Executive Summary

Blank at the moment

Acknowledgements

While this thesis was the culmination and result of many long solitary hours of research and development no effort is truly without support. I would like to extend immense thanks and gratitude to Dr Benjamin Sach for his guidance and support throughout the project. He has been an unrivalled source of information and helping me find a deeper appreciation of computer science and mathematics.

I would also like especially thank my dear Tingting for being willing to subjected to innumerable proof reads and generally being incredibly patient and understanding throughout the project's development process. Without her feedback, support, and encouragement, this project would not have been anywhere near as successful as it has been.

Contents

Chapter 1

Aims and Objectives

"The obvious objective of video games is to entertain people by surprising them with new experiences."

Shigeru Miyamoto

1.1 Background, context, and motivation

An NP-complete problem is one with a solution that can be *verified* quickly in polynomial time, though an existing solution is not readily known or *identifiable* [8]. It has been proven that certain classic video games, such as *Super Mario Bros* [1], are computationally hard in that the character could be made to solve an arbitrary instance of a Boolean satisfiability problem (SAT). SAT is a type of NP-complete problem in which the variables of the formula may be consistently replaced by the values TRUE or FALSE in a way that the formula evaluates to TRUE, or 'satisfiable'.

The primary motivation of this research is to investigate the feasibility of utilising the video game medium as an effective visualisation of computational complexity. If successful, this type of visualisation could have a wide range of effects, including use as teaching tool to new learners approaching the subject.

There are several challanges that need to be addressed in order for this method to be feasible.

1.2 Research objectives

The aim of this project is to produce a game engine that is capable of visualising and demonstrating concepts in computational complexity through the evaluation of different instances of SAT. The game engine would be able to take an instance of SAT and generate a human-playable level that is computationally hard. This level would then be able to be solved by a SAT solver. The solution to the level would then be visualised on screen. In order to fulfil this aim, the research project will implement a game engine through achieving the following objectives:

1. Develop a platform game in which the mechanics are NP-hard;
2. Develop a level generator which converts an instance of SAT into a playable level;
3. Develop a visualisation of the SAT solving these levels by choosing a suitable path;
4. Evaluate level-generation and display of the SAT solver's decisions for efficiency and success against an established criteria or one developed during the project.

As shown in the preceding section, the research being undertaken brings together a diversity of broad topic areas, which is challenging. Given the research aims and objectives described above, the scope of the research review is to understand the current state of research and development in the following areas:

1. The principles and concepts of computational complexity which class a task or activity as 'difficult' or 'hard'. This includes an assessment of the characteristics which define the differing classes of complexity;
2. The tests of computational complexity that prescribe the different classifications through Boolean satisfiability. This includes how these tests may be applied to video games, with particular focus on 2D platform games;
3. The generation of video game content in order to produce the problem instances as levels in an accurate and efficient manner.
4. Assessing the means for implementing these divergent areas of research in order to produce a bespoke game engine that realises the research aim.

As shown from the range of topics above, the areas being investigated are expansive and multi-faceted. Therefore, this research review surveys those elements most relevant for the implementation and development of the game engine software.

1.3 Thesis Structure

This research project is divided between 66% Type I (software development) and 34% Type II (investigatory, research). The structure of this document reflects this division in that the theoretical concepts are broadly introduced and discussed insofar as relevant to the research aim. The discussion strongly focuses on the implementaiton and development challenges in realising these concepts in the project.

Chapter 2 covers the fundamental concepts in computational complexity, identifying which characteristics render a task difficult. The chapter discusses the well-known P vs NP question and explains the NP-hardness complexity class. Other classes which are later explored within the project are also demonstrated and analysed.

The overview of computational complexity provides the basis for the discussion in Chapter 3 on Boolean Satisfiability problems and SAT solving software. In addition to explaining the major features of SAT solvers, the chapter also identifies SAT solvers which had been considered during the project's development.

Chapter 2

Computational complexity theory

"In general, it is much harder to *find* a solution to a problem than to *recognize* one when it is presented"

Stephen A Cook [8]

2.1 Fundamental concepts

2.1.1 Definitions

The goals of the research project are to visualise computational complexity through video games. Therefore, the literature review begins with identifying the relevant key concepts in computational complexity theory in order to understand what characteristics makes a problem computationally hard.

Computational complexity theory is concerned with assessing and classifying the difficulty of defined tasks as well as the relationship between such tasks. Within computational complexity, there exist different types of problems, termed *complexity classes*. Complexity classes rank the difficulty of differing types of problems. Of the many types of problems which can exist in computational complexity theory, the two fundamental types this research project is interested in are (i) decision problems and (ii) search problems. The importance of the distinction between these two types will be made clearer when discussing Boolean satisfiability (Chapter 2) and procedural content generation (Chapter 5).

Decision problems and search problems

A decision problem is a binary choice one wherein, on an infinite set of inputs, the question may be answered '*yes*' or '*no*' It is common to define the decision problem equivalency as: "the set of inputs for which the problem returns *yes*". Most problems may be reduced to a decision problem [35], which is important in that the determination of whether a solution exists or not is required in order to solve the complementary search problem [23].

Search problems represent a significant area of research in computer science (citation needed), covering topics such as sorting and identifying the shortest path [23]. A search problem consists of the identification of a solution from a set of solutions (possibly infinite or empty). Therefore, given an instance of a search problem, a solution must be found or return a determination that no solution exists in such instance.

The concept of reductions

Different complexity classes may be defined through the ability to transform an instance of one type of problem into another. This ic alled a reduction and is the basis for classifying the

The most common reduction is a polynomial time reduction which means - ////-something ////

///// COPY PASTE FROM WIKI //
Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem X can be solved using an algorithm for Y, X is no more difficult than Y, and we say that X reduces to Y. There are many different types of reductions.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

//

2.1.2 Survey of complexity classes

P and NP

The two fundamental complexity classes relevant for this research project are **P** and **NP**. The P (polynomial) class of problems contain those decision problems that are often described as ‘easy’ problems [35] as the time required to solve such problem has an upper- bound that scales by a polynomial function of the input [50]. The other complexity class, NP (non-deterministic polynomial), relates to those problems that do not have an efficient way of finding a solution but can have a solution verified in polynomial time [23, 47].

P vs NP Problem

The P vs NP problem refers to search problems to which there exists an efficient algorithm that given a solution to a given instance determines whether or not the solution is correct[50, 23, 35, 18].[1] Although any given solution to an NP- complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place. A distinguishing characteristic of of NP- complete problems is that there is no known efficient solution to them [23]. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.

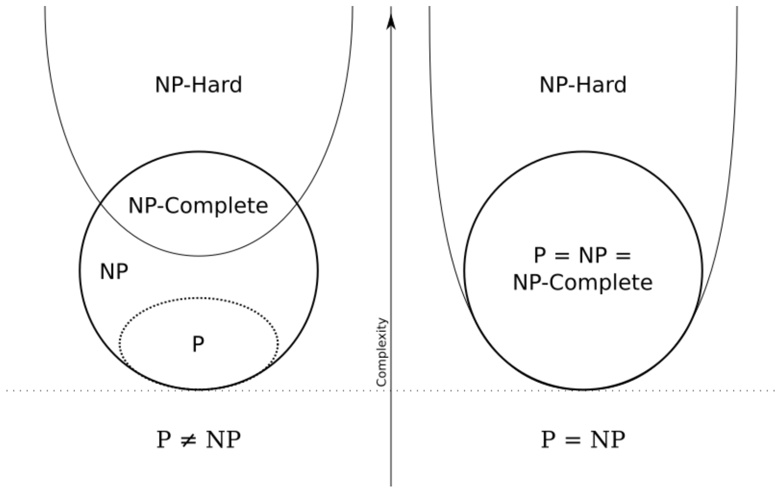


Figure 2.1: Euler diagram illustrating the relationship between P and NP [19].

Proving that P = NP or P $\neq$ NP is one of the most important open questions in theoretical computer science and has been described as one of seven principal unsolved problems in the field of mathematics [15]. The implications of P = NP are far-reaching. If it were possible to demonstrate that P = NP (i.e. all NP problems can be solved in polynomial time), then it would then be possible to solve difficult problems in polynomial time (see Figure 2.1). For every problem that has an efficiently verifiable solution, that same solution could also be efficiently identified. While the problem is still being researched and open to debate, there is wide belief that P $\neq$ NP is more likely than P = NP [22].

NP-hard

A problem is considered to be classed as **NP-hard** if the solution's algorithm may be adapted to solve any problem in NP. By extension, this will then include all problems in P, which are contained within NP. It should be noted that not all NP-hard problems fall within NP nor are NP-hard problems necessarily decidable problems [50, 23, 35, 18].

NP-complete

When a problem is classed as both NP and NP-hard is said to be **NP- complete** and are considered to be the hardest problems in NP [50, 23, 35, 18]. A problem that is NP- complete can have every other problem in NP reducible to it [48]. 'Reduction'

[1]This is also sometimes described as the P vs NP Question in literature

in this sense that there exists an algorithm in P that change a problem into another instance [18].

PSPACE

Another relevant complexity class to this research review is **PSPACE**, as some games have been proven to be in this class [20, 13].[2] PSPACE refers to decision problem which may be solved with an amount of space polynomial in the size to its input [50].

EXSPACE

2.2 Chapter summary

This chapter has identified the relevant complexity classes and problems in order to establish that video games are computationally complex in Chapter 4. In the next chapter, these concepts will be used to show how problem instances may be proven to be NP-complete through the use of Boolean expressions. This will provide the foundation for being able to develop a game engine which produces computational hard levels in a video game.

[2]Please refer to Chapter 3 of this research review for further information.

Chapter 3

Boolean satisfiability problems

"It is not of the essence of
mathematics to be conversant with
the ideas of number and quantity."

George Boole

Boolean satisfiability problesm (or SATs) serve as the basis for proving the hardness of video games and have implications for the game engine design as well as the design of the game level (see chapters X and Y).

3.1 Principle concepts and definitions

A Boolean formula is a logic expression with variables that have a value of either TRUE or FALSE. These formulae also contain logical connectives, which subscribe to an order of precedence: negation ($\neg$), conjunction($\wedge$), disjunction ($\vee$), implication ($\leftarrow$), and equivalence ($\leftrightarrow$) [2, 24].

Formulae are constructed through literals, which is a variable (e.g. x or y) or its complement (e.g. x or $\neg x$). Literals are joined together in groups called clauses. The literals and clauses may be joined either disjunctively (i.e. or conjunctively (i.e. the expressions must collectively evaluate to true)

Of these constructions, the conjunctive normal form (CNF) is of particular importance, especially in relation to finding solutions to satisfiability problems, particularly those solving methods based on DPLL algorithm [24] (See Section X below for further discussion). To be in CNF, a formula must a conjunction of clauses, wherein those clauses are a disjunctive of literals. The example below illustrates conjunctive normal form with three clauses (joined together by $\wedge$ (AND)) containing three literals (x, y, z).

$$(\mathrm{x} \vee \neg \mathrm{y}) \wedge (\neg \mathrm{x} \vee \mathrm{z} \vee \mathrm{y}) \wedge (\neg \mathrm{z} \vee \mathrm{x})$$

To satisfy this expression, each clause must evaluate to TRUE as they are conjunctively joined. If any clause were to be FALSE, the formula twould be unsatisfiable.

3.2 Boolean Satisfiability problem and NP-completeness

The Boolean satisfiability problem (SAT) is the problem of determining if there exists an assignment that satisfies a given Boolean formula. In order for such assignment to be *satisfiable*, the variables' values are replaced constantly with TRUE or FALSE until the formula evaluates to TRUE. Where no such assignment is possible (i.e. the formula evaluates to FALSE in all combinations), the formula is determined to be *unsatisfiable* [2, 24]. Therefore, given a CNF formula F, SAT may be posed as: does F have a satisfying assignment?

This question has an important place in computer science as it was the first problem proven to be NP-complete [7].[1] In solving a SAT problem, the question involves not only providing a yes or no, but also in finding an actual solution to the assignment [66]. Specialised software, SAT solvers (discussed in Section 3.2.2 below), provide these solutions, if one exists.

3.2.1 Examples of SAT reductions

Three-satisfiability or 3SAT is the SAT reduction is most relevant for the research project as it has been used to demonstrate the NP- completeness of 2D platform games [1]. In 3SAT, there may only ever be three literals in each clause and at least of of the literals contain a value of TRUE in order to achieve satisfiability [2].There is a variant of the 3SAT, exactly 1-3-satisfiability or 1-in-3SAT above reduction wherein the each clause must contain *exactly* one literal [2, 18].

3.3 Solving SAT instances

SAT instances can be solved through a specialised piece of software called a SAT solver. Given a SAT instance, a SAT solver can either find a solution, which would be a satisfying variable assignment or prove that no solution exists [66]. There are also stochastic methods based on local search may be able to find satisfiable instances in a fast manner but are unable to prove that an instance is in fact unsatisfiable [24]. Understanding SAT solvers is fundamental to the development of the project's game engine.

3.3.1 Overview of the mechanics of SAT Solvers

SAT solvers work recursively. SAT solvers features and functions.

The DPLL algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [10] is a complete, systematic search algorithm which is based on backtracking.

////BACKTRACKING IS WHAT???

[1]While this is often described as *Cook's Theorem* it had been independently verified in [37] and is more accurately termed the *Cook- Levin Theorem*

The DPLL algorithm (see Figure 3.1) is able to decide the satisfiability of a CNF SAT reduction as well as identify the satisfying assignment. It can also show that a given Boolean formula is unsatisfiable. The algorithm achieves these results through the use of branching and where clauses are found to be FALSE, removes these from its search space.

```
Input  : A CNF formula $F$ and an initially empty partial assignment $\rho$
Output : UNSAT, or an assignment satisfying $F$
begin
    $(F, \rho) \leftarrow$ UnitPropagate($F, \rho$)
    if $F$ contains the empty clause then return UNSAT
    if $F$ has no clauses left then
        Output $\rho$
        return SAT
    $\ell \leftarrow$ a literal not assigned by $\rho$          // the branching step
    if DPLL-recursive($F|_{\ell}, \rho \cup \{\ell\}$) = SAT then return SAT
    return DPLL-recursive($F|_{\neg\ell}, \rho \cup \{\neg\ell\}$)
end

sub UnitPropagate($F, \rho$)
begin
    while $F$ contains no empty clause but has a unit clause $x$ do
        $F \leftarrow F|_x$
        $\rho \leftarrow \rho \cup \{x\}$
    return $(F, \rho)$
end
```

Figure 3.1: DPLL in pseudo-code. Adapted from [2, 24].

The DPLL algorithm works with the CNF of a propositional logic expression (as described in Section 3.1 above). Given that any formula may be converted into CNF through the addition of new variables corresponding to the sub-formulae [61, 23], DPLL may used to in all cases.

Features of modern DPLL-based SAT solvers

Modern DPPL-based SAT solvers are readily available and useful for this project. There are currently a number of highly scalable SAT solvers, all based on the classic DPLL search framework. These solvers, now also known as conflict-driven clause learning (CDCL) solvers, can generally handle problem instances with several million variables and clauses [33]. In the section below, a collection of important features have been identified and have relevance for the research project.

1. **Selection heuristic** : Also termed the 'decision strategy', this is the procedure of *how* variables are selected and assigned values. This aspect varies the most between different solvers can significantly impact the efficiency of a solver [38, 24, 66]. There are a range of strategies that can be employed including: Maximum occurrence in clauses of minimum size heuristic [32], Bohm's heuristic [38], dynamic largest individual sum heuristic [39], variable state independent decaying sum [42].

2. **Clause learning** : This method allows the solver to learn the causes of unsatisfiability (sometimes termed 'conflict') in clauses, using this information to

minimise the search space [3]. When a conflicting clause is encountered, the solver will needs try to identify the cause of a conflict and attempt to resolve it. This is achieved by asserting that a solution does not exist in the particular search space, backtrack (undoing the previous decisions) and continuing the search in a new one. [66]. This feature has been identified as one that improves the underlying DPLL algorithm [66, 24].

3. **Conflict clause minimization** Een and Sorensson [55] had first explored the use of this method in their `MiniSat` solver. By utilising subsumption resolution, the size of the learned conflict is minimised through the removal of literals that are implied to be FALSE when the rest of the literals in the clause are FALSE [66]. This increased efficiency come with an increased computational cost [24].

4. **Conflict-directed backjumping** : Stallman and Sussman [57] proposed a method to allow a solver to backtrack directly to a decision-point p if the variables at level p or lower are causing conflict. The assumption is therefore that there is no solution to be found in this search space. It is agreed that this provides greater efficiency and completeness to the procedure [24].

5. **Fast backjumping** : Gomes *et al.* [24] describe this feature as allowing for a solver to directly go to a lower decision level if even one branch in the search space is in conflict. The authors note that this may not always increase efficiency however. The branch at depth level d is not marked as unsatisfiable but rather a new variable and value is selected for that level, continuing with a new clause. There has been some experimental work that suggests an increase in solving efficiency [24, 36].

6. **Watched literals scheme**: This feature monitors two classes of literals in an unsatisfied clause: TRUE or unassigned [42]. As an empty clause will cause the DPLL to stop, This feature had been introduced in the the `zChaff` solver and is utilised by many other solvers due to the efficiency of the constraint propagation [24]. This somewhat of an advancement of the 'lazy data structures' which had been introduced by the `Sato` solver [65]. This feature has furthered development of clause learning [66, 24].

7. **Assignment stack shrinking** : In the `Jerusat` SAT solver [44], Nadel had introduced this feature which is based on conflict clauses. Assignment stack shrinking servers to make the search area more local by 'shrinking' the conflict clause once it reaches a set threshold [45]. The 'shrinking' is achieved by finding the lowest decision level that is less than the immediate higher level by at 2. The solver will then backtrack to this level and where possible, sets unassigned literals of the clause to FALSE [45].

8. **Randomised restarts** : This provides that the clause learning algorithm can restart the branching process from decision level 0 while retaining all the learned clauses *et al.* [25]. Pioneered `zChaff` [42] many modern SAT solvers utilise highly

aggressive restarts strategies, even below 20 backtracks, as it demonstrably lowers the solution time [24].

3.3.2 Converting a formula for SAT Solver Input

The SAT instance must be read by the SAT solver. The standard input into a SAT solver is a DIMACS CNF file[?]. The file has a specified format which allows the a formula to be input into the solver. Consider the following example:

```

c example.cnf
c
p cnf 3 3
 1  2 −3 0
−2  3  1 0
−1 −2  3 0
```

The `c` character is a comment indicator and will not be read by the SAT solver. The SAT solver begins reading from the `p` character, which indicates the problem line. On this line the SAT solver is given information about the instance such as the format, in this case CNF, and the variables. and the clauses (). th. The reader ends a the '0' and begins reading the next line until the end of file.

There are other formats available in a DIMACS satisfiability file, which include the `sat` indicator instead of CNF[?]. These other formats were not explored during this project as the CNF format fulfilled the project specification and was able to be rapidly deployed.

The DIMACS file requires that the clause only contain one literal of a variable per clause [?]. This means that x and not x are unable to appear in the same clause. Attempting to create and run a file with this structure will generate an error or an incorrect assessment of the SAT instance.

3.4 Chapter summary

Boolean satisfiability is the foundation for proving NP-hardness and NP- completeness for the project, as will be demonstrated in the next chapter. The availability of SAT solvers is also useful in order to try and test different ones with the research project's game engine. Currently in the available literature, there is very little research in this regard to the subject of linking SAT solvers and video game engines, therefore representing a fertile ground for development with this research project. The issue may be that these solvers will not scale well with the game engine. In that respect, Brummayer et al. [5] developed 'fuzz testing' techniques which allow for debugging errors in larger SAT instances. These techniques are useful in ensuring the scalability of the SAT solver when attempting to solve large levels.

Chapter 4

Computational complexity and the hardness of (video) games

"I strongly approve the study of games of reason, not for their own sake, but because they help to perfect the art of thinking."

Gottfried Wilhelm Leibniz

4.1 Overview

Having established the concepts and means of testing for certain classes of computational complexity in the previous chapters, this chapter will address the assessment of complexity in video games. There is a growth of interest and research in analysing 'mainstream' video games, including two- dimensional (2D) platform games [63, 20, 1, 51]. This chapter will identify the characteristics and tests required to class a game as computationally hard. This will inform the design considerations of the research project in Chapter 6.

Puzzles and games have long been a subject of complexity research with many games being found to be at least within NP-hard [35]. This is often based on the assumption that P $\neq$ NP [11]. The list includes a variety of grid and block placement games, wherein a player manoeuvres a block around a map to a specified location, such as *Minesweeper*[34] and *Tetris*[14]. Titles such as *Sokoban*, *Push* and *PushPush*, which have been shown to be not only NP- hard [12], but also PSPACE-hard [9, 16]. In the case of *PushPush*, there have been proofs of NP-hardness not only in 2D but in 3D as well [46]. More recently, Gualà *et al.* have demonstrated that a number of 'match-three' games, grid-based games which are popular on mobile devices, such as *Candy Crush Saga* and *Bejewled* are NP- hard [28].

The current trend in this field has been to assess classic franchises such as *Prince of Persia*, and *Donkey Kong Country.* These studies are more relevant to the research project as visualisation will utilise a 2D platfomer resembling a game like *Super Mario*

Bros.

4.2 2D platform games

The research project focuses on developing a two-dimensional (2D) platform game and thus the proof for NP-completeness is of particular interest. These games are (very often) single-player games wherein the player traverses levels in order to proceed to the next level. A 'level' is a virtual spaces in which the player character can manoeuvre and interact [6]. The player's game-play is influenced by the physics of the world, which often mean a limitation of jump height and distance [6].

Characteristics of 2D games

In an examination of 2D platform games, Forišek provides an indicative list of common puzzle elements found within the genre [51, 20]. These include:

- **long fall**: the maximum height from which the player character may fall without receiving damage.
- **door opening**: the game world may include a variable number of doors and suitable mechanisms to open them
- **door closing**: the game world contains a mechanism to close doors and a way for the player to trigger the mechanisms
- **collecting items**: a set of items the player collects as part of the game. These may be necessary or optional content.
- **enemies**: Navigating the level may be made more challenging by having 'enemy' characters that must either be avoided or defeated by the player

4.2.1 Characteristics of NP-hard games

These above elements can be modeled into Boolean satisfiability problems, which are known to NP-complete [20, 7]. The proofs of video games rely on this analogy and if a proof then removes certain game elements, it is expected that the complexity of that game is reduced [63].

Viglietta [63] surveyed several classic video games games published between 1980 and 1998, identifying general common elements and schemes in video games that class them as computationally hard. These include collectable items, activation of pathways (e.g.by key or button) by, or the ability to destroy paths. Forišek [20] further identifies several features specific to 2D platform video games which provide for complexity, with particular focus on *Prince of Persia.* In the work, he posits a number of meta theorem, of which four are particularly relevant for the research project:

- **Meta-theorem 1**: A 2D platform game where the levels are constant and there is no time limit is in P, even if the collecting items feature is present.

- **Meta-theorem 2**: A 2D platform game where the collecting items feature is present and a time limit is present as part of the instance is NP-hard.
- **Meta-theorem 3**: Any 2D platform game that exhibits the features long fall and opening doors is NP-hard.
- **Meta-theorem 4**: Any 2D platform game that exhibits the features long fall, opening doors and closing doors is PSPACE-hard.

Following the criteria above, it can be demonstrated that many platform games are not only able to be classified as NP-hard but even PSPACE-hard in some cases.

Aloupis *et al* [1] surveyed the complexity of classic Nintendo franchises by considering that the games are essentially a decision problem of reachability: "given a level, is it possible to reach the goal point t from the start point s"? The authors also review subsequent *Super Mario Bros* titles as well as other platform games such as *Donkey Kong Country.*

In the work, the authors had generalised the map size and left all other elements of the game in the original settings. The proofs in [1] rely on a reduction of the game to 3SAT and the 'level' was modelled to the problem instance. This represents two potential points of difference from the current research project. First, the research goal is to develop a means to generate large levels and scale accordingly. Secondly, further reductions aside from 3SAT may be explored.

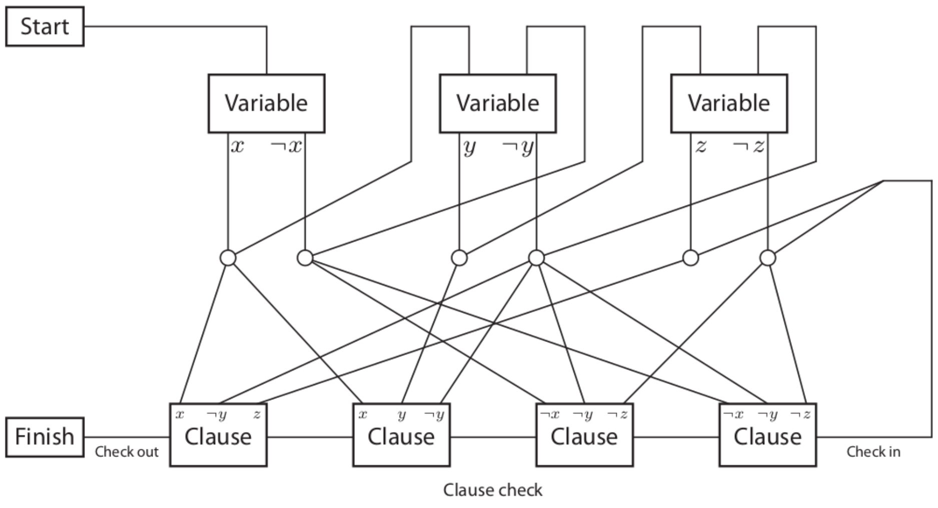


Figure 4.1: Framework for NP-hardness [1].

The framework for proving NP-hardness of games (see Figure 4.1) is reduced from 3SAT (described in Section 3.2.1 of this review). The player begins the level at the `START` gadget and then traverses through variable gadgets. A *variable gadget* requires the player to make an choice, which amounts to TRUE (x) or FALSE ($\neg x$) for the values in the formula. These decisions allows the player to follow paths leading to *clause gadgets.* A clause gadget creates a permanent state change and as a consequence, the player cannot access other paths connecting to that gadget. As can be seen in the framework above, there are several areas of potential crossover. To prevent such occurrence, *crossover gadgets* prevent a player from switching paths.

Once the player has passed through all the variable gadgets, the player can proceed to the `FINISH` through a clause check path. This can only be successfully traversed only if every clause gadget has been unlocked.

4.3 Chapter summary

The frameworks provided by [1] give a comprehensive model for the project's implementation. Modelling these gadgets to SAT clauses will form a significant area of the research project's work. By utilising the metatheorms provided by Forišek [20], it may be possible to incorporate other complexity levels including PSPACE-hard.

Having established how a 2D platformer may be classed as NP-hard (or even PSPACE- hard), the focus of the literature review shifts to the details of how these levels/problem instances will be generated, explored in the next chapter.

Chapter 5

Game Engine Design

"QUOTE."

Someone

This chapter describes the relevant game design considerations which drove the game engine's development as well as provide a general overview of the modules and functionality. The chapter serves as a reference and basis for the later chapters on level generation and player AI.

5.1 Design considerations

This section describes the design considerations which shaped the development of the game engine. The purpose is to provide the rationale and justification for design decisions.

The game engine was developed on Ubuntu Linux 15.04 on a HP laptop with an AMD A8 processor and GRAPHICS CARD. It was compiled using LLVM/Clang (clang++). There is no appreciable difference when using the GNU Compiler Collection (gcc).[1]

5.1.1 Development language

The game was developed in C++ using the C++11 standard. The C++ language was selected because the DPLL-based SAT solvers considered for the project in Section FIXTHIS are all written in C or C++ for efficiency reasons [66].It is therefore a reasonable decision to utilise the same programming language as that used in the source code for SAT solvers. This ensures that the SAT solver's code base can be more easily be adapted into the game engine without the need to develop an application program interface (API) or wrapper, which would have detracted from the allotted time developing the game engine.

[1]It is trivial to utilise another compiler, such as g++. Instructions to switch the makefile to gcc rather than clang++ are contained in the wiki as well as the comments within the project's makefile.

Furthermore, an overwhelming number of video games have been and are written in the C/C++ languages for not only historical reasons but also due to performance optimisation and the abundance of APIs [27].

Code style

One of the project outcomes to provide an extensible game engine for educational outreach. Therefore, the development of the project included the adoption of a coding style with the intention to provide contributors a stylistic standard, which would increase code readability and uniformity. To this end, the coding style used for the development was the Google C++ style guide [26]. This particular style guide was chosen because it is widely available and provides in-depth documentation.

5.1.2 Multimedia library

The project requires a multimedia library in order to provide graphical output and player input. For this purpose, there are a number of suitable multimedia libraries which can provide graphics/sound management and user input. The primary criteria in selection was based on (i) ease of use, (ii) in-depth documentation, and (iii) rapid integration with the rest of the game engine.

The candidate libraries identified for the project were:

- Simple and Fast Media Library (SFML);
- Simple DirectMedia Layer (SDL); and
- Allegro.

These libraries were identified during the project as being among the most common choices for C++ video game development.

The decision to use SFML was based on the use of C++. SFML distinguishes itself from the other two libraires in that it is primarily written for C++ and thereby uses its multimedia library in manner consistent with object-orientated programming. Another strong choice was SDL. However, SDL lacks the support for object-orientation that SFML provides.

Use of existing games

Given that there exists an extraordinary amount of 2D platformer game source code openly available, one possible approach considered for this project was to utilise an existing code base as a foundation and develop modules for the level- generation and player movement solution. Conceptually, this would reduce development time spent on more basic elements of a game engine, allowing the focus to remain on integration of SAT instance solving and level generation.

Potential candidate source code had been identified, which included the *Secret Maryo Chronicles* [49], an open-source clone of *Super Mario Bros* and *Nikki and the*

Robots is an open-source 2D platform game, with content generation written in Haskell [29]. Each candidate was given a survey of the source-code and assessed for feasibility.

Ultimately, the decision had been made to develop a bespoke game engine as the project goals are quite specific and require far less code that provided by the candidate code bases. Furthermore, the time saved developing the code is re- allocated to learning the existing code base. As previously mentioned, one of the project's outcomes is to provide a specific learning tool that can be extended for the purposes of understanding computational complexity.

5.1.3 Mapping the game engine to *Super Mario Bros*

The game engine models the level on the framework provided by Aloupis, using the *Super Mario Bros* (1985) game elements.[2] In the game, the player is Mario, a plumber who traverses the game world in order to save the Princess. The player completes levels by overcoming obstacles through jumping onto different platforms whilst avoiding damage from enemies and hazards.

Within a level, Mario can receive power-ups which can provide the player with the ability to increase in size and smash certain types of blocks (Super Mushroom), shoot a projectile (Super Flower), or be rendered invincible for a short period of time (Super Star).

For this project the *Super Mario Bros* game was selected because of the game's popularity, making the medium relatable to a wider audience. Even if the game is not known by the viewer, the concepts are simple and easy to understand. Furthermore, the previous literature on the computational complexity of video games included *Super Mario Bros* which gives the theoretical underpinnings and validity to the game engine as well as opportunity to test an implemetation of the proof.

5.1.4 SAT solver integration

The game engine relies on the successful integration of a SAT solver.[3] The SAT solver's role is to read in the formula, determine its satisfiability, and, where satisfiable, provide the solution. This aspect of the game engine is essential to the success of the research goals.

Selection criteria

The selection criteria for the SAT solver centred on (i) ease of integration (ii) ability to manipulate the existing code in order to retrieve the relevant information for the level generation and the player AI. In order to meet these criteria, a small review of existing SAT solvers was carried out. This task included identification of sufficient literature and documentation and examination of SAT solver source code. The review of solvers identified three potential SAT solvers:

[2]The *Super Mario Bros* name and franchise is a registered trademark of Nintendo Corporation Ltd. The graphics and names are reproduced in this paper under the 'fair dealing' as it pertains to the private research and study.

[3]See Chapter 3 for a fuller discussion on SAT solvers and general functionality.

- zChaff [21, 42];
- Minisat [55];
- SATO [65];

Each one of the above listed has been successfully integrated into other projects and had sufficient literature and documentation provided. The processes ended with zChaff being selected as it had well-managed documentation as well as a pre-defined library of functions in order to export to an existing project. This was not true for the other two solvers, which would have required futher development time to spent on creating a wrapper to access the solvers' function.

The `ZChaffManager` class

The `ZChaffManager` class is the interface between the game engine and the zChaff library. The class is comprised of a suite of member functions which read in the DIMACS file, extract the variables and clauses, as well as provide a satisfying assignment, if one exists.

Where other classes require this information, the `ZChaffManager` is able to export a more condensed `VariableManager` class which is a flexible, light- weight means by which to access and manipulate the above information. The `VariableManager` class plays a significant role in level generation (Chapter X) and player AI (Chapter Y) and each class has different requirements.

5.2 Overview of modules

The game engine design integrates several modules with specific functionality in order to transform the arbitrary SAT instance into a playable, solveable video game level. This section provides an outline of the game engine and its functionality. Understanding these components will provide the context to investigate the level generation and player AI modules.

There are several fundamental components to the game engine. These include the SAT solver, the Level Generator, and the Player AI. These are supplemented by other components responsible for lower level functions such as graphics management, sprite definitions, and game states which are incorporated in the general 'Game Engine'. The interconnectivity between these higher level and lower level components is shown in Figure 5.1.

The modules are designed to be as de-coupled and independent as possible, so as to allow for greater independent development and more effective debugging.

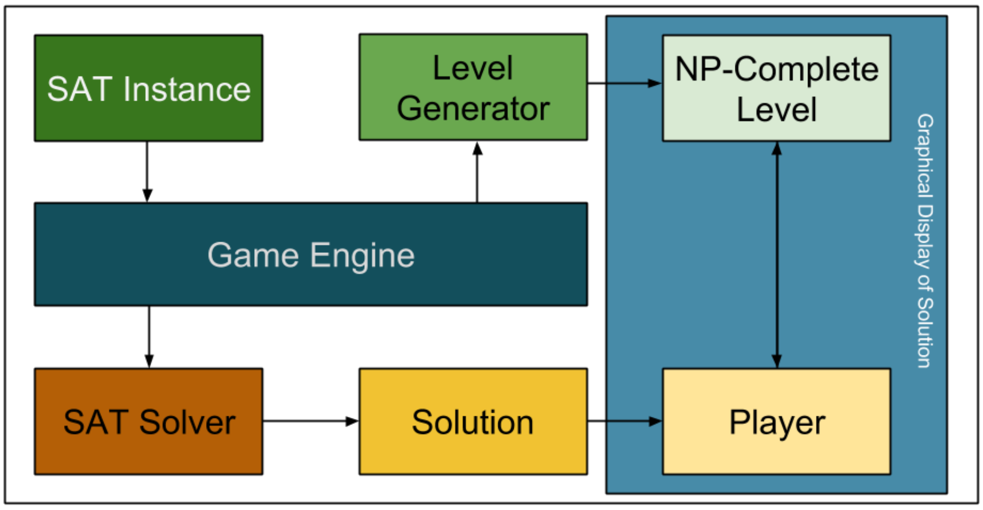


Figure 5.1: Diagram demonstrating how the game engine modules integrate with one another to generate the level instance and provide the mechanism to provide a graphical solution.

5.3 Game engine mechanics

5.3.1 Game loop and updating the game

A *game loop* provides the basic structure to implement a graphical game as it provides the ability to update the sub-systems of the game and render those changes to the screen [27, 30]. The loop is updated with a change of time since the last event (signified as the variable `delta_time`). As the game engine will incorporate player AI, which will force the player sprite to solve the level, it is necessary that the movement and rate of change is uniform.

This uniformity of change is achieved by creating a fixed `delta_time` value through creating fixed time steps. Variable time steps would be more appropriate in game loops that have a wider diversity of game object's updating [27].

The sprite and map classes are designed as tree structures that can be extended to have child nodes pertaining to their particular class. This design, adapted from [30], allows for the game to be extended to incorporate more features. The SceneNode class is extended into game entities and objects. The inheritance allows for specialising certain class types while retaining the core functionality and integrity of fixed time step in the game engine design. These nodes are initialised and added upon the level generation. This is explained in further detail in Chapter LEVEL GENERATION.

5.3.2 Game states

The game engine operates on game states. These states are kept by the `game_state_manager` class in a private `state_stack_`. Upon intialisation of the game engine, the GSM registers and populates the stack. The game engine has distinct states:

- menu staet

- instance selection
- game state
- pause state

In order to change the state, the current state requests to be 'popped' (i.e. removed) from the stack and 'push' (i.e. add) the next desired state.

Changing states

The game engine states with the menu state. The menu state contains a select instance option, a input option and an exit option. The first option takes the user to a screen that has four arbitrary instances to demonstrate the game engine's capabilities. The first is a single variable, the second is two variables in two clauses. The third is a 3SAT instance and the 4th instance is ten variables in ten clauses.

The pause state can be activated during the game _state as well as resume the game state.

5.3.3 Managing input

The input to the game is managed by the `player_manager` class. The input relies on assigning directional movement to the `mario` sprite class. The `player_manager` has a private associative map, `action_binding` which has predefined movement values. This is made extensible for human input by being able to be assigned to the keyboard via the `key_binding` member of the class.

The `player_manager` pushes these action bindings into the `comamand_queue` object which is then

This percise mechanism of selecting the appropriate path is detailed in Chapter X, TITLE.

5.3.4 Displaying to screen

The `level` class contains the mario node, the mapnode and could be extended to have a world object node. Scene nodes are tree structures that draw and interact.

The node receives a delta_time variable which is the change in time from the last update.

This is given to the scene node and any objects that have updated will.

SFML provides for vertical synchronisation, which sets the call rate to the display function in the library to be that of the monitors refresh rate. In this way, the display of artefacts such as 'screen-tearing', wherein the display shows the information of two frames at the same time [30].

5.3.5 Miscellaneous functionality

Mini-map

The game engine display also includes a miniature map that contains the whole level. This allows the viewer to see the scale of the level without

Zoom

User input

This is further detailed in the Evaluation chapter.

5.4 Chapter summary

The game engine's design considerations and specifications have been grounded in the concepts of re-usability and extensibility of the produced code in mind. The core of the engine provides the necessary mechanisms for being able to generate levels and display the solution, which will be covered in the next chapters..

Chapter 6

Level generation

"Human beings are never more ingenious than in the invention of games."

Gottfried Wilhelm Leibniz

6.1 Overview and specification

One of the core featues of the game engine and stated research project goals is to be able to generate game levels based on an arbitrary SAT instance. As mentioned in the previous chapter, within the research project's game engine, a 'level' is a problem instance for the SAT solver to assess if there is a satisfying assignment [1] .

6.2 The fundamentals of content generation

In order to be able to generate game elements, the project utilises *procedural content generation* (PCG), which refers to automating the creation of content for a game through the use of an algorithm [31, 60]. In this context, *game content* refers to all game elements, including the translated SAT gadgets, excluding the player's own actions and the game engine [6]. In the game engine, this means

- level layout
- generation of interactable objects

6.2.1 Fundamentals of procedural content generation

Content generation in video games combines a broad range of dynamic subject areas including graphics, image processing, mathematics, artificial intelligence, and ludology [31]. PCG is increasingly popular with recent titles employing elements of PCG to varying degrees. This is due to being able to reduce the prohibitive expense of manually creating game content [31] and the ability to generate new types of games based around content- generation [60]. From a game player perspective, a game that has infinite levels has greater potential replay value, especially if the game adapts to the player's playing style [64].

There are other, more practical reasons such as memory consumption. The content. This has been used to tremendous effect in *.kkrieger*, a first-person shooter game, whose executable was a mere 95 KiB [31].

For the purposes of the project, PCG allows for the ability to modify the level instance, complete with the requisite gadgets, based on an arbitrary instance of a SAT problem. This would remove the need to explicitly specify the game content and could rely on algorithmic generation based on defined rules and limitations.

6.3 Taxonomy of game elements and SAT instance gadgets

This section provides an overview of broad definitions which will be used to characterise the game engine's level generation mechanism.

Online vs offline generation

There is a distinction between *Online content generation* occurs during a video game's runtime. This is in contrast to *offline content generation* which is performed during the development of the game. The requirements for effective online content generation are (i) that generation is fast (ii) the content is reliable and correct [60].

Necessary vs optional content

Another set of distinctions is between *necessary* and *optional* content. Necessary content is required for the progress and completion of the game. Optional content is content with which the player may wish to engage but does not prevent basic gameplay or completion of the game itself.

The importance is that necessary content must always be generated in a correct manner. For example, a level must not be unsolvable nor should the game have an undefeatable enemy, if such things prevent essential gameplay or render the game impossible to complete.

6.3.1 Requirements for Game Engine Level generation

The project's game engine requires online content generation of necessary game content. The elements need to be generated and are necessary for the successful run of the game.

6.3.2 Challenges in PCG

Content generators can automatically create a large amount of differing levels in a short amount of time but this is not a guarantee of the *quality* of the generated content [52].

Existing literature regarding PCG identifies a lack of reliability and consistency in quality as a main challenge area and barrier to wider adoption of PCG in commercial game offerings. Zafar and Mujtaba [64] assert that the majority of PCG techniques are characterised by what they term *catastrophic failures*, which make them unsuitable for wider commercial deployment. These failures affect the reliability and accuracy of the game. The work identifies the following as specific failures in 2D platform games:

- height of ground;
- height of ground and optional content;
- simple level generation
- unplayable levels
- placement of the character's start position

The presence of these failures in the game engine would result in a lack of communicability and visualisation of computational complexity. It is therefore imperative that these failures be specifically guarded against and tested for.

Protecting against failure

The generation algorithms may be tested through the use of *fitness functions*, which are objectively evaluates how closely a proposed solution achieves a particular design aim. There is a question about which fitness functions are useful and what sort of rules should be employed when generating level. For example, Togelius *et al.* [58] developed personalised fitness functions that would automatically generate different racetracks based on the player's actions and skill.

6.4 Content generation methodologies

A number of approaches exist for content generation, of which X of the more relevant were considered for level generation module of the game engine. Three differing approaches will be surveyed and compared and evaluated for suitability for use in the research project (see Figure 6.1).

The simplest methodology is the constructive approach, wherein an algorithm generates content based around a series operations that ensure that content is produced according to the definitions of rules. In this way, the content generated is 'correct' insofar as adherence to the specified rules [4]. The algorithm does not test that the content has in fact adhered to the rules however. While this method is restrictive, it has been used to generate less vital game content such as terrain [41].

The other approach considered is 'generate-and-test' approach. As suggested by the name, there are two parts to this algorithm: After the content is generated, it is then tested against constraints and assessed for compliance [60]. Content that does not pass assertion against these criteria is discarded and is regenerated until it passes [60].

Search based generation is a specialised type of generate-and-test procedure. Rather than merely accept or reject the generated content, the candidate content is tested through a fitness function. The generation of new content is dependent on the fitness value assigned to the prior evaluated instances. The goal is to produce content that is qualitatively of increased value [60].

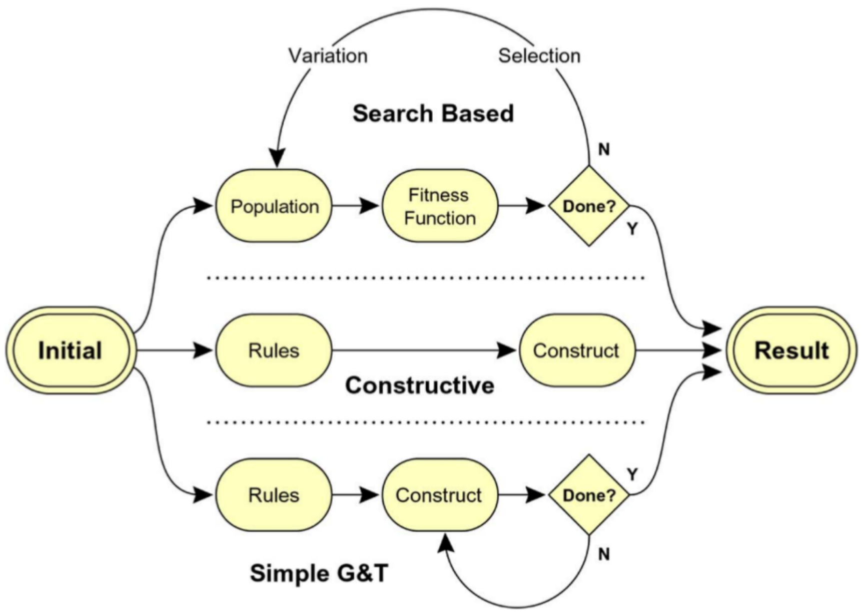


Figure 6.1: Representation of three types of level generation [60].

6.4.1 PCG heuristic techniques

In each of the approaches above, there exits a difference between deterministic and stochastic methods of generation. In the former, the generation tool will generate specified content on each output. In a stochastic method, the results are somewhat random.

Dormans and Bakkes [17] presented a generative grammar which allows for the generation of levels which are 'correct' on a syntactic level. This was achieved by dividing the content into mission and spaces. The missions (or game tasks) are generated through the use of recursive, non-linear graphs. From this output, the structure and shape of the level are constructed from a separate 'shape' grammar. The authors also utilised differing player models in order to dynamically adjust difficulty, resulting in more adaptive levels.

In a survey on the the evolution of game designs, Togelius and Schmidhuber [59] posit that 'fun' in a game is derived from learning how to play and ultimately mastering the mechanics. While this may be true to an extent [6], the declaration and incorporation of these constraints into an algorithm for generation is limited insofar as the design matches human experience [54].

6.5 Designing NP-complete levels

Implementing the frameworks established in literature

The level design implemented in the project uses the model in Aloupis *et al.* For the purposes of this project, the size of the levels are $n \times n$, which means that the size is essentially unlimited.

Figure 6.2 demonstrates how the gadgets may be phsyically linked together in a three variable, three clause 3SAT reducation.

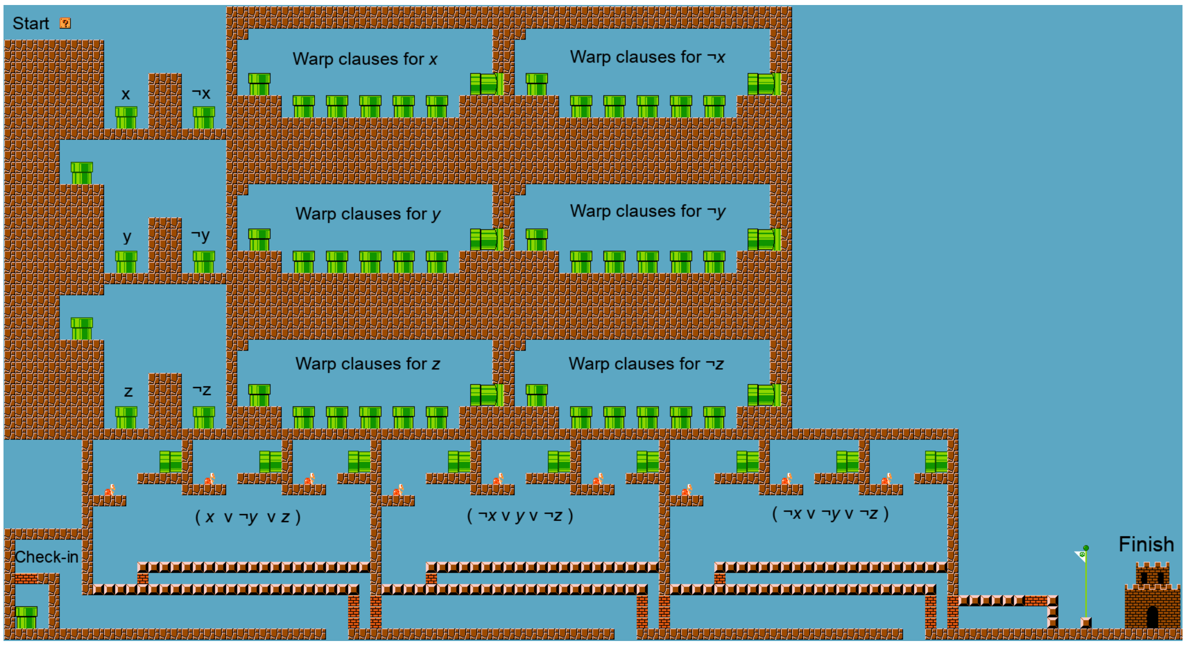


Figure 6.2: Representation of physical linking of gadgets.

This will be briefly described below.

The player starts as in the framework described by Aloupis *et al*[1] with a start gadget. In the current game engine, there is no necessity to have an animation that turns 'small' Mario (composed of a single 16x16 tile) into Super Mario. Super Mario can jump under bricks and break certain types of blocks.

The variable gadget relies on the meta-theorem provided by Forišek [20] (long fall). In Aloupis et al [1], the authors created a cross-over gadget that prevented the player from by- passing or accessing areas of the level which impact the reducation to a 3SAT instance. The authors are not clear exactly how the gadgets are physically linked together in order to create a complete functioning level. In particular, the This presentes a number of challenges when scaling a level. In Aloupis, the 3SAT instance has three clauses. However, when dealing the amount of cross-over instances increases.

In order to efficiently

In *Super Mario Bros.*, the player may discover that certain pipes are able to be utilised in order to travel to different hidden areas or even other levels. This game mechanic provides an immediate solution to scalability for the purposes of the project's development.

In the project's implementation, after the player picks a variable by falling and using the pipe, the player is taken to a clause choosing gadget. Here there are a series of pipes, each leading to a particular clause where the literal appears.

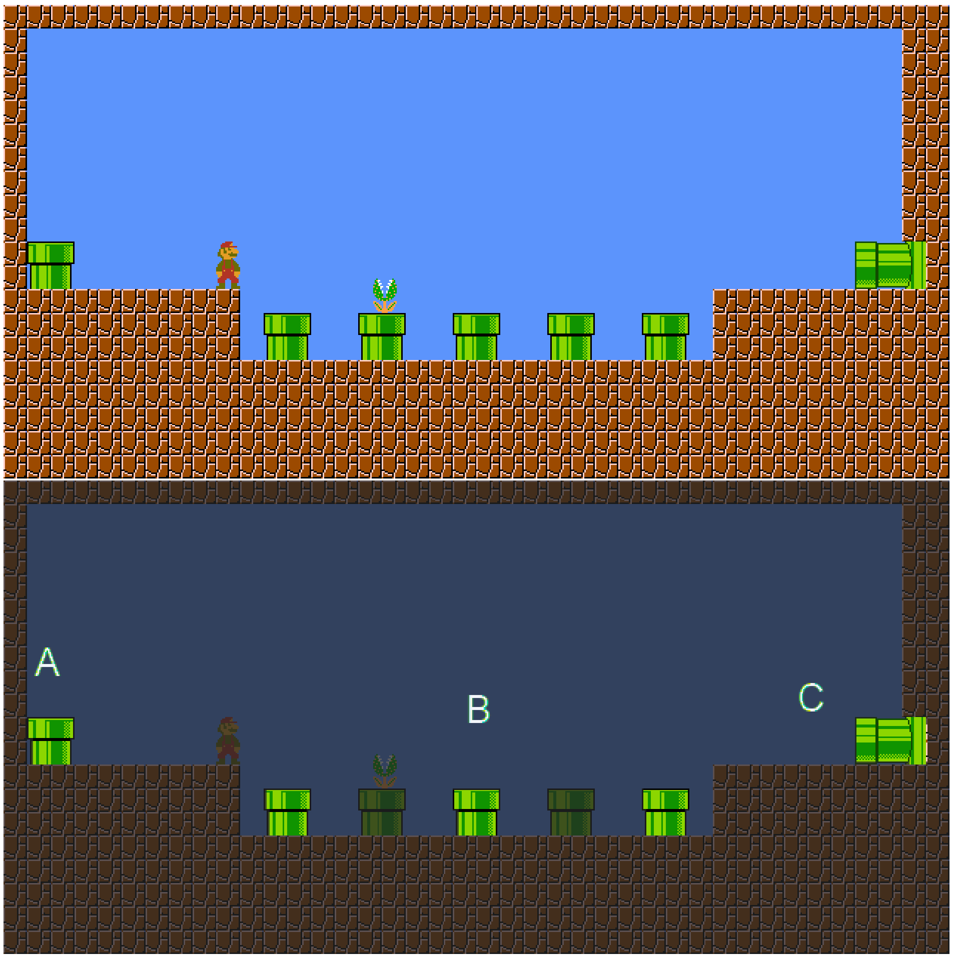


Figure 6.3: Crossing into other clause areas.

In the clause-crossing gadget, the player enters on the pipe on the left (marked A). The player is not able to return through this pipe. Even if it was the case, the player cannot overcome the variable gadget.

There are five pipes of which three lead to clauses and the other two contain *Piranha plants*, venus-fly trap like enemies in *Super Mario Bros* (Marked B). Once the clauses have been visited and unlocked, the player returns through the pipe on the right side of the screen (marked C) and can proceed to the next variable assignment. If the player has no more variable assignments to make, the pipe will lead to the check-in gadget, illustrated in Figure 6.4. The check-in gadget allows the player to progress through the check-out and ultimately to the finish gadget.

Once the player checks-in, the player then advances through the check-out of the level. The check-out phase is only passable if the clauses have been opened. The only obstacle at this point of the game is the drop in the floor. The player then proceeds to the finish gadget, shown in Figure 6.5.

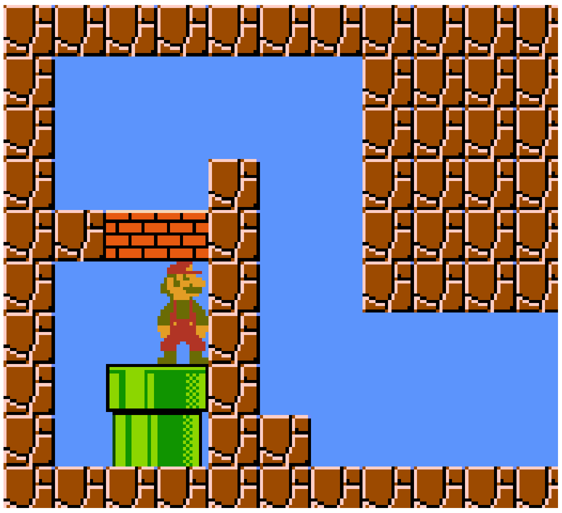


Figure 6.4: The check-in gadget. Note the player can only advance as Super Mario.

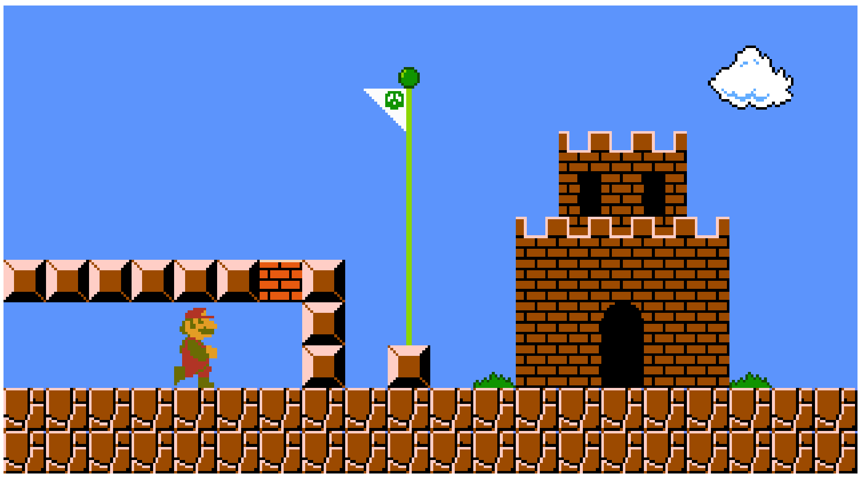


Figure 6.5: The check-in gadget. Note the player can only advance as Super Mario.

6.6 Chapter Summary

There exist innumerable means of achieving online generation for necessary game elements. In this project the approach has been to utilise a generate-and-test methodology for the game elements. 7

Chapter 7

Player AI

"It's not hard to make decisions when you know what your values are."

Roy Disney

7.1 Overview

Having established the procedural generation of the game level based on the SAT instance in the previous chapter, this chapter focuses on the game engine's player AI. The player AI directs and controls the player sprite's movement and path selection, which corresponds to the solution to a SAT instance.

7.2 Identifying the appropriate path

As descibed in the above section (SOME SECTION) in Chapter (Game Design), the movement of the player sprite is controlled and regulated by the player-manager class. To briefly summarise, the player-manager class sends a queue of commands (via the command-queue object) to the level class, where the movements are rendered to the screen.

7.2.1 Receiving SAT instance values

After the initialisation of the game-state, the level class loads the zchaff manager nad the variable manager, solving the arbitrary SAT instance.

If the ENTER button is pressed, these objects are set as values within the player-manager, in order for the

The player-class has a var-manager member.

7.2.2 Generating commands

The player AI relies on `command` objects which, when executed, move the player sprite to the appropriate location and identify to both the level instance and the player manager which gadget is being solved and where the location is provided. This section provides an indepth look into the mechanics of commands and how these are generated within the player-manager.

Overview of command structures

A command is a data structure comprised of a `Location`, an enumeration of game gadgets, and an `Action`. The `Action` is a type alias for a class template for a polymorphic function wrapper for a lambda expression. This is provided in the C++11 standard. `Action` takes a `SceneNode` and `delta_time` as arguments. These are downcast in some manner that needs explaining.

SFML BOOK NEEDED

The available actions

The actions that are available to the player-manager to move the sprite in the appropriate direction are:

- MoveRight
- MoveLeft
- Down
- Jump
- Wait

As mentioned in the previous chapter, the level instance is generated on a 16×16 tile world. The MoveRight and MoveLeft movements move 2.f or 1/8th of a tile space per movement. The Down movement represents a fall and consists of 16.f (e.g. a whole tile). Similarly, the Jump action displaces the player- sprite a whole tile as well. This means that the sprite's jump is exactly twice its height. This is in keeping with the meta-theorem of characteristics of 2D platform video games that are NP-complete.

Identifying the correct clause location

The display of the solution requires that the assigned variable satisfy the appropriate clause when it evaluates to true in that clause. The player sprite should not warp to those clause gadgets where the variable assignment does not render the clause true. In order to achieve this, the player-manager sets the appropriate location of each assigned variable prior to providing the requisite commands.

Once the solution display has been initiated, the level instance passes a pointer to its command-queue member to the player-manager through its public `SetSolutionQueue` member function. Within this function, the player- manager initialises its assigned-variables- member, which sets the assignment solution and passes this to the `SetAssignmentLocation` member function. Here the player-manager iterates through the clauses and where the current variable being assigned matches one in the clause, this is pushed onto the location-map.

The location-map is a 2D vector structure. Each row corresponds to the variable. Each column in that row contains a value which corresponds to the clause which contains that assigned variable.

Providing the requiste information to each command

The player-manager class has a number of member functions that have pre-defined movement elements to traverse each type of game gadget. The pre-defined elements concern length of walk, jumps, falls and waits. These are then added in a for-loop with the necessary information about location, current variable, current clause, and whether there is an animated action sequence involved.

```
// Move the sprite right
for (int i = 0; i < 8; ++i) {
  Command c           = action_binding_[MoveRight];
  c.location_         = c.Clause;
  c.var_assignment_   = current_variable_;
  c.current_clause_   = target_clause;
  c.has_action_       = false;
  commands.Push(c);
}
```

After the location-map has been generated, the command-queue is passed to `InitStartQueue`, which provides the start sequence. This then passes the command-queue to the Warp Gadget, then to the Clause gadget. If there are no clauses left, the player-sprite proceeds to the warp exit and to the next variable. The process starts again.

Where there are no more variables to solve, the warp exit will take the player-sprite to the checkin gadget, where the checkout process commences. After clearing the checkout, the player proceeds to the finish gadget and the instance is solved.

Making an assignment decision

The gadgets in the game require a decision to be made regarding the assignment of variables. This means that the player-sprite may choose left for a TRUE assignment or right for a FALSE assignment. This is achieved by assessing the current-variable member and initiated the correct sequence which has been predefined.

7.2.3 Positioning the sprite in the level

Having established the correct commands for the appropriate game gadgets, the player-manager class returns the command-queue to the level instance.The level instance will take current command and, depending on the Location enumeration, positions the sprite in the appropriate position as well as trigger and world object events.

The positioning of the player sprite is achieved through its update

Travelling between the warp and the clause gadgets

The player-sprite has a location member variable that is updated upon every move. Where the command requires that the player warp to the clause gadget, the level instance keeps the location prior to the warp. The player-sprite is then warped to the clause gadget and the actions are performed there. Once that action is performed the player return through the pipe and is re-assigned the prior jump position, ensuring that the player-sprite is returned appropriately and the game actions adhere to the game mechanics and logic.

7.3 Chapter summary

The player AI utilise the SAT instance's specifics as well as the definitions of the variable gadgets in order to determine the appropriate course of action. This is achieved through the command structure which passes the necessary information between the level instance and the player-manager class.

Chapter 8

Implementation and evaluation

“ One of the great mistakes is to judge policies and programs by their intentions rather than their results”

Milton Friedman

To this point, the discussion has focused on the theoretical background of computational complexity of video game, followed by the design considerations and model of the game engine. This chapter therefore provides analysis of the analysis of the functionality of the developed game engine and evaluates the results against the project’s success criteria.

8.1 Establishing assessment criteria

Within a project that is strongly focused on implementation, the assessment of whether the development has been successful lay within the domain of software evaluation methodology. To this end, there exist a number of evaluation methodologies for software development [62]. These range from a series of tests to examine the qualitative merit of functionality and ability to meet specification.

The assessment methodology employed in this project has been to determine whether the software produced met its original goals. These goals have been consistently reiterated since the inception of the project and formalised as the design specification.

8.1.1 Evaluation criteria for this project

At the start of the project’s development cycle, an evaluation criteria was developed in order to track as objectively as possible whether the project had met its goals. This had been established prior to any code being written as it objectively signalled when progress had or had not been made.

The success of the project is comprised of the following elements:

- the game engine should be able to take an arbitrary SAT instance and transform it into a video game level modelled on a framework establishing the NP-completeness of a 2D platform video game;

- the game engine should incorporate a SAT solver in order to solve the SAT instance;
- if a satisfying assignment to the instance exists, then the player AI should identify the correct path to solve the level.
- the game engine should not experience catastrophic failures in its content generation
- the game engine should be extensible and adaptable to other SAT instances and complexity classes.

8.2 Generating a graphical solution display

After the initial game loop and drawing mechanisms were achieved, the project's attention turned to SAT solver integration and graphical display. To examine differing approaches, prototypes were devised and tested. In the Graphical SAT experiment, a simplified game loop was developed.

The zChaff library was then linked to the game loop and variable objects were created. The variable object class contained a shape and an assignment value. The zchaff manager class would have a sequence container (i.e. `std::vector`) which would load the variable objects in the manner in which they appeared in the formula.

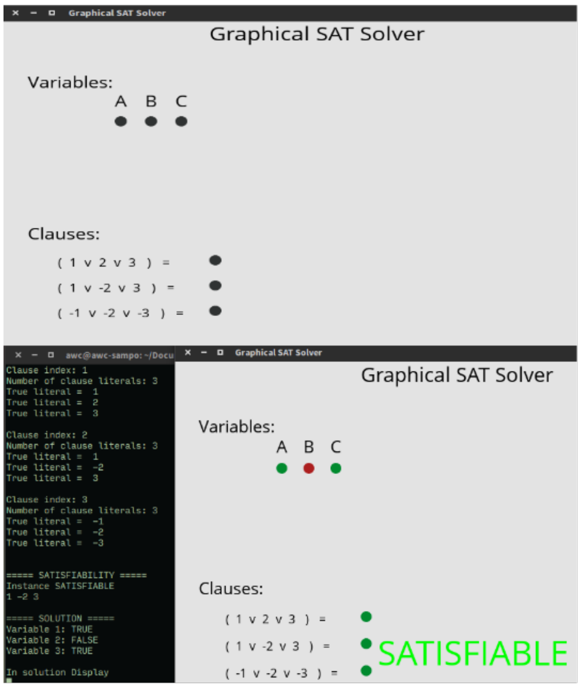
The game engine would then step through each assignment and change the colours of the gray circles to green (for TRUE) or red (for FALSE). When a clause was satisfied, the grey circle near the clause would light up green. Once all clauses evaluated to true, the game engine would declare the formula satisfiable on screen, as illustrated in Figure 8.1 below.

8.2.1 Graphical SAT experimental outcomes

This experiment had implications for both the level generation and the player AI. The level generation module benefitted from understanding how the zchaff manager could be extended to read in the correct values and store these in what would become the variable-manager. Work from this point went on to the MapChunk and MapChunkManager classes (described in the next section).

Non-scalability of drawing methodology

The display is drawn to the screen in a sequential manner through a simple do- while loop. This drawing implementation was appropriate for this rudimentary, experimental prototype, but was identified as not being scalable to the game engine. This is because the do-while loop requires constant calls to the draw() function which intereferes with the game loop design and fixed time steps (see Chapter X).



```
Clause index: 1
Number of clause literals: 3
True literal =  1
True literal =  2
True literal =  3

Clause index: 2
Number of clause literals: 3
True literal =  1
True literal =  -2
True literal =  3

Clause index: 3
Number of clause literals: 3
True literal =  -1
True literal =  -2
True literal =  -3


===== SATISFIABILITY =====
Instance SATISFIABLE
1 -2 3

===== SOLUTION =====
Variable 1: TRUE
Variable 2: FALSE
Variable 3: TRUE

In solution Display
```

Figure 8.1: An experiment in displaying the satisfiabilty solution to a SAT instance. The top screenshot illustrates the unassigned variables within the formula towards the top and the clause representations on the bottom. The bottom screenshot shows the correct assignment in the terminal and the corresponding graphical representation.

8.3 Level generation verification and testing

Having successfully been able to sequence the satisfying assignment and display to screen, the next development focus was the MapChunk and the MapChunkManager. The specifics of these classes and their functionality are detailed in Chpater LEVEL GENERATION.

To test and assert that the specific map chunks were (a) being correcty loaded and (b) correctly called, were tested in the MapChunkTest.cpp file. The test loads mapchunk strings into a 2D vector. These are then tested against those read into game engine.

The test checks if there is an equality of values in the test vectors and those of the game engine. If the so, the value should return true. If such equivalence is not present or there is an error, the returned value should be false.

This form of testing allowed for the verification that the MapChunkManager was correctly perfoming its role and allowed for further implmentation in level generation and player AI development.

//////SOME KIND OF GRAPHIC /////

8.4 Graphical element positioning and offsets

A consistent challenge in the implementation of the game engine software was managing the positioning off sets in an extendable and flexible level generation. Where a world object (e.g. koopa shell, brick, or flag) needed to be drawn

The tests for this occurance where assertions within the code (i.e. expecting certain elements to be drawn in certain places) and visual inspection. As mentioned in previous sections, positions of objects are provided in as float values in SFML so as to provide precision. However, the pixels are drawn in integer values. The rounding of the values is done internally within SFML, so if certain offsets are desired, these have to be provided before being drawn to screen.

8.5 Verification of formula satisfiability

One of the most crucial features of the. To achieve this, the satifying assignement of the formula is provided by zchaff, this value is then checked that it is consistent throughout the game engine and the output is able to be visually inspected in the terminal.

8.6 Animations

Animations for the sprite and events had also been addressed during the implementation and assessment of the game engine. The game engine includes modest and simple animations in order to display in a more relatable manner to audiences. This aspect of the game engine was considered a lower priority, as animations and their implementation were not explicit research goals, being incidental to the creation of a game engine. To this end, animations were addressed insofar as being a means to faciliate visualisation.

The player sprite has a basic walk cycle that is composed of three frames. The play in sequence when the moving left or moving right Boolean flags are set in the Mario class. When there is no movement, the cycle resets to the standing frame.

To move in the opposite direction, the frames are mirrored so that the sprite now faces the opposite direction. This is beneficial in that

During the development an Animation class was added, based on tutorial material available [30]. This could allow for further extensibility by adding more animated Koopas or other eneimies.

8.7 Implementation analysis summary

As with any software implementation, the game engine development was beset by challenges which spanned trivial to fundamental in nature. This chapter serves as a highlight of the most significant milestones and issues that characterised the development of the implemetnation. The nature and evaluation of the obstacles and problems challenging the success of the project are further elaborated upon in the following chapter.

8.7.1 Assessment against evaluation criteria

Game Engine Evaluation		
Assessment Goal	Completed	Notes
Integrate suitable SAT solver	Yes	
Create Game Loop with fixed time steps	Yes	
Successfully read in DIMACS CNF file format	Yes	
Generate NP-complete level based on SAT instance	Yes	
Solve NP-complete level with adaptable player AI	Yes	
Identify the location of literals within clauses for AI	Yes	
Capture user input for arbitrary SAT instnace	AO	AGO

8.7.2 Identifying solutions

8.8 Alternative approaches to content generation and player AI

As with any software development cycle, a number of alternatives to the selected methodology are available and may have been better.

8.8.1 Using Binary Decision Diagrams

The project's implementation has relied on a SAT solver in order to provide a solution to the SAT instance. However, this is not the only means by which to solve the instance. Binary decision diagrams are data structures that represent Boolean functions.

/////wiki ///

s a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations.

///

This could be an alternative means upon which to base the player AI. Howver, there do exist certain issues. The use of BDDs is somewhat limited as the size of at run-time may increase a lot.

///// Unfortunately, their use is rather limited due to the size explosion problem. It has been observed that so long as the BDD size can be contained, solutions can be found quickly, often with predictable run-times [6] [7] [23]. ////

In literature, a SAT solver may be used or a BDD but not both. However, there are efforts to combine the two.

8.8.2 Alternative procedural content generation approaches

The approach taken in this

Establishing a scripting language would provide even further granularity.

Sorenson *et al.* [54] proposed that content generation should levels that is founded on an explicit model of the relationship between challenge and fun. The author wanted to increase the fun level of player and the challenges should not bore the human. Therefore he modelled a fitness function and levels were designed for *Super Mario Bros* with high accuracy. The technique insures that human will generate certain portion by hand and then the model used will evaluate the content generated – this evaluation is similar to the evaluation of automatic content. The levels will be a mixture of straightforward and difficult portion like the rhythm-group structure of high and low nodes, giving games more interesting levels.

Mourato *et al.* [43] proposed using a genetic algorithm in order to generate game content for human-players. A genetic algorithm is a search heuristic that generates approximations through selection and mutation [56]. The authors achieved modest but positive results including increased level variety and ability to add necessary and optional content correctly. This method is not entirely suitable for the current research project as the authors' goals has been to

Smith *et al.* [53] presented a design tool (TANAGRA) that merged the input from the game designer and the PCG algorithm to produce levels for a 2D platform game. The technique allowed the designer to establishes rules and constraints to specify specific content, but allows the algorithm to decide that which is unspecified. The authors content that this provides for more diversity in level design as well as enhanced entertainment value.

A rhythm-based technique is common approach to automatic level generation. Mawhorter and Mateas [40] devised the 'occupancy regulated extension' algorithm which stitches together samples (referred to 'chunks') authored by human developers into a playable level. This method described here could be combined with. This is further elaborated upon in the next chapter on design considerations.

Chapter 9

Conclusions and future work

"QUOTE."

Someone

9.1 Overview

9.2 Future work

Bibliography

[1] G. Aloupis, E. D. Demaine, and A. Guo. Classic Nintendo games are (NP-)Hard. *CoRR*, abs/1203.1895, 2012.

[2] T. Balyo. *Solving Boolean satisfiability problems.* PhD thesis, Charles University, Prague, 2010.

[3] A. Biere, M. Heule, H. van Maaren, and T. Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[4] C. Browne. *Automatic generation and evaluation of recombination games.* PhD thesis, Queensland University of Technology, 2008.

[5] R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 44–57. Springer, 2010.

[6] K. Burgun. *Game Design Theory: A New Philosophy for Understanding Games.* Taylor & Francis, 2012.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[8] S. A. Cook. Can computers routinely discover mathematical proofs? *Proceedings of the American Philosophical Society*, pages 40–43, 1984.

[9] J. Culberson. Sokoban is PSPACE-complete. In *Proceedings in Informatics*, volume 4, pages 65–76, 1999.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[11] E. D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *Mathematical Foundations of Computer Science 2001*, pages 18–33. Springer, 2001.

[12] E. D. Demaine, M. L. Demaine, and J. O'Rourke. PushPush and Push-1 are NP-hard in 2d. *arXiv preprint cs/0007021*, 2000.

[13] E. D. Demaine, R. A. Hearn, and M. Hoffmann. Push-2-f is PSPACE-complete. In *CCCG*, pages 31–35, 2002.

[14] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. In *Computing and Combinatorics*, pages 351–363. Springer, 2003.

[15] K. Devlin. *The Millennium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time.* Basic Books, 2002.

[16] D. Dor and U. Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[17] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):216–228, 2011.

[18] D.-Z. Du and K.-I. Ko. *Theory of computational complexity*, volume 58. John Wiley & Sons, 2011.

[19] B. Esfahbod. Euler diagram for P, NP, NP-complete, and NP-hard problem. `http://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg`, 2011. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported and 2.5 Generic and 2.0 Generic and 1.0 Generic license.

[20] M. Forišek. Computational complexity of two-dimensional platform games. In *Fun with Algorithms, 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings*, pages 214–227, 2010.

[21] Z. Fu, Y. Marhajan, and S. Malik. zchaff SAT solver, 2004.

[22] W. I. Gasarch. Guest Column: The Second P =?NP Poll. *SIGACT News*, 43(2):53–77, June 2012.

[23] O. Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[24] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[25] C. P. Gomes, B. Selman, H. Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[26] Google. Google c++ style guide, 2015.

[27] J. Gregory. *Game Engine Architecture.* Taylor & Francis Ltd., First edition, 2009.

[28] L. Gualà, S. Leucci, and E. Natale. Bejeweled, candy crush and other match-three games are (np-)hard. *CoRR*, abs/1403.5830, 2014.

[29] S. Hahn. *NIkki and the Robots.* `https://github.com/nikki-and-the-robots`, 2012. Date Accessed: 2015-04-28.

[30] J. Haller, H. V. Hansson, and A. Moreira. *SFML Game Development.* Packt Publishing, 2013.

[31] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, Feb. 2013.

[32] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[33] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 343–356. Springer, 2011.

[34] R. Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[35] G. Kendall, A. J. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

[36] S. Kottler. Sat solving with reference points. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 143–157. Springer, 2010.

[37] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[38] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence*, pages 62–74. Springer, 1999.

[39] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[40] P. Mawhorter and M. Mateas. Procedural level generation using occupancy-regulated extension. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 351–358. IEEE, 2010.

[41] G. S. P. Miller. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.*, 20(4):39–48, Aug. 1986.

[42] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[43] F. Mourato, M. P. dos Santos, and F. Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, page 8. ACM, 2011.

[44] A. Nadel. The jerusat SAT solver. *Master's thesis, Hebrew University of Jerusalem*, 2002.

[45] A. Nadel and V. Ryvchin. Assignment stack shrinking. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, SAT'10, pages 375–381, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] J. O'Rourke and T. Group. PushPush is NP-hard in 3D. *arXiv preprint cs/9911013*, 1999.

[47] C. H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK.

[48] C. H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[49] F. Richter. *Secret Maryo Chronicles*. `http://www.secretmaryo.org`, 2013. Date Access: 2015-04-27.

[50] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[51] G. Smith, M. Cha, and J. Whitehead. A framework for analysis of 2d platformer levels. In *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games*, Sandbox '08, pages 75–80, New York, NY, USA, 2008. ACM.

[52] G. Smith, M. Treanor, J. Whitehead, and M. Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 175–182, New York, NY, USA, 2009. ACM.

[53] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):201–215, 2011.

[54] N. Sorenson, P. Pasquier, and S. DiPaola. A generic approach to challenge modeling for the procedural creation of video game levels. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):229–244, 2011.

[55] N. Sorensson and N. Een, N.n. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

[56] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[57] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977.

[58] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 252–259. IEEE, 2007.

[59] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 111–118, Dec 2008.

[60] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, Sept 2011.

[61] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[62] M. Unterkalmsteiner, T. Gorschek, C. K. Cheng, R. B. Permadi, R. Feldt, et al. Evaluation and measurement of software process improvement—a systematic literature review. *Software Engineering, IEEE Transactions on*, 38(2):398–424, 2012.

[63] G. Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

[64] A. Zafar and H. Mujtaba. Identifying catastrophic failures in offline level generation for mario. In *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pages 62–67, Dec 2012.

[65] H. Zhang. Sato: An efficient prepositional prover. In *Automated Deduction—CADE-14*, pages 272–275. Springer, 1997.

[66] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36. Springer, 2002.