes are chosen randomly from a pool and loaded in the game through an `xml` file containing all possible quizzes.

4.2.3 Player Movements with A*

When the player click on a point on the map, the character should move from its position to the new one following the shortest path on the map, considering obstacles. In our application, we use A* to do this. We implemented it following the state of the art algorithm explained in Chapter 3.2. In our case the input parameters are: the starting position (e.g. the current position of the player on the map), the goal position (e.g. where we want to send him) and the matrix representation of the current room.

Grid Representation

Each room of CASTLE is represented by a 8x6 binary matrix data structure where the blocking object are represented by ones. On the screen it is composed by 8x6 tile each one of 128x128 pixel summing up in 1024x768 total resolution. It has been chosen because it is the native resolution on iPad. Each object rendered on the screen occupies one or more tiles in the room and can be blocking or non-blocking for the player. With this type of representation the player can only move in a discrete way from square to square (the player can not move itself from the centre to the edge of a square, but it can only move from the centre to the centre of another square)

Heuristic Function

Considering our problem and his matrix data structure representation, we have chosen the Manhattan distance[1] (or taxicab

metric) as heuristic function. The Manhattan distance in an Euclidean Plane is defined as:

$$f((x_1, y_1), (x_2, y_2)) = \mid x_1 - x_2 \mid + \mid y_1 - y_2 \mid$$

were $x1$ and $y1$ are the starting coordinates and $x2$ and $y2$ the final coordinates of the player. As previously described in Chapter 3.2.4, heuristic used by A* must be admissible and monotone to satisfy optimality. Manhattan distance is admissible since it calculates the path assuming that the player can go straight to its goal position ignoring obstacles. Accordingly, the estimated path can never be greater than the actual path. For demonstration of consistency we use the "triangle inequality" theorem [18]."triangle inequality" state that an edge of a triangle can not be greater than the sum of the remaining edges. We can consider consistency a form of "triangle inequality". Consider the vertices of a triangle as node in our map and edges the cost function between the nodes, as shown in figure 4.6. "triangle inequality" holds in taxicab geometry so, we can applied it to our triangle obtaining the equation: $h(start) \leq cost(start, n) + h(n)$. This equation is the same of 3.2.4 so, the consistency condition is satisfied.

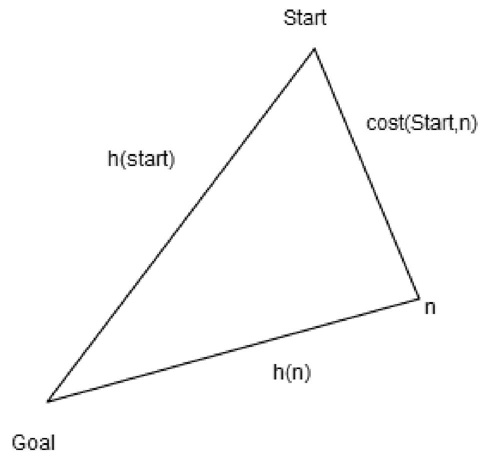


Figure 4.6: *Triangle inequality*

Implementation Description

Algorithm 4.1 shows our implementation of A*. Variable `aNode` represent the current node which we are considering. List `frontier` is an ordered list containing nodes waiting to be expanded. In the beginning it will only contains the player position (`CCPoint start`). List `explored` contains the states already processed. this is used to implement the algorithm in his graph search version. The first step is the expansion of the player position node, through the `graphExpandNode(...)` function. When a node is expanded the algorithm considers all squares near him (only the ones that can be reached by a single movement, not diagonal.) then, the available ones, the ones that are not occupied by any object, are put into the frontier which is sort based on the node total cost from the lowest to the highest. The sorting is done in the `insertList(...)` function. Starting from a node n, the generic expanded node n' total cost is:

$cost(n') = cost(n) + 1 + f(n', goal)$
where $f(n', goal)$ is the Manhattan distance from n' to the $goal$. The state of the node is also inserted in the `explored` list, so the same state is not processed twice. Since the algorithm expands always the first node in the frontier it is always the one with the lowest total cost. The algorithm is repeated until the goal node is reached or the frontier is empty. When the goal node is reached the function `buildPath(...)` computes a list of actions to actually move the player.

Listing 4.1: A* Implementation

```
1 CCArray *MovementUtilities::graphSearch(CCPoint goal, int
     wallMap[WIDTH][HEIGHT], CCSprite *sprite, CCPoint start)
2 {
3         //Declarations
4         while(!frontier.empty())
5         {
6                 iterations++;
7 
8                 aNode = frontier.front();
9                 frontier.pop_front();
10                explored.push_front(aNode->currentState);
11
12                if(aNode->currentState.x == goal.x && aNode->
                     currentState.y == goal.y)
13                {
14                        CCArray *solution = buildPath(aNode,
                             iterations, sprite);
15                        lastPoint = aNode->parentNode->
                             currentState;
16                        return solution;
17                }
18                //Expand current node and sort the frontier
                     based on the total cost
19                frontier = insertList(graphExpandNode(aNode,
                     wallMap, explored), frontier);
20        }
21        return NULL;
22 }
```

If the path is too long, for performance reason, it is split in two different sub-path. First the path between the start and a mid-point is calculated, then from the mid-point to the goal point. The mid-point is calculated as the midpoint of a line segment starting in $(0, 0)$ and ending in the goal node.

Chapter 5

Conclusion

In this work, we illustrated our implementation of map generation and player navigation, in video game CASTLE.

5.1 PCG: Evaluation

The two algorithms for content generation performed well together, overall maps geometry result appealing to the player while the room are furnished in a pleasant way, and the quizzes are always placed in reachable spots. The introduction of a new constrain during map generation (a new room cannot be near a room which is not connected to) helped us to have more branches resulting in more fun to explore maps. Furthermore we can generate castles with an arbitrary number of rooms or add new furniture to it simply tuning certain parameters on the algorithms. PCG has saved us time, speeding up the process of level creation. From a performance point of view the algorithms are fast enough to generate 4 castles, for a total of 80 furnished rooms, at the startup

with acceptable loading times.

5.2 A* Search: Evaluation

We have found some minor problems during the implementation of A* especially regarding performance. While normal path were calculate in an acceptable time, certain pair of start-goal position yielded to execution times too long to be acceptable. This problem has been fixed calculating the total length of the path and splitting it if it was too long. Other problems were found were too many input were dispatched or permitted goal position was unreachable. All this problems have been fixed and now the algorithm works as intended.

5.3 Further Works

Since the game is completed our top priority is to release it to iTunes marketplace. Our current release works on iPad, so, all test on this platform have already be made. Subsequently a porting for other platform like Android or Windows phone 8 is desirable. Thanks to cocos2d-x the code can be ported easily on different platform but a new testing phase on different devices must be done.

Bibliography

[1] M. Barile. Taxicab metric, 2010.

[2] T. Cross. All the world's a game. *The Economist*, 2011.

[3] D. Peachey K. Perlin S. Worley D.S. Ebert, F.K. Musgrave. *Texturing and Modeling. A Procedural Approach.* Morgan Kaufmann, third edition, 2002.

[4] A. Farnell. An introduction to procedural audio and its application in computer games., 2007.

[5] A. Ilachinski. *Cellular Automata: A Discrete Universe.* World Scientific, 2001.

[6] P. L. Lanzi L. Cardamone, D. Loiacono. Interactive evolution for the procedural generation of tracks in a high-end racing game, 2011.

[7] J. Togelius L. Johnson, G. N. Yannakakis. Cellular automata for real-time generation of infinite cave levels, 2010.

[8] P. Muntean. Mobile robot navigation on partially known maps using the a* algorithm, 2012.

[9] P. Pasquier N. Sorenson. Towards a generic framework for automated video game level creation, 2010.

[10] B. Raphael P. E. Hart, N. J. Nilsson. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transaction of Systems Science and Cybernetics*, SSC-4(2), 1968.

[11] A. Lindenmayer P. Prusinkiewicz. *The Algorithmic Beauty of Plants.* Springer-Verlag, second edition, 1996.

[12] K. Perlin. Improving noise, 2002.

[13] F. Phillips F. Scheepers S. F. May, W. E. Carlson. Al: A language for procedural modeling and animation, 2007.

[14] P. Norvig S. Russell. *Artificial Intelligence - A Modern Approach.* Prentice-Hall, third edition, 2010.

[15] Anthony Stentz. The focussed d* algorithm for real-time replanning. In *In Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

[16] Bryan Stout. Smart move: Intelligent path-finding. *Game Developer Magazine*, 1996.

[17] C. D. Toth. Binary space partitions: Recent developments, 2005.

[18] E. W. Weisstein. Triangle inequality, 2010.