TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Teemu J. Heinimäki

Technology Trees and Tools: Constructing Development Graphs for Digital Games

Julkaisu 1349 • Publication 1349

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1349
Tampere University of Technology. Publication 1349

Teemu J. Heinimäki

Technology Trees and Tools: Constructing Development Graphs for Digital Games

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB224, at Tampere University of Technology, on the 27th of November 2015, at 12 noon.

Tampereen teknillinen yliopisto – Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3633-5 (printed)
ISBN 978-952-15-3647-2 (PDF)
ISSN 1459-2045

Abstract

In the recent years, digital games have solidified their role as important parts of life for a considerable portion of the population. Game development has become an extremely important industrial branch with a great deal of competition between developers and publishers. There is only a limited amount of resources to put in the development of a game, but the modern customers expect high quality.

Taking these constraints into account, this dissertation focuses on developing implementations of a structure that is used widely in different games: technology trees (TTs). This term covers here also so-called skill trees, talent trees, perk trees, and other such structures used to limit and guide in-game development and define development possibilities. The aim is to propose methods and usage of tools helping to achieve high TT quality, simultaneously facilitating the actual development process and reducing human workload.

The main contributions of this dissertation consist of ideas, models, methods, and software tool prototypes constructed during the research work. The significance of the thesis is amplified by the fact that there are only very few previous academic studies focusing on TTs.

The thesis proposes a generic approach to implement TTs. The design and implementation work are facilitated by tool support and automated code generation. The central prototype tool, Tech Tree Tool (TTT) is introduced, first in its core form and then as improved by TT measuring (and limited automatic adjusting) capabilities. The challenge of modifying TTs during runtime is addressed, also taking advantage of related improvements on TTT. Because TTs are often operated by artificially intelligent entities, discussion on a generic artificial intelligence approach and related tools is included. Moreover, contemporary real-life TTs are analyzed and generic TTs characterized.

Keywords:
code generation, scripting, software tools, tech trees, technology trees

Preface

We are living in extremely interesting times. Lately, the academic community has finally started to take research efforts concerning digital games somewhat seriously. Although there are still people uncertain about this field, the situation keeps changing. Digital gaming evolves and expands forcefully to new areas such as education. Gamification has become a buzz word. So, I consider myself fortunate for having been able to step on the shoulders of giants and start my research career in these exciting circumstances.

My research work started a couple of years ago under supervision of Professor Tommi Mikkonen. I learned much working in his projects and appreciate his contribution for my initial research efforts. Professor Tapio Elomaa has been my supervisor throughout most of my graduate studies. His expertise and guidance have continuously been invaluable assets. I would also like to express my gratitude to my colleague Doctor Juha-Matti Vanhatupa for inspirational discussions and his contributions for joint publications, one of which is also included in this dissertation. During the pre-examination phase, I got excellent advice from Doctor David W. Aha, Professor Ulf Johansson, Professor Peter Quax, and Adjunct Professor Jouni Smed. They deserve special thanks for their efforts, as does my opponent, Professor Erkki Sutinen.

Writing the doctoral thesis would certainly have taken more time, had I not have the opportunity to make my living by solving the research problems. Fortunately, I had the chance to carry out my doctoral studies and the thesis-writing process working with the late Department of Computer Systems and later with the Department of Mathematics of Tampere University of Technology. I would hereby like to thank everyone involved in creating

and maintaining the stimulating and pleasant working environment. In addition to the funding provided by these departments, my work was financially partly supported by the Tampere Doctoral Programme in Information Science and Engineering, the Academy of Finland, and the Foundation of Nokia Corporation.

The nature of research work is such that it cannot really be confined within office rooms or predetermined office hours; it is more like a way of living than a daywork. Therefore, it also casts a shadow on private life and free time. I would like to apologize for being irritating and thank all the affected persons for their understanding and patience so far.

Tampere, October 20, 2015

[illegible]

Teemu J. Heinimäki

Dedicated to Pörri and Ninjakissa.

Contents

List of Figures

Abbreviations

3D	three-dimensional
4X	eXplore, eXpand, eXploit, and eXterminate
ADR	action design research
AI	artificial intelligence
Civ	*Sid Meier's Civilization* (MicroProse 1991)
Civ 5	*Sid Meier's Civilization V* (Firaxis Games 2010)
CLI	command line interface
CO	collectible object
CRPG	computer role-playing game
FSM	finite state machine
GOAP	goal-oriented action planning
GTR	graphical interactive technology tree representation
GUI	graphical user interface
LOD	level of development
NPC	non-player character
PC	player character
RQ	research question
RTS	real-time strategy
Shōgun 2	*Total War: Shogun 2* (The Creative Assembly 2011)
Skyrim	*The Elder Scrolls V: Skyrim* (Bethesda Game Studios 2011)
TBS	turn-based strategy
TT	technology tree (tech tree)
TTRC	technology tree representation component
TTT	Tech Tree Tool
UI	user interface
Witcher 2	*The Witcher 2: Assassins of Kings* (CD Projekt RED 2011)
XP	experience point

List of Included Publications

This thesis is a compendium of six original publications referred in the text with the Roman numerals as indicated in the following list. The listing order is not chronological or alphabetical, but the publications are presented in the natural order considering the discussion to follow.

I T. J. Heinimäki. Technology Trees in Digital Gaming. In *Proceedings of the 16th International Academic MindTrek Conference 2012 (AMT2012)*, pages 27–34. Tampere, Finland, October 2012.

II T. J. Heinimäki and J.-M. Vanhatupa. Implementing Artificial Intelligence: A Generic Approach with Software Support. *Proceedings of the Estonian Academy of Sciences, vol. 62, no 1*, pages 27–38. Estonian Academy Publishers, March 2013.

III T. J. Heinimäki. Considerations on Measuring Technology Tree Features. In *Proceedings of the 4th Computer Science and Electronic Engineering Conference 2012 (CEEC'12)*, pages 145–148 (orig. 152–155). Colchester, UK, September 2012.

IV T. J. Heinimäki and T. Elomaa. Facilitating Technology Forestry: Software Tool Support for Creating Functional Technology Trees. In *Proceedings of the Third International Conference on Innovative Computing Technology (INTECH 2013)*, pages 510–519. London, UK, August 2013.

V T. J. Heinimäki and T. Elomaa. Quality Measures for Improving Technology Trees. *International Journal of Computer Games Technology, vol. 2015, article ID 975371*, 10 pages. Hindawi Publishing Corporation, April 2015.

VI T. J. Heinimäki and T. Elomaa. Augmenting Technology Trees: Automation and Tool Support. In *Proceedings of the 7th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games 2015)*, pages 68–75. Skövde, Sweden, September 2015.

All of the published conference papers (i.e., Publications I, III, IV, and VI) have undergone double-blind peer review processes having at least two referees each. The known acceptance rates (stated in the proceedings) of the conferences corresponding to the conference papers are as follows: for Publication I, 45%, and for Publication IV, 34%.

Concerning Publication V, the editor based his decision on the opinion of at least one external reviewer (unknown to the author). The acceptance rate of the journal is 33%.[i] Publication II was born as an invited journal article based on a previous (refereed) conference paper. The article was reviewed by three new reviewers before the decision of the editor.

The kind permissions of the copyright holders of the original publications for republishing them as parts of this thesis are hereby gratefully acknowledged. Publication-specific copyright information can be found on the corresponding cover pages of the publications.

[i]`http://www.hindawi.com/journals/ijcgt/`, accessed in May 2015.

Author's Contribution to the Included Publications

The author of this dissertation was the main author of all the included publications. The roles and contributions concerning individual articles and conference papers are explained in more detail below.

I This conference paper is a short review-like introduction to technology trees used in digital games. The author of this dissertation is the sole author of the paper.

II This journal article on implementing artificial intelligence is based on a prior paper of the same authors, for which the second author contributed some text and our original script state machine specification. The author of this dissertation implemented the software tools presented and wrote the most of the text. He also extended the original paper into this journal article.

III The paper contains original ideas of possibilities to measure technology tree features. The author of this thesis is the sole author of the paper.

IV This paper introduces an approach to implement technology trees separately from the programs using them and a software tool implemented for this approach. Mostly the paper has been written and the software solely implemented by the author of this dissertation.

V This article is effectively continuation for Publication III and presents new ideas for measuring technology trees, as well as a software implementation. The author of this dissertation has written the main body of the text, implemented the software, and conducted the tests.

VI This conference paper considers technology generation and ways to make technology trees dynamic. Most of the paper has been written, the software has been implemented, and the tests have been conducted by the author of this dissertation.

Chapter 1

Introduction

"It may be that all games are silly. But then, so are humans."
– Robert Lynd [a]

During the past few decades, digital gaming[1] has emerged as a noteworthy phenomenon, both culturally and economically. Playing is an important part of the daily life for many people, and the game industry generates a revenue of tens of billions of dollars annually. Game development has become a serious business, and, therefore, game developers should pay special attention to the quality of their products.

Game development projects can span over several years and employ people by hundreds [3]. However, the human resources involved in game development are limited, so using them efficiently is a matter of paramount importance. Automation and suitable methodology can help in improving

[a] `http://thinkexist.com/quotes/robert_lynd/`, accessed in May 2015.

[1] In this thesis, digital gaming refers to playing digital electronic games interacting with human players via (at least) a view capable of presenting graphics and a hardware interface to make it possible to control the game execution. These games include "computer games" (referring to the games for general-purpose desktop computers or laptops), "console games" (referring to the games for dedicated gaming machines), "arcade games", game applications for mobile devices, etc. "Video game" is another generic term that could be used to cover the same ground, but we prefer "digital game" emphasizing the (digital) computing involved, not the view. Sometimes "video game" is also used synonymously to "console game", so in order to avoid confusion, "digital game" may be a better choice. The author of this thesis has personal experience mostly on computer games, but the issues discussed in this dissertation are, to an extent, relevant for all the current digital gaming platforms.

the quality of games and facilitating the development work. There is much to meliorate, and this thesis aims to present usable ideas to these ends.

This chapter is divided into several sections. In Section 1.1, the motivation for this work is discussed further, after which the scope and the objectives are presented in Section 1.2. The exact research questions are given in Section 1.3, and the research methodology is treated in Section 1.4. Then, the central contributions of the work are stated in Section 1.5, and finally the structure of the rest of the dissertation is introduced in Section 1.6.

1.1 Motivation

Digital games have nowadays a huge impact to our society in a number of ways. In 2013, the global game revenues were over $75 billion, and they seem to still continue their rapid growth [74]. Recent American gamer demographics show, for example, that most of the Americans play digital games, about two of every five players are at least 36 years old, and there are nearly as many female as male players [27].[2] Similarly, games have become an important part of life also elsewhere; for instance, there are over 33 million gamers (concerning digital games) in the UK alone, which is nearly 70% of the population [50]. There also the number of gaming females has actually exceeded that of males [50].

So, games and producing them are definitely not just some marginal phenomenon and an awkward hobby of strange teenager boys to be ignored anymore, but gaming reaches a huge portion of the population. The user base of games is becoming indistinguishable from that of conventional applications [25]. Computing hardware has evolved rapidly, and digital games as products have also changed considerably in a relatively short time. Digital game development is still a young field of industry, as is also software development more generally.

Recent decades have seen a large number of new kinds of methodologies, paradigms, and tools, and game-developing companies and individual

[2]Published statistics may have their problems [92], but the main point here is the fact that huge masses of different people play games, nevertheless.

developers have for sure learned from the games produced so far. However, there are still many problems left to be solved and room for improvements. Game development has become harder as the project sizes and the general complexity of games have increased, and as a software type, games are typically very complex and demanding [10].[3] Therefore, the author has been interested in providing useful ideas and tools to improve the development process. Both big companies and independent developers can benefit by using suitable methods, software tools, and automation. Besides the savings in working time, efforts, and development costs, possible improvements in game quality manifest themselves as happy and loyal customers, thus creating continuity and serving the common good.

1.2 Scope and Objectives

The leading idea of this thesis is that games should be of high quality, but developing them ought not to be too tedious and need too much human work. This thesis is about finding means to facilitate producing better games more efficiently.

Developing better and faster graphics hardware and drivers was the main approach in improving the user experience in digital games for years, and as a result, current computer graphics can be extremely impressive. Nowadays, many people agree that improving other areas such as the game artificial intelligence (AI) has become a more important approach in developing better games; even magnificent graphics cannot save a game, in which the basic idea is poor, content is dull, or the gaming experience is frustrating due to route-finding bugs, dumb dialogues, or other AI-related handicaps.

Several aspects of AI and related fields – for instance, pathfinding – used in games have been studied rigorously. However, there are still many gaps to fill and much work to do. The complexity of modern digital games makes

[3]Recently, games developed for mobile platforms have become an important subtype of digital games. They naturally tend to be simpler and smaller in size than computer and console games. However, the contributions of this thesis may be of help also in mobile game development.

creating AI for game agents a challenging task [71]. Conventional AI algorithms cannot often be used – at least not without major modifications: a large portion of algorithms published by academic researchers so far has consisted of methods that are unsuitable to be applied in digital games as such due to the special requirements involved [10].

This thesis cannot tackle all the common problems present in contemporary games and their development processes. Therefore, the dissertation focuses only on one mechanism, which the author finds interesting among all the partial systems forming digital games: development choices and offering them to players during the gameplay.

Typically, some kind of a development graph structure is used for structuring the possibilities to advance in a game. The author calls those structures *technology trees* (TTs). Creating such structures determining virtual development possibilities and managing the effects of advancement is not trivial – at least, when one strives towards high quality – and requires a good deal of human labor. Therefore, the main theme of this work is improving TTs and finding better ways to implement them economically without compromising the quality.

Moreover, just implementing good TTs effectively is not enough, because typically they should be usable by AI agents in addition to human players. Therefore, some generic game AI discussion is also included. As far as prior publications concerning these issues are concerned, capability to somehow use a predetermined TT (e.g., by constructing buildings) has been included in several academic game AI implementations (see, e.g., reference [81]). However, the foci of these studies have been elsewhere, so the scope of this dissertation is unique.

As the context of the discussion the author mainly uses dedicated single-player computer game applications that are run locally. However, some of the ideas can be applied also to the currently trendy world of browser games, cloud gaming, and other network-based gaming solutions.

Personally, the author finds single-player digital games more interesting than multiplayer ones, but the popularity of the latter must be taken into account, so there will also be some comments concerning them. Gamers "are driven to networked games because of the failings of the computer characters" [57]. Therefore, the author thinks that single-player games deserve special attention, especially as far as game AI is concerned. However, for the main topic of the thesis, TTs, the number of players as such is a matter of relatively low importance – at least considering this particular study. Nevertheless, when implementing TTs, one must, of course, take the cardinality of the users into account. Some of the modern game applications and architectures behind them must be able to serve a very large number of users [85].

Besides desktop computer environments, implementations generated with tools to be presented in this thesis have been partially tested also in mobile devices such as the Jolla phone. Computer games[4] have been used as real-world digital game examples.

1.3 Research Questions

Narrowing down the higher-level objectives and research interests has led into a set of specific research questions. These are the essential questions, to which the included publications and this thesis as a whole aim to answer:

Research Question 1 (RQ1):
How could the current practices of digital game development (especially from the viewpoint of implementing TTs) be improved?

Research Question 2 (RQ2):
What kind of imperfections and problems are typical with TTs?

Research Question 3 (RQ3):
Can automation help with producing better TTs?

[4]The purpose of this thesis is not to specifically promote any single games. Representative examples have been picked among the games that were conveniently available when writing this thesis, but there are a large number of games that could have been used instead or in addition. The selected ones, however, are able to fully demonstrate all the essential facts intended to be demonstrated.

Research Question 4 (RQ4):
Can automation help with reducing human workload in TT construction?

Naturally, the possible approaches and methods that are found in connection to answering the questions may be matters of interest also when considering the other questions than RQ1. If RQ3 or RQ4 is answered in the affirmative, the corresponding "how" question is a natural continuation for the research question. The downsides and practicality of the possible approaches are, of course, also important and deserve attention.

1.4 View to the Research Work and Applied Methodology

When categorizing and describing the nature of the efforts taken for this dissertation, the author considers *design science research (in information systems)* as the main (high-level) research paradigm applied. "The design-science paradigm seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts" [46]. Basically all the work done has, indeed, been about creating artifacts in order to extend the boundaries of the practices and tools used in digital game development. The approach is conventional – constructing artifacts has been a typical means of proceeding in computer science and software engineering research [49]. The work has been a cyclic alternation between building and evaluating. These two activities can be seen as the basic activities of design science [65].

Because of the heavy emphasis on creating artifacts, methodologically the research work is easy to classify as constructive. The used methods have included both conceptual and technical development – both of the types of constructive research methods [48]. During the process, a wide array of artifacts has been built. For example, terms have been defined, models and algorithms have been developed, and prototype software tools have been implemented. This means also that artifacts of all the four design science product types (i.e., constructs, models, methods, and implementations/instantiations) [65] have been produced. The artifact construction has been based on acquired knowledge on recent and contemporary digital game development and hypotheses on improving the process.

Construction, however, is not enough. It has been realized that constructs, models, and methods working in theory may not be viable in practice [65]. In order to acquire evidence of the (non)viability of the proposed approaches and tools, empirical tests have been carried out. Therefore, considering Järvinen's taxonomy of research approaches [51], the work conducted for this dissertation could fit partly in the category "artifacts-evaluating approaches", and, additionally, partly in "artifacts-building approaches".

Carrying out the research project has manifested certain features of the action design research (ADR) [93] approach. Most notably, evaluation has been present all the time alongside building the artifacts, though a specific evaluation step has often been taken after getting an artifact ensemble "ready". The ADR process has four stages: "problem formulation" (Stage 1), "building, intervention, and evaluation" (Stage 2), "reflection and learning" (Stage 3), and "formalization of learning" (Stage 4) [93]. Problem formulation has been done before actually moving on to the artifact creation phase, during which also evaluation has been constantly performed. However, because all the work has been carried out by a single person, proper "organizational intervention" (see [93]) has been impossible. Trying to learn generally – not only concerning specific instances – along the way has been an important part of the process. This corresponds to Stage 3 in the ADR methodology, and suggesting design principles and refinements to them could be seen as Stage 4 operations.

The artifacts (e.g., concepts, software, frameworks, and methods) that were created during the research process and are essential concerning this dissertation are documented in the included publications (see Sections 1.5 and 1.6). The experiments that have been performed with these artifacts are also described in them.

1.5 Contributions of the Thesis

Six peer-reviewed publications are republished as parts of this thesis to serve as a basis, on which to build. Their central ideas and results are put together, filling gaps, in order to present a coherent view of the considered problems and possible ways to tackle them.

The main contribution of this thesis is the presentation of tools and methods designed to facilitate implementing high-quality TTs and using them as a part of digital games. In more detail, the dissertation

- introduces software tools and methods for designing TTs and generating corresponding functional script code automatically in order to reduce workload of game developers,
- discusses and introduces tool support for measuring TT properties for adjustments in order to achieve better quality,
- tries to convince and motivate the academic community to conduct further research on TTs and implementing them in order to have better games in the future, and
- provides answers for the research questions.

Due to the nature of the research process and this thesis, mostly the evaluation presented consists of hardly more than initial impressions. It is difficult to give numerical results, when dealing with subjective topics, such as the quality of a TT. The same applies to, for instance, savings in working time or efforts needed using certain tools or methods, because such matters depend on the workers involved and the desired result of the work, among other things. However, based on the presented ideas and results it should be easy to carry on and conduct further studies thus advancing the collective knowledge and understanding of the field, hopefully leading to better games.

1.6 The Outline of the Thesis

The dissertation consists of an introductory part and six publications. The remainder of the introductory part is organized as follows. Chapter 2 presents some background in order to establish a groundwork for the rest of the thesis. The main themes are in-game (simulated, virtual, game-changing) development, TTs, and their typical use and roles in digital games. Also, genres of digital games are discussed. Chapter 3 continues on the topics of game development, people involved in it, scripting, and AI. Moreover, related work is covered.

In Chapter 4, the insights and visions of the author concerning creating digital games and especially TTs are explained; the chapter presents the approach proposed to improve game development and solve the research problems. Also, the content of the included publications is briefly explained in order to clarify their roles as parts of the dissertation.

The results, achievements, feasibility, and problems related to the methods applied and the study as a whole are discussed in Chapter 5. After that, the introductory part of the thesis is concluded in Chapter 6, which includes answers for the research questions. Also, some future directions and possibilities are covered.

After the introductory part, the six included publications can be found at the end of the thesis. The main ideas and content of them are explained in Section 4.6, and the publications are referred to wherever necessary.

Chapter 2

Games and In-Game Development

"If Pac-Man had affected us as kids, we'd all be running around in dark rooms, munching pills and listening to repetitive electronic music."
– Marcus Brigstocke [b]

It cannot be omitted that "videogames are a time-based medium" [38]. The same holds true for any game of serial nature, considerable duration and several different game states – i.e., basically all interesting and nontrivial games. For a game to really appear interesting, it has to be able to keep its players interested for a continued period of time. There is a common, fundamental mechanism aiming to achieve this: including upgrades and development options in the game.

[b] `http://izquotes.com/quote/213270`, accessed in May 2015.

For example, a *strategy game* (see Section 2.1.2) should feature several unit types to keep the number of battle combinations large [1],[5] but this is not enough. All the types should not be available from the beginning, so that the weaker unit types would also get to be used [1]. The possibility to develop better units during the game also adds to the complexity and interestingness of the game.

The continuum of the in-game development structures that had been used in the board gaming domain was reshaped, when TTs emerged in digital games and evolved to take advantage of the possibilities offered by computerization. TTs are graph-based structures offering upgrades and options for development. Game mechanics based on these elements are used heavily, because they can enrich games in various ways. For instance, TTs can keep a game instance interesting by offering choices and defining development effects. TTs also enlarge the replayability value of a game by providing different paths to victory to be experimented with.

Traditionally, TT is a term associated with strategy games, but the idea of such development possibility–defining and selection-guiding graphs has been adopted also into other genres, like computer role-playing games, sports games, management (god) games, action games, and different genre mixtures. Creating portable TTs of high quality with ease and without genre restrictions is the focal point of this thesis. However, genres are discussed a bit further in Section 2.1. Then, development generally as an in-game phenomenon is considered in Section 2.2. Thereafter, Section 2.3 zeroes in on TTs – the most essential structures and mechanisms of development concerning this thesis.

[5]In other words, several unit types are needed for strategic and tactical use of combined arms, which is essential in many strategy games. So-called *Condorcet cycles* concerning the types are important; generally there should not be a single unit type dominating all the others, and this can be achieved by intransitivity of strengths or weaknesses against other unit types [70]. This kind of a “rock-paper-scissors” setting is a classic way to avoid dominant strategies [1]. The mechanism can lead to interesting games by itself, but more depth is achieved by having also a possibility to gradually develop better units within unit type categories as a part of the gameplay.

2.1 Genres of Digital Games

Digital (and other) games are diverse and can be classified and categorized in various ways into distinct genres. Several taxonomies have been created (e.g., reference [43]), and there is some variation between different genre definitions found in the literature. Nevertheless, typical features characterizing common genres can be pointed out. This section briefly describes some of the most essential digital game genres concerning this thesis in order to give the necessary background for non-gamers.

These genres are not the only ones, within which the ideas presented in this thesis can be used, but they are the most obvious. It is also worth mentioning that many contemporary games cannot be classified purely into any of the conventional genres, but they are actually mixtures of several of them. Moreover, genres change over time, and new ones keep emerging continuously due to, for example, the evolution of gaming media and platforms.

2.1.1 Computer Role-Playing Games

In role-playing games, "players create or take on a character represented by various statistics, which may even include a developed persona", and "this term should not be used for games [. . .] in which identity is not emphasized or important, nor where characters are not represented statistically" [114]. *Computer role-playing games (CRPGs)* follow the same basic idea of having characters – typically, at least, a player character (PC) – to be distinguished with their attributes, skills, or other (typically numerical) statistics.

These property values allow and enable character development, which constitutes an essential part of these games; getting opportunities for a PC to develop and grow continuously during a game is considered a matter of paramount importance within this genre [44]. *Experience points* (XPs) are used to track experience, and accumulating them leads to improvements of skills or character levels [44].

In some CRPGs, the PC is initially formed using a character generation process, which fixes various properties, such as the appearance, name, sex, specializations, and so on [44]. The generation may be randomized, or the player can be given the freedom to allocate, for instance, some initial "points" available to be allocated into different statistics [44]. Also *character classes* (e.g., a wizard, a warrior, a rogue, or a Jedi consular) are often used in specializing a character. A class, on the one hand, often limits abilities that can be developed or used, but on the other hand, also opens up special possibilities of development and abilities that are characteristic to the class.

There are a large number of different digital games considered to be CRPGs and also games officially or commonly classified differently with features typical to CRPGs. As far as this thesis is concerned, the development of characters (or creatures or objects representing players and possibly other "player-like" entities) in terms of abilities, skills, and other statistics is an important aspect of this genre.

2.1.2 Strategy Games

Strategy games let players control entities consisting of several individuals (or other units). Typically, new units are created during a game, and in order to do so, resources have to be collected and used [43]. There is much variety within the genre, but the common factor of strategy games is – strategy. Sometimes it might be better described just as tactics, but there are also strategy games, in which more comprehensive overall strategies are important.

Decision-making has a central role in these games; players have to keep continuously deciding, how to use their units, how to improve and develop their empires, and how to overcome different obstacles – set by the game or other players – along the path leading to the victory. Conflict and direct acting against opponents are typical features distinguishing strategy games from some of the other genres such as management and construction simulations and puzzle games [1]. In addition to the aspect of physical combat, there can be others, such as espionage and diplomacy [1]. Certain symmetry is typical: different parties competing with each other in a strategy game play somewhat similarly (which is not true for, e.g., action games featuring a PC struggling against the rest of the world) [1].

Strategy games cover a variety of quite different titles. Therefore, in order to be more specific, representatives of the genre can be classified further. Strategy games are often divided into two main subgenres. These are discussed next separately, because there are certain fundamental differences – concerning both gameplay and this thesis.

Real-Time Strategy Games

Games of the “strategy” genre have been described as ”games emphasizing the use of strategy as opposed to fast action or the use of quick reflexes, which are usually not necessary for success in these games” [114]. However, *real-time strategy (RTS) games* are sometimes seen merely as “button mashing”. As the name suggests, in RTS games events take place and the situation changes in “real time”. The players do not have to wait while others are completing their turns, but all the players are free to take actions at their own pace, as their situations allow.[6] Therefore, commands should be given rapidly in order to be able to compete successfully with AI players having the advantage of being able to command their forces via a programmatic interface without any need to use clumsy pointing devices, controllers, or keyboards.

In the ideal case, however, it should be more important to be able to make good strategic or tactical decisions in reasonable time than to hit buttons or point and click with a pointing device very quickly. Typically, the focus of RTS games is on military combat [82], and constant time pressure and the need for making quick decisions as the situation develops suit well into this kind of a setting. Especially this is true in tactics-oriented games trying to mirror real-world tactical decision-making challenges.

Turn-Based Strategy Games

Classical *turn-based strategy (TBS) games* do not press the players to make their decisions as quickly as RTS games. In a TBS game, a player makes decisions and gives commands regarding one game turn at a time, typically

[6]The actual implementation may handle the game as a series of short “microturns”, but from the human perspective, there are no observable distinct turns for different players; they are allowed to play “simultaneously”.

being able to use as much time as needed before passing the turn to the next player (which may be a human or an AI).

The time-related pressure in TBS games differs from that of RTS games; it emerges from the fact that the turns should be spent as effectively as possible in order to gain the upper hand over the opponents. The number of turns can also be limited. Because there is more time to ponder on different options than in RTS games, TBS games are also able to offer more possibilities for (micro)management and making choices. Many players appreciate the depth that can be offered this way.

On the other hand, sequential play and a variety of options combined with unlimited time to be used for playing a turn also easily lead to increased downtime. The situation called analysis paralysis [9] occurs when one is overwhelmed by the large number of alternative ways to proceed and cannot decide the next action to take. Such a condition is avoided in real-time games [61], but it can be a problem in a TBS setting. The duration of a TBS game can be considerable. To mitigate this and to keep the game pace tolerable, it is possible to limit the allowed time for taking a turn. This is especially useful in multiplayer game settings with several human beings involved: unlike computers, humans tend to get irritated easily.

2.1.3 Other Relevant Game Types

In addition to CRPGs and strategy (RTS and TBS) games, different taxonomies introduce various genre labels and game types that may be relevant, when seeking application areas for the approach of this thesis. For instance, so-called *4X (eXplore, eXpand, eXploit, and eXterminate) games* can be treated as a subgenre of strategy games – as is done in this thesis – but 4X can also be seen as a genre distinct from strategy games [43].

Management games (or *"god" games*) resemble strategy games: there is a population in the game world, and the player can affect the population [43]. Also *resources* (see Section 2.3.2) are involved in these games [43]. Resources, and possibly converting them into different forms (and sometimes also destroying items in order to acquire resources for reuse) – thus affecting the game world or the PC abilities – play a central role in building-oriented games (including strategy games).

Development aspects can be present also in, for example, sports games, in which one might train players or teams in different skills or improve vehicles by adding and changing parts in them. Often, action games also feature development elements. For instance, in a space shooter, the player may be able to fly around hunting for weapon upgrades, or a first-person shooter game may include CRPG elements and let the skills and weaponry of the protagonist be improved along the way.

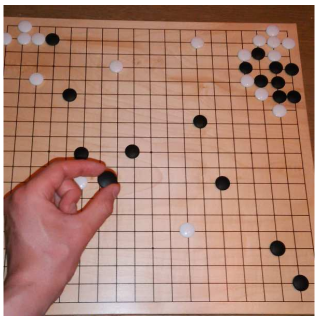
2.2 Development as a Part of Gameplay

Life consists of changes of, for example, situations, physical properties, capabilities, knowledge, opinions, and relations – life can even be seen as nothing but a constant change. For humans it is natural to strive towards better life and attaining objectives that seem beneficial. Developing oneself and human relations, advancing in social hierarchies, and amassing wealth and resources are hardly strange ideas. Also, entities composed of humans, like nations, federations, and organizations, have their objectives and ways to proceed towards their respective goals.

Objectives, as well as changing conditions and properties, are typical elements also in games; in a game, one aims at winning within the rules, and there are different game states involved, through which the game proceeds. Non-trivial games are largely won by successful *development*.

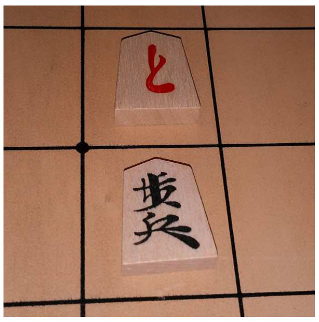
2.2.1 Non-Digital Games

Development in games takes different forms and happens on different levels. For example, in *go*, the game proceeds, when the players build the "game world" and the overall situation on the board stone by stone (see Figure 2.1a). Being able to develop the situation on the board to eventually favor oneself more than the opponent is the key to victory.


(a) In *go*, the game situation develops stone by stone.

(b) In *Magic: The Gathering*, properties are tracked using, e.g., counters, dice, and card orientations.

(c) In *Clash of Cultures*, plastic cubes and cardboard tokens are used to track advances and resources.


(d) In *shōgi*, most of the pieces can be promoted by flipping them over to show the promoted side.

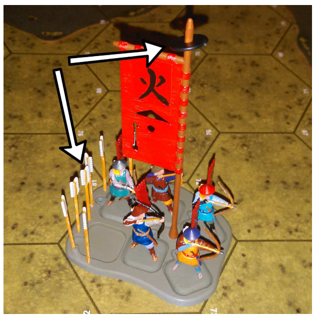

(e) In *Samurai Battles*, there are many ways to track unit statistics.

Figure 2.1: Development and keeping track of it in traditional games.

Many card-based games involve constructing tableaux of cards giving benefits for the rest of the game or points needed for the victory. Sometimes the properties of individual cards can also change and be tracked by, for instance, adding or removing different counters on them or attaching modifying cards to them. Also, card orientations can be used for tracking purposes. These means are used, for example, in *Magic: The Gathering* (Wizards of the Coast 1993). Example tableaux are illustrated in Figure 2.1b.

There are board games, like *Clash of Cultures* (Z-Man Games 2012), in which advancement possibilities and their prerequisites (i.e., TTs) have been defined in the form of cardboard templates, and the advancements of the players are tracked with markers added on them, as illustrated in Figure 2.1c.

Some traditional board games – with boards and pieces – have been featuring possibilities for individual piece promotions for at least several hundreds – possibly even thousands – of years. This kind of unit upgrading adds interesting elements to tactical games. Often the promotion is from one piece type into another, beyond which one cannot proceed, because of the practical limitations of board gaming. For instance in shōgi, most of the pieces have two usable sides: the basic version and the promoted version of the piece, and pieces are flipped over on the board to reveal the promoted side, when appropriate. In Figure 2.1d, two similar fuhyō (foot soldier, "pawn") pieces are shown. The upper one has been flipped to show the promoted side (tokin).

To follow the development and properties of individual game pieces, it is also possible to let them remain as they are, but attach (or remove) markers to them or, for instance, to make notes to associated status cards. For example, in *Samurai Battles* (Zvezda 2012) (see Figure 2.1e), an *honor token* can be added to a unit flag pole as an indicator of the better-than-normal statistics of the unit, and arrow models can be used to keep track of the arrow (or other projectile or ammunition) situation of a unit. Models can be removed from and added to unit bases, and besides, there is a unit card (not shown in the figure) for each unit (consisting of a unit base and the models on it) to, for example, record statistics.

2.2.2 Digital Games

The roots of many contemporary digital games run deep in board gaming.[7] Board gaming is a very old invention [47], and it is impossible to say, when and where exactly upgrading or acquiring benefit-giving entities within given

[7]Board gaming in this thesis refers to a wide array of tabletop gaming – or, basically, any gaming that is not physically overly demanding and is performed by human players in a local, constricted space. The board is not always necessary. Pen-and-paper role-playing games are typically seen as a totally different gaming genre, but for our ends, they can be handled as a subset of board gaming as well.

development rules during a game occurred for the first time. However, it can be said that computerization has opened up new possibilities.

Traditionally, the development processes in games have been limited by the human capabilities of the players, bookkeeping resources, and presentational difficulties, even though – as demonstrated in Section 2.2.1 – there are different ways to track development. Computers have alleviated the difficulties considerably, and, therefore, one can find a large number of development options in digital games. A computer can be used to keep track of, for instance, career histories of simulated characters, modify their current properties accordingly, and illustrate the essential status facts for the players conveniently. Delegation of bookkeeping, calculations, and presentational matters to a computer frees the human players to enjoy the development without tedious work that a computer is more suited to handle.

2.3 Technology Trees

Section 2.2 discussed development in games generally. This section continues by introducing and giving background on an especially important structure used to enable in-game development in a variety of modern digital games: a TT.

In role-playing games in general – and thus also in CRPGs – perceptible character advancement is essential [44]. Development of a character is easily tracked using numerical statistic values for attributes and skills [44]. Besides straightforward basic skills to develop, a character may be able to acquire totally new abilities during the game, and the corresponding player may be able to steer the development and choose which perks are learnt. This is an element typical also to strategy games, although in them the effects of development might be quite different and target more than a single character.

Often the possibilities for development depend on the current status.[8] This status comprises the aspect of the acquired development, possible wealth

[8]If development was a stochastic process, the Markov property would often hold. However, typically the point is to let the players control the development, and normally the process is deterministic, though some randomness may be involved.

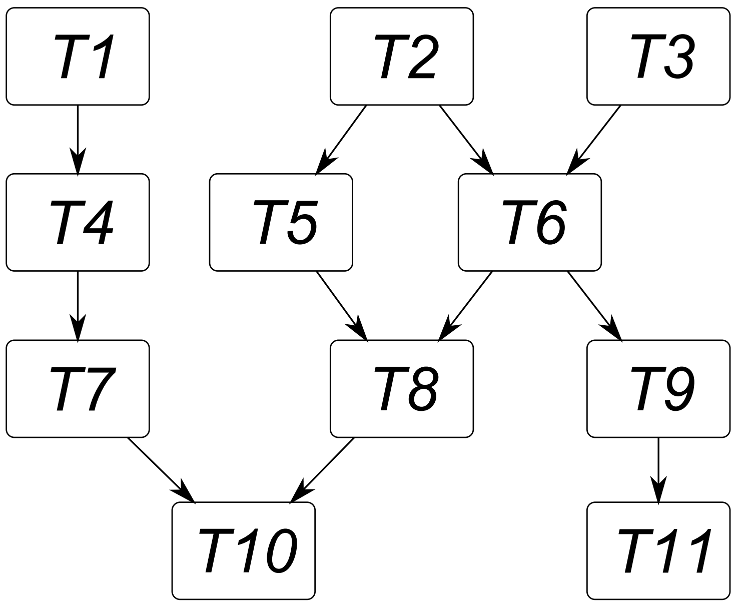


Figure 2.2: The basic idea of a TT.

in forms of different resources related to development options, and possible other limitations. It has been found convenient to model and represent such development possibilities using augmented graph structures that define possible prerequisite relationships of these options and contain also other necessary data.

TT is a common and traditional term to refer to such structures in strategy games. When searching the origins of such conventional TTs, typically the board game *Civilization* by Francis Tresham (Hartland Trefoil 1980) is given the credit of introducing them. *Sid Meier's Civilization* (MicroProse 1991) (*Civ*), an iconic computer strategy game, is famous, among other things, for using them. Since *Civ*, TTs have been used in various digital games.

Although creating a good TT is generally a demanding task, and the details vary, the basic idea of TTs and using them in digital games is simple. Consider the graph illustrated in Figure 2.2. Let the nodes *T1*–*T11* be benefit-providing entities and let the player be able to proceed in the structure by "developing" these nodes one by one in order to obtain their respective benefits. Let the edges of the graph control the availability of the nodes in such a way that initially, only *T1*, *T2*, and *T3* are available. Developing, for instance, *T2* unlocks *T5* and *T6* so that they can be developed as well. Depending on the TT semantics, developing *T5* or *T6* alone may suffice to unlock *T8*, or both of them may be required. Advancing in a TT requires typically time and (other) game resources. Therefore, the player is forced to prioritize and encouraged to plan ahead, when making decisions on what to develop and when.

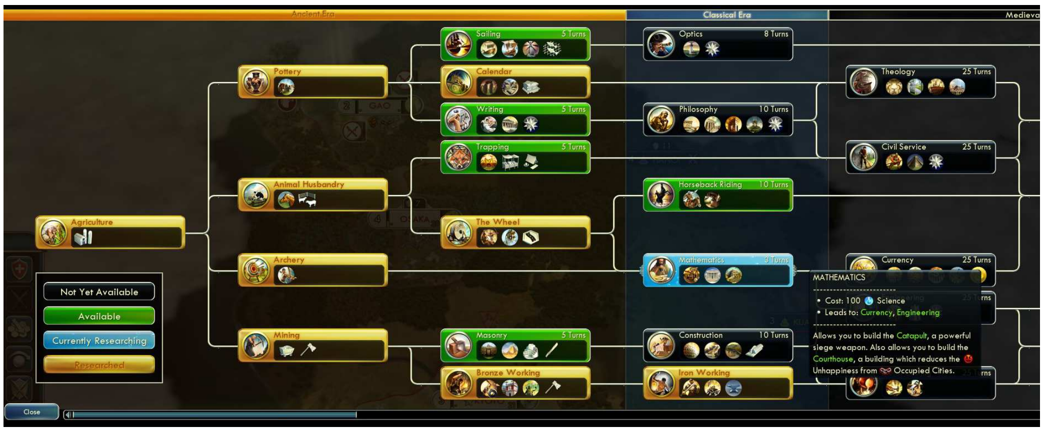
As stated in Publication V, TTs "are typically seen either as (1) structures (mechanisms) for defining and controlling development (upgrading), based on dependency relations of technologies, or (2) flow-chart-like presentations on these dependencies". This thesis focuses on the former interpretation; TTs should generally be understood as abstract models, which may or may not have visualizations. An example of an in-game visualization of a TT can be seen in Figure 2.3.

In order to further clarify and crystallize the intended interpretation of TTs concerning this thesis, let us state that a TT should generally be understood as an abstract structure that

- can be modeled as a (multi)digraph with associated additional data and
- can be implemented programmatically so that this implementation can be used to define or model, within a game, development options (corresponding to the nodes in the underlying graph) and their mutual dependencies (corresponding to the edges in the graph).

Similar rather generic TT interpretations can be found in the game literature. A TT has been described, for example, as "a structure that controls progress from one technology to a better technology, enabling the player to create better facilities or more powerful units" [70].

However, mostly the term TT tends to be used only in relation to strategy games. For other genres, different names (e.g., "talent tree" or "skill tree") are used. This is where the interpretation used in this thesis distinguishes itself from others; it is very generic and nonlimiting, allowing all these depelopment-guiding structures to be discussed as TTs. There is no



MATHEMATICS

- Cost: 100 Science
- Leads to: Currency, Engineering

Allows you to build the Catapult, a powerful siege weapon. Also allows you to build the Courthouse, a building which reduces the Unhappiness from Occupied Cities.

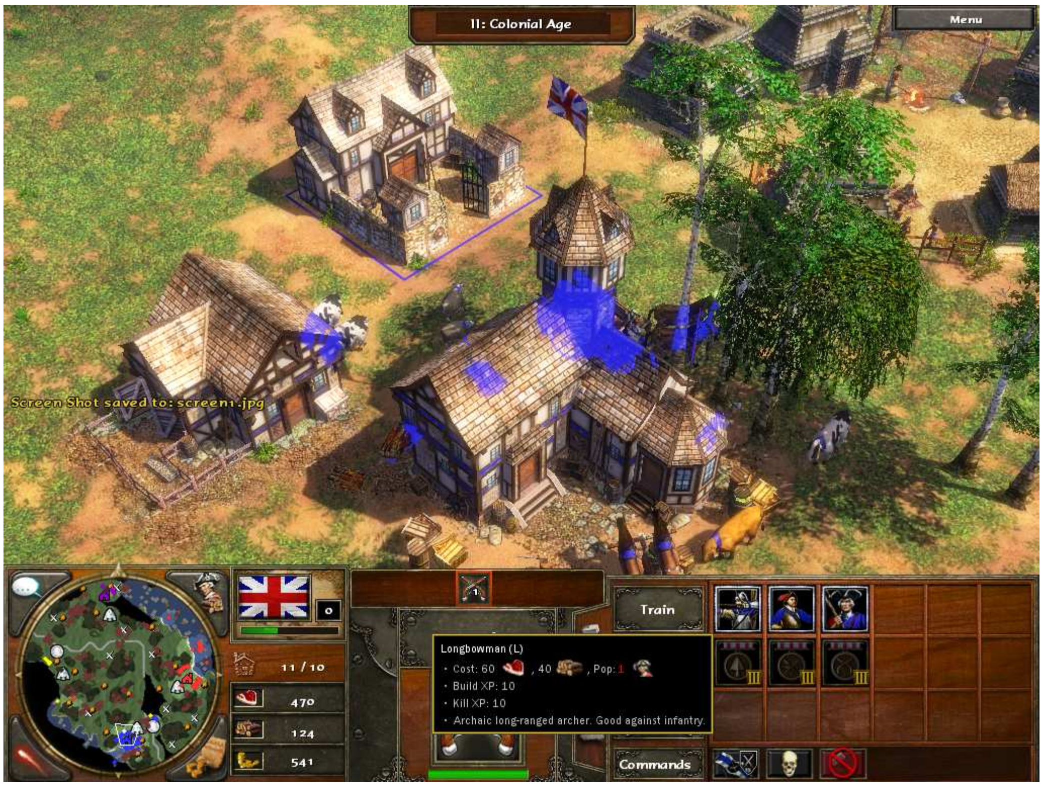
Figure 2.3: A partial TT visualized within a game. A screen capture from *Sid Meier's Civilization V* (Firaxis Games 2010) (*Civ 5*). This kind of graphical representations of development possibilities (prerequisite graphs) are often offered to players for making decisions about the current and future development focus areas.

need to use different words only because there is some variation concerning the objects of the development and the nature of the elements offered to be developed – which the author invariably calls *technologies*[9] for the rest of the introductory part of this thesis.

Technologies – the nodes in TT graphs and goals for development efforts – come in various forms. They can represent technologies in the normal sense of the word, but they can also be, for example, new building types (or abilities to build such), new political systems, or new spells to cast. The benefits given by technologies are also diverse. One distinction could be made between passive abilities – affecting, for instance, by giving a permanent income boost of a certain percentage – and abilities to do something new and requiring some kind of an activation to make the technology count, like learning a new spell or obtaining the ability to train samurai cavalry units.

TTs are simple structures by nature, but they can add to the interestingness of a game considerably, if used correctly. They tend to combine two "basic attitudes governing play" of Caillois [16], namely *agôn* and *mimicry*.

[9]It is also common to use the abbreviated version *tech* for TT-related technologies, and correspondingly refer to TTs as *tech trees*.



Screen Shot saved to: screen1.jpg

Longbowman (L)
• Cost: 60 , 40 , Pop: 1
• Build XP: 10
• Kill XP: 10
• Archaic long-ranged archer. Good against infantry.

Figure 2.4: In *Age of Empires III* (Ensemble Studios 2005), barracks offer, e.g., access to training of several military unit types, and buildings of other types offer their respective benefits and development possibilities. A screen capture from the game.

TTs are about calculating and using the possibilities offered by the game in a clever and efficient way in order to win (agôn), but on the other hand, they can help in customizing, for instance, the PC in order to play a role as wanted (mimicry). It is also possible to add the attitude *alea* by introducing randomness to a TT.[10]

For a more thorough introduction to the essence of TTs, see Publication I. It offers a general overview on the subject of TTs and their roles in digital games. In the paper, aspects for classifying TTs are pondered, and examples of real games are presented. In Publication IV, TT functions are explained and TT components are characterized in a generic way.

[10]Caillois considered alea and mimicry as a "forbidden relationship", but here mimicry is seen more widely – including role-playing-like, not totally free, simulation.

2.3.1 Technology Tree Types and Related Mechanisms

The publications should be consulted concerning the basic nature of TTs as prerequisite-defining graphs (with extra information). The same is true concerning the variety of TTs and classifying them. However, this section present a focused view to these topics.

In addition, a mechanism of acquiring benefits based on game-world objects – common in action games and CRPGs – is discussed. This mechanism is worth mentioning, because it is commonly used and related to TTs. The two approaches can even both be used in a single game.

Technology Trees in Different Games

Conventional TTs in digital games emerged as a part of the strategy gaming genre in the early 1990s, and the TT of *Civ* is an example that cannot be omitted – for many people *Civ* is probably the first mental association with the term. The game created a model that was applied and further modified by countless games since.

Civ is a TBS game, and *Civ*-like TTs – like the one of *Civ 5*, partially seen in Figure 2.3 – are typical to this genre. A characteristic property for such TTs is being composed of several (tens or hundreds of) heterogeneous technologies. Several (more than one) prerequisite dependencies for developing a technology are typical, although there may also be “linear”, chain-like portions, in which a technology is the sole prerequisite for another one, which is the only prerequisite to yet another, and so on.

In RTS games, there may not be TT structure visualizations shown to the players for development selections, but often TTs are effectively present nevertheless. It is common that development happens in or in relation to buildings. Constructing a building of a certain type may open up possibilities to train certain kinds of units (see Figure 2.4) or build other kinds of buildings. In addition to opening up new options for development, buildings may bring along their individual in-game effects and offer, besides building and unit types, also other kinds of technologies to be developed.

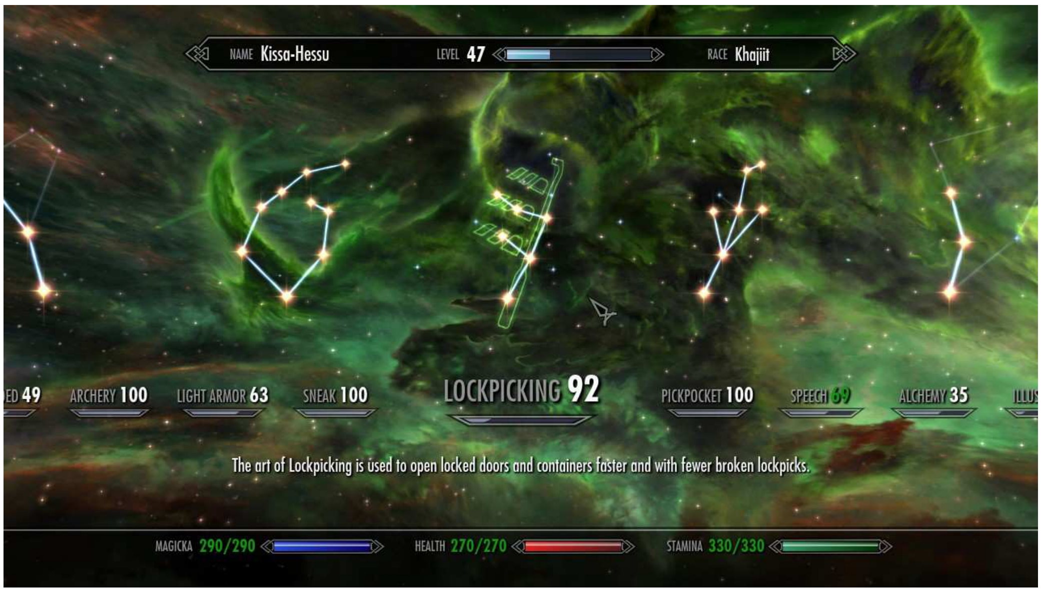


Figure 2.5: In *Skyrim*, the skills available to the PC have been divided into several small TTs ("skill trees"). A screen capture from the game.

The applicability of the idea of modeling development options and prerequisite relationships with graphs is by no means restricted to strategy games only. TTs can be used to offer options, for instance, for character advancement in CRPGs. Eric Schaefer describes the "character skill tree" implemented in *Diablo II* (Blizzard North 2000) as "the most revolutionary new idea" of the developers [90]. In the 21st century, such TTs have been rather common in CRPGs and games featuring role-playing elements. These TTs are usually rather modest in size. However, there can be several of them within a single game. For instance, *The Elder Scrolls V: Skyrim* (Bethesda Game Studios 2011) (*Skyrim*) features separate "trees" for different skill groups. A few of these are visualized in Figure 2.5.

Similarly to CRPGs, the TT idea can be straightforwardly applied to action games featuring role-playing elements as well as to games of other genres involving individual or collective in-game development of virtual entities. This is useful especially, when there are prerequisite dependencies between the technologies. For instance, in a racing game, the player might be able to obtain new vehicle designs based on the existing ones and the parts and partial design blueprints he or she acquires.

Collectible Objects and Their Relation to Technology Trees

Especially in CRPGs, also additional means of offering chances to power up a character are sometimes used. For instance, a PC may be able to obtain (e.g., buy, find, or loot) and equip items in the game world, and these items may alter PC statistics or give abilities. Despite the fact that such *collectible objects* (COs) are – at least apparently – outside the TTs, the available COs or their properties can, for instance, be determined by the development status of the PC. It is also possible to have COs "leveling up" with the PC, or let players upgrade, enhance, or modify COs. For such purposes, COs may use their own TTs.

The main reason to try to acquire or develop technologies in a game is to gain benefits, and thus make it easier to survive and eventually win the game. In addition, developed technologies may serve as "collectibles", something to boast about. In this sense, COs obtained are similar entities; by obtaining them one can typically get benefits, and they can be collected. One main difference between them and technologies of TTs is the typical lack of prerequisites, when COs are concerned.[11]

In addition to the prerequisite relationships, there may also be other restrictions to developing technologies in a TT. Similarly, possessing or using COs can be restricted in various ways. A typical restriction is the limited capacity of possessing objects simultaneously. Carrying each CO often requires space in a corresponding inventory, and COs may also have associated weight attributes (or such) affecting the capability to carry them. These features are uncharacteristic for technologies.

Often COs can also be easily discarded, and sometimes they may be usable only once. Developed technologies, on the other hand, are typically

[11]However, technology sets can be treated as TTs even without any prerequisite relationships. This may not be typical, but, for instance, the tool of Publication IV does not require defining prerequisites, and typically TTs are not graph-theoretically proper trees anyway.

quite permanent – at least in TBS games. In RTS games featuring building-based technology development, the loss of the last of the buildings of a certain type is often seen as losing a "developed" status of a technology corresponding to the building type.

Rewarding players continuously during a game is important in order to keep them interested [44]. It is natural to use COs as such rewards for completing quests such as clearing a dungeon, but similarly obtaining new technologies can be rewarding, and letting players make meaningful selections is also a way to nourish their interest. Such strategic selections are typical to TTs.

Technology Trees as Prerequisite Graphs

Let us also consider the aspect of TTs as prerequisite-defining graphs a bit further; the prerequisite relations between technologies (the edges of the TT graph structures connecting technology nodes) are, after all, basically the reason of discussing TTs and not merely sets of technologies. Also, concerning this aspect, TTs differ from each other. However, the definitive idea is that technologies may act as prerequisites for other technologies, and these relationships are naturally represented or modeled as edges in the corresponding TT graphs.

Let us consider distinct technologies A and B that are prerequisite technologies for yet another technology C. Let the development statuses of A and B be binary: either a technology is "developed" or not. (The author calls such technologies Boolean ones, because it is possible to store the essential development information with Boolean variables.) As far as the semantics of the relationship edges are considered, there are two base cases: in the case of *conjunctive* prerequisite relationships, both A and B are required to have "developed" statuses in order to develop C, but in the case of *disjunctive* relationships, having either A or B developed is enough. Sometimes TTs mix these two relationship types, using conjunctive prerequisites for some technologies and disjunctive prerequisites for others.

This kind of a simple description of the basic idea and semantics of a TT applies to many real TTs quite well, but in Publication IV a more general characterization and related notation are presented in order to be able to cover also more eccentric cases. For instance, instead of using binary development status values, technologies may have several *levels of development* (LODs) for a player to achieve[12], and development-allowing or development-denying conditions for edges may be something more complicated than the development statuses of single technologies. Publication IV can be consulted for more details.

The Witcher 2: Assassins of Kings (CD Projekt RED 2011) (*Witcher 2*) is an example of a game, in which some technologies ("abilities" that can be purchased using "talents") have more than one LODs. There are technologies that can be further modified by mutation, and sufficient development in one of the four "separate" TTs is required before the player can invest into the other three trees. The *Witcher 2* TT structures are illustrated in the screen capture presented as Figure 2.6.

Categorizing Technology Trees

Because of the diversity of TTs, discussing them generally is difficult. This difficulty can be alleviated by a categorization of TTs. It enables discussing subsets of TTs with common properties, and, therefore, one can express oneself more clearly. In Publication I, several classification criteria are presented.

Besides structural issues, rigidity, technology types, resources, and prerequisite relationship types (mentioned in Publication I), TTs can be categorized based on the entities that are affected by the developing technologies. In CRPGs, the affected entity is typically the PC or other character. In strat-

[12]The author uses a convention, in which LOD 0 corresponds to a state, in which the technology has not been developed, and LOD 1 corresponds the first possible state (lowest level) of the "developed" status. All the possible higher development levels are numbered by always adding one to the LOD number of the previous, required LOD. It should be noted that common Boolean technologies can be discussed more generally as technologies with two LODS. The possible development modes of them, "not developed" and "developed", can be seen as LOD 0 and LOD 1, respectively. In this thesis, "achieving", "acquiring", "obtaining", "buying", etc., a technology means performing an operation that raises the LOD value of the technology under scrutiny from zero to one.

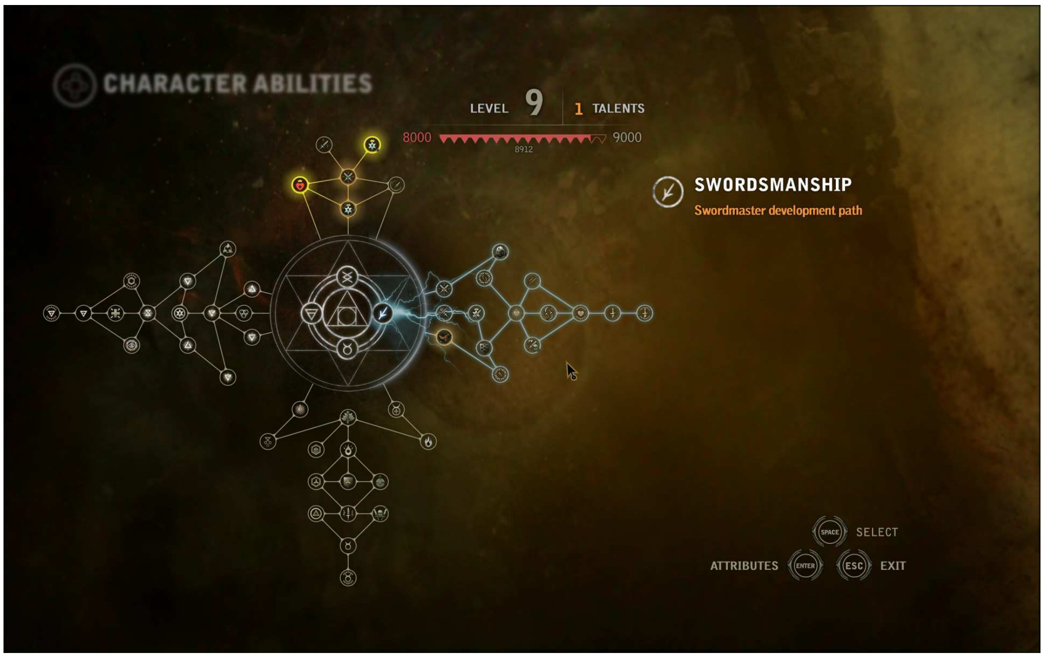
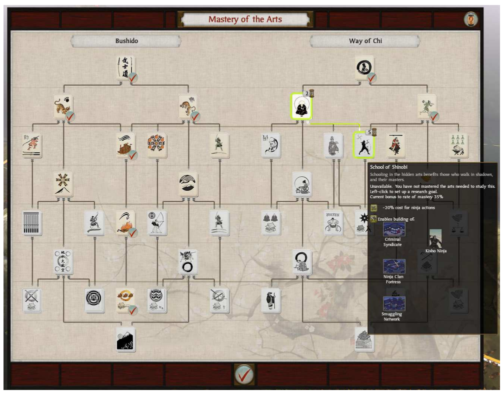


Figure 2.6: A screen capture from *Witcher 2*. Despite the ostensibly small sizes of the TTs, the development system is nontrivial, and the possibilities for character development are numerous.

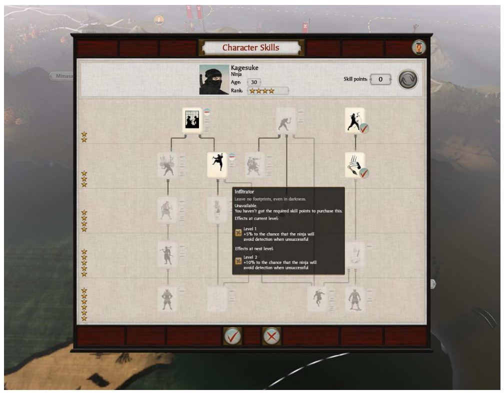
egy games, TTs – at least the "main TTs" – affect typically the whole faction (e.g., a clan, a nation, or a species) the player is guiding. However, there are also strategy games, like *Total War: Shogun 2* (The Creative Assembly 2011) (*Shōgun 2*), featuring, besides two "main TTs" (see Figure 2.7a), also TTs for individual characters (units) (see Figure 2.7b).

2.3.2 Using Technology Trees

Sometimes all the technologies are intended to be achieved in a game, but often this is not the case. In a strategy game, not being able to achieve them all leads to the factions remaining personalized [79], and the same principle applies to different genres as well. This way the resulting winning entities – such as PCs, factions, or species representing different players – probably differ from each other. However, TTs can also be seen as a wrong place for such personalization [79], and not being able to obtain all technologies may irritate some players. On the other hand, one of the main reasons to include



(a) A partial screen capture from *Shōgun 2*. Two “main” TTs of the game: “Bushido” and “Way of Chi”.



(b) A partial screen capture from *Shōgun 2*. A TT of a ninja.

Figure 2.7: In *Shōgun 2*, there are two TTs with clan-wide effects, but also generals, ninjas, and other important characters have their own TTs.

TTs in a game is exactly to force players to make – possibly even difficult – meaningful decisions. Differences of game types must be taken into account, but a general rule is that TTs should be able to offer enough interesting technologies to develop for the whole duration of a game – regardless of, for instance, the winning conditions.

TTs set different limitations for the development: if a player could, in the beginning of a game, just decide to take all the technologies with their corresponding benefits, the TT structure would be meaningless. Therefore, in addition to mutual prerequisite relationships of technologies, there are costs for developing them. The costs can be paid with in-game resources that come in many forms.

Even if other resources are not demanded, *time* is practically always spent in the process of acquiring a technology, at least in strategy games. In CRPGs, on the other hand, advancements are often atomic and momentary operations, but in these games acquiring the other resources necessary for a development operation requires time, nevertheless.

Sometimes the “raw” resources are not eligible to be directly converted into technology LODs, but there may be also intermediate “refined resource” forms. For instance, conversions from mud to bricks, ore to iron, or wood logs to planks may be required. In 4X games and other conventional strategy games, typical ways to proceed in a TT are by *constructing buildings* (especially in RTS games) and by *research.* Construction tends to require at least "raw" resources and sometimes also refined forms of them. Research work can be a totally abstract process, requiring at least time and possibly other resources as well, or it can be modeled to take place, for example, in research facilities by scientists. Other possibilities to obtain technologies or achieve LODs are, among others, espionage and trade with other game factions. Such mechanisms also require investing resources in them.

Games with CRPG elements normally base their development in gathering XPs by performing activities that make the PC more experienced. These activities require at least some time to be spent, and possibly other resources as well, depending on the nature of the activities. Experience can be gained, for example, by finishing quests, killing beasts, crafting items, or just running or jumping around, depending on the game.

For instance, a predefined amount of XPs – either fixed or depending on, for instance, the current character level – may be required to gain a *skill point* (or, e.g., a "talent" as in *Witcher 2*), and skill points (refined resources) can be used to buy technology LODs allowed by the TT. Also, buying LODs directly with, say, game currency may be possible, for example, in the form of getting trained in a skill by a non-player character (NPC) acting as an instructor (in which case the education offered by the instructor serves as a refined resource). Sports games and action games use somewhat similar mechanisms.

2.3.3 Common Problems with Technology Trees

Most TTs are rigid in the sense that once they have been defined, the structure and properties remain fixed until possible changes and corrections are made in patches and expansion packs. This kind of rigidity applies to almost all TTs in digital games published so far, and it is a direct consequence of the fact that TTs ought to make sense and be "good" and balanced, and, therefore, they are designed and created manually by humans. Publication VI introduces ideas to partly tackle possible rigidity-related problems and to generate technologies at runtime as needed.

Even if a customary design process is carried out by experienced professionals, the resulting TT may be of poor quality. When a TT offers many alternative ways to proceed – which is generally the idea – it is extremely hard to test all the possible routes of development combined to the actual gameplay. Even if problems in the tree are found, correcting them may not be trivial. A simple local correction – for instance, to a variable value defining a property of a single technology – may solve the immediate problem observed in a certain situation, but the fix may also have influence concerning well-behaving development path options and parts of the tree, thus introducing other problems manifesting themselves in other kinds of situations. In order to design TTs of high quality, evaluating their goodness is naturally important. Recognizing TT properties that are useful for such evaluations is paramount, as is also being able to measure these properties. TT properties and measurements are discussed in Publications III and V.

There are many ways for a TT in a game to manifest its poor quality and have a negative effect to the enjoyment of the gameplay. For instance, one can simply find that a technology progression allowed by the TT does not make sense semantically. For example, having a technology called *Calendar* to allow developing *Cat Food Factory* is a rather questionable choice without further explanation, because the nature of the relationship is not obvious. Even if technologies are clearly related, temporal oddities and "incorrectly" defined prerequisite relationships may cause the players to react negatively: it does not feel right, for instance, to have *Cavalry Archers* available before achieving *Archery* or *Horses*. Some kind of familiar – or at least understandable – causality is generally expected.

Another way for a TT to be flawed is to offer too obvious good choices. The TT is basically rendered needless, if the players always make the same, optimal-looking selections. Dominant strategies should be avoided. This is related to the issue of TT rigidity. Fixed TTs have a problem of letting players follow the same paths over and over again – something people tend to do; therefore, such TTs may appear as dull [104].

In any case, players should know what they are doing. It is problematic, if one does not know enough about the effects and consequences, when making choices, in which case selecting technologies easily becomes a frustrating "lottery".[13] Other common TT-related problems are listed in Publication V.

[13]To mitigate the effect of bad choices, sometimes possibilities to reassign, e.g., skill points are offered. Also, in strategy games an option to switch developed upgrades to others can be seen as a good idea [70]. However, such reallocations cannot often be used for free, so frustration may still occur.

Chapter 3

Game Development, Tools, and Academic Interests

"Man is a tool-using animal. Without tools he is nothing, with tools he is all."
– Thomas Carlyle [c]

As a phenomenon and an art form, digital games are still relatively young. It is impressive, how rapidly game development as a process has evolved. In a couple of decades, the hobby of lone coders has become a massive branch of industry of great companies with hundreds of employees, dedicated methods, and specialized tools. However, the motivation for this thesis lies in the fact that there is still room to improve game development processes further in many ways. Developing good techniques, methods, and tools for implementing digital games is becoming more and more important. Modern games should be produced in an economically viable manner, and the products of the process – games – ought to be of good quality in order to please the demanding customers of our times with continuously increasing expectations.

Game development has certain special characteristics distinguishing it from typical software engineering. This section continues covering essential background by discussing the nature of game development and related topics.

[c] `http://www.brainyquote.com/quotes/quotes/t/thomascarl399446.html`, accessed in May 2015.

Software tools play an important role in game development. Therefore, Section 3.1 covers programs and reusable program components and their use. Thereafter, Section 3.2 gives an overall view to the variety of people needed in the process. Moreover, two main game developer categories of interest are introduced and game modifications by end users are discussed. Section 3.3 continues by offering an introductory view to scripting – a widely used method that is important also concerning this thesis. After that, game AI is considered in Section 3.4. Finally, this study is given more context in Section 3.5 by discussing prior publications by others, other work in related fields, and tendencies within the academic community.

3.1 Game Engines and Software Tools

Nowadays, developing a digital game starts only rarely from scratch – especially, when considering commercial games, which are produced by large companies and targeted to masses. Instead, digital game development is largely based on using and specializing different *game engines*[14] and combining them with other ready-made software components.

Game engines offer varying degrees of support for designing, implementing, and compiling a game from pieces. For instance, the educational game engine CAGE [109] provides basic support for AI scripting, importing 3D models created with an external 3D modeling tool, playing sounds, rendering graphics, and so on. Some commercially used engines come with specialized visual tools for content creation and a multitude of functionalities. On the other hand, there are also engines with rather specific and limited scopes; for instance, graphics engines focus quite strictly on offering graphics support only. Customization and fresh code is needed in order to add on features and to modify the existing ones to suit the game under construction.

[14]Here the term game engine (and just *engine*) refers rather widely to different frameworks that are specialized and extended in order to create games. In this context any clear distinction between, e.g., graphics engines, physics engines, and "real" game engines – used to create game mechanics – is not needed; anyway, the terminology varies. A game engine can also be understood as an ensemble comprising parts such as rendering engine and physics engine [108].

Using engines, libraries, and other components saves time considerably and may also be practically necessary to meet the high requirements (concerning, e.g., graphics) of the customers. Also, portability can often be improved by using the engine approach – at least, if engines supporting multiple computing platforms are used. Moreover, game engines can be reused: exploiting an engine built for a specific game probably continues by using it to create also other games [108].

Hence, generally game engines are considered useful, like using ready-made libraries and such; it is commonly acknowledged that in software development a wheel should not be invented but once. On the other hand, however, it is highly improbable for any single engine to be perfect for the needs of a new game realizing novel and innovative ideas. Effects or functionalities that are not directly supported and that are hard to implement on a given engine may be needed. Using frameworks and components designed by others limits the freedom of the developers and sets constraints in various ways. For instance, the possible formats of usable external data may be dictated by the engine used.

An example of such data are three-dimensional (3D) models. Importing them to be used within a given rendering engine may only be supported for models in a very limited set of formats. Such a limitation may affect several aspects of the development process and the product: the quality of 3D models, viable modeling tools to be used, space requirements, and so on. For these reasons selecting wisely the ready-made components and frameworks to be used in a game development process is of utmost importance to minimize the extra work needed, to keep the structure of the program clear, and to be able to finally end up with a satisfying product.

Besides making use of benefits offered by different frameworks, libraries, and such components, game creation is facilitated by creating and putting parts of a game together using a wide array of software tools. Graphics are created with suitable graphics manipulation programs, character models and world shapes are created using 3D tools, and so on. Such applications, typically designed for some other ends, however, may not be optimal for the needs of game development [10]. In addition to such general-purpose programs, there are different tools designed especially for game development, and, as mentioned, tools may also be bundled with game engines. Moreover, game-producing companies often use self-made, proprietary special tools for different tasks. Level editors are typical tools created specifically for game development projects [87]. Typically these in-house tools are, unfortunately, kept undisclosed, and, therefore, the proverbial wheels are continuously invented over and over again.

Tools of high quality are crucial, but often creating them is still overlooked, and programming time for tool development is allocated insufficiently [87]. In real game development projects it has been found that suitable tools can result in considerable time savings and have a huge impact on the success of the process, and on the other hand, without good tool support the development can be quite tedious [84, 90, 98, 115]. Publishers consider toolsets as important factors when making decisions on backing game development projects [68]. Powerful tools that one could keep using to create different games and that still could answer to their specific demands well enough would be invaluable, but there is a trade-off situation between generic usability and suitability for a specific task.

Nowadays, a typical application interface is graphical. Graphical user interfaces (GUIs) have several advantages over more traditional command line interface (CLI) approaches; for example, learning using GUIs is faster [37] and easier [107], their use leads to fewer errors [37], and they can be seen as more intuitive [107]. However, GUIs may also be objected for lacking the needed precision, and designing good GUIs is not a trivial task [107]. CLI is sometimes more suited to tasks requiring more control, but it suffers its own problems.

As far as world creation tools (level editors and such) are concerned, it is important that the created content is presented visually to the designer during the editing [87]. The human visual system has powerful capabilities to recognize patterns [19], and the author firmly believes that using visualization (already during the design phase) is generally recommendable, because it helps to understand abstract structures. GUIs and visual representations have also been the mainstream approach already for a long time and become a norm concerning various tools and other computer applications, so they are typically expected by potential users.

For these reasons, the tools introduced in this thesis are graphical. They are intended to be easy to use and suitable for design tasks (besides their other aspects and purposes). When creating the tools, the goal has also been to make them let users have as much control over definitions, processes, and results as they would have using comparable CLI tools tailored to fulfil similar purposes.

3.2 Game Developers and Content

This section discusses mainly the diversity of game developers. The presentation is simplified by classifying some of them into categories of interest.

In addition to discussing the employees of developer companies, game modifications by end users are covered. Modifiability of games is an important issue which also relates to the developer classification, because gamers can be considered as developers, if they are able to modify games.

3.2.1 Heterogeneous Developers: Content Providers and Programmers

The design is, naturally, a crucial part of every piece of software of some quality. As far as traditional software engineering and conventional applications – such as text editors or database systems – are concerned, software architects (and such experts using other titles) design the structure of the software based on the customer requirements. They may dictate, for instance, technologies to be used in the actual implementation, since they have solid understanding of them. In addition to the software architecture, for example, interfaces may require design work.

Partly such basic facts of software development apply also in the case of developing games. However, when discussing "design", it should be taken into account that when the term is used in game-related contexts, it tends to contain (even) more aspects, meanings, and connotations than in non-game settings. In fact, "game designers" often have only modest, if any, actual programming skills [106].

The development teams of the first digital games were small in size; a team could consist of a single person who did everything involved in the process. However, when games grew larger, more complex, and graphical, more developers were needed, and thus the team sizes also grew. Furthermore, specialization was called for due to the increased demands. On the other hand, it also occurred naturally, when teams grew larger: all the persons involved in the development were not able to know and do everything equally well, but each had his or her own areas of expertise.

In order to create a "good" and interesting game, one should be able to provide a variety of different elements. Good game idea and design are crucial for all games. Depending on the game, a well-written story or meaningful quests may be needed. Nowadays, customers are used to require also lively sounds and vivid graphics. A seemingly intelligent AI is greatly appreciated and can turn a game to a success. All these elements should, of course, be of high quality and form a balanced ensemble together. The game should also be marketed, distributed, and the development process should be directed. Therefore, large heterogeneous developer teams with many different groups of people involved are typical.

Even if marketing, public relations, administration, and legal aspects – requiring their own respective staff members – are omitted and the focus is kept solely on the people contributing more directly to the actual product, there are still various roles left to consider. There are artists, software architects, programmers, game designers, playtesters, and so on. Of course, in order to achieve high product quality (i.e., to be able to create good games) and to keep also the developers happy, the goal is to distribute these roles based on the skills and interests of the individuals.

Therefore, creating narratives, level designs, graphics, 3D models, music, etc. are tasks often taken care of by specialists. On the other hand, there are also programming-oriented people – close to "traditional software developers" – involved, and they form a quite different personnel group [76]. This kind of an arrangement of using specialized developers makes it possible to excel concerning individual areas of development, if suitable experts are found. However, the other side of the coin is that creating and selling the actual game successfully requires good communication with mutual understanding between the personnel groups despite their different viewpoints and skills. This communication is one of the great challenges of the modern game development. Difficulties can be somewhat mitigated by using suitable tools, and this thesis aims at introducing such – see Publication IV.

In this thesis, the two personnel groups mentioned are called respectively *content providers* and *programmers*.[15] The basic ingredients of a game can be classified into two categories according to and explaining the nature of these categories; the division reflects the idea of a game consisting of a generic part (game engines and other general-purpose components) and *content* (distinguishing a game from others).

As discussed in Section 3.1, engines, generic libraries, and other such program components are essential in game development. People referred in this thesis as programmers are those who actually make additions and modifications to game engines and other "core" program components and implement the features needed. They also integrate different parts contributing to the whole together. Programmers are expected to be skilled in programming and in software development more generally.

On the other hand, besides generic program components, also content is needed in order to create a meaningful game. There are various types of content: game rules, textures, items, quests, characters, music, and so on [105].[16] Content is created by content providers. These people include graphic artists, story writers, narrators, composers, modelers, voice actors, level designers, and so on. They are a rather heterogeneous group of experts of different art

[15]Many classifications divide game-developing personnel into several professional groups. For instance, Fullerton [36] uses seven distinct titles to address these people: game designer, producer, programmer, visual artist, quality assurance engineer, specialized media, and level designer. However, concerning this thesis, using only two groups suffice because of the focus and the nature of the approach. This also simplifies the presentation that follows.

[16]TTs define rules to be applied in the game, often they have visual representations, and they can be closely related to characters (or other corresponding entities). Therefore, it is easy to add TTs into the set of our examples. In other words, labeling TTs as content based on their properties is natural.

forms and able to effectively use corresponding tools needed to create content for games. In contrast to programmers, however, content providers may lack actual programming and software development skills. Typically the majority of a game development team consists of content providers [112].

Concerning a single person, the roles of a content provider and a programmer may sometimes overlap. However, many people involved in game development processes can be clearly classified into exactly one of these categories; in modern large-scale game projects it is rare to have, say, a designer that is also a programmer [87].

3.2.2 User-Created Content

Nowadays it is common not to have to rely only on the content offered by the developer company. The end users of a published game are often provided with means to modify its content and even to create new assets for it. The terminology concerning the game-altering consumers and content created by them varies, and, for instance, the word "mod" can be interpreted in various ways [89]. The fan-programmer taxonomy of Postigo [83] makes a distinction between "modders", "mappers", and "skinners" based on the type of the provided content. In this thesis, all these are called *modders*. Respectively, the phenomenon of the end-users creating modified or additional content or changing the way a published game works is called *modding*, and the creations of modders, modifying or extending games, are called *mods*.

Modding phenomenon is not only beneficial for the modders (who gain fame and enjoyment based on their creations), but also for game developer and publisher companies. The value of fan-created content and gaming communities has been recognized by several companies [83]. Support for mods can, for instance, extend the full-price shelf life of a game [68], and is appreciated among the end users – naturally modders, capable of creating their mods, but also other consumers, able to get additional or alternate content and "fixes" to the basic game from the modding community.

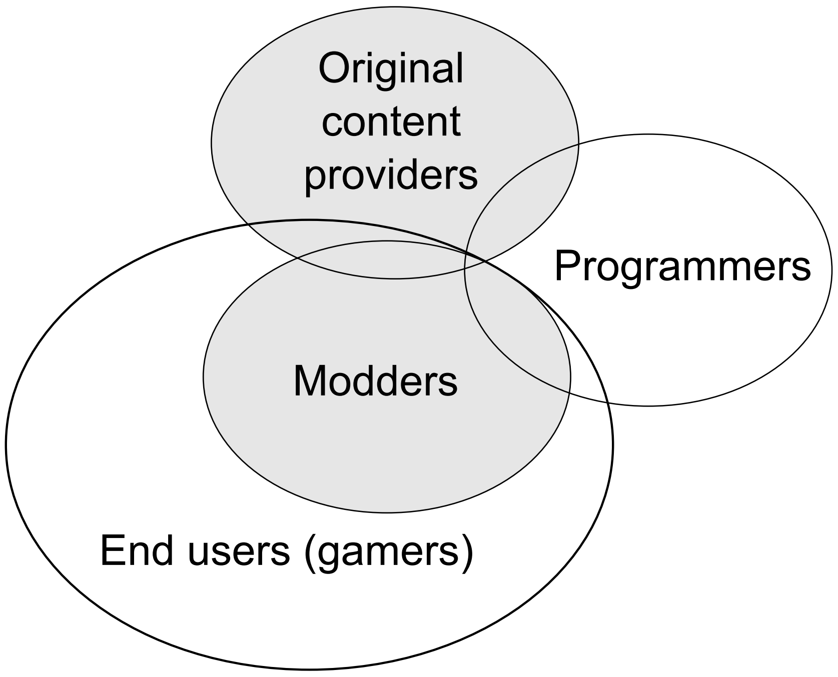


Figure 3.1: Some essential groups of people associated with digital game development. The modders have been depicted as a subset of end users, as it is hard to imagine a modder that does not use his or her own mod, but technically there could, of course, be such modders also. However, typically modders are seen as gamers (and fans of the games they provide content to). The areas of the set and intersection depictions are chosen arbitrarily and are not meant to demonstrate, e.g., cardinality relations.

Modders can be seen as content providers[17] for games, but typically they come from outside of the corresponding original game development teams. Often they are active gamers themselves. As end users they are normally able to start modifying a game and adding new content to it only after its publication. Figure 3.1 represents the people groups of interest as a diagram. The shaded region corresponds to content providers (union of the original content-providing developers and the gamers creating additional content or modifying the game).

Although development tools generally tend to be kept undisclosed, in order to allow modifying game content, it is common for a developer company

[17]Let us focus here only on the typical modifications – the ones concerning solely content, not the generic parts of games.

to publish alongside the game itself a "*modkit*", a special tool set for modders. Offering an attractive UI and restricting and documenting the modding possibilities as wanted typically requires modifying and polishing the original developer tools. This additional work, however, may pay itself back in additional revenues generated due to the user-made content. With adequate documentation and tool support, the creativity of potentially large masses of end users can be harnessed.[18]

All the companies have not, however, chosen to offer tools or even make it legal to modify their games. There are many reasons for this, such as the additional work required to develop modding tools and the desire to "protect" the company image and the content created by the employees from unlicensed or inappropriate use; it must be kept in mind that producing games is highly competitive business. It is also a safe choice not to allow arbitrary tinkering with the content used by a game, since users do not typically know the requirements for the data that can be used within the game successfully. For example, picture files may have very specific requirements concerning color palettes, sizes, alpha channels, etc. While this protective attitude of many commercial game companies is understandable, it may still be exaggerated.

One way to get user-created content created in quantity is, instead or in addition to offering traditional modding tools, to move content creation to be a part of the game. This lowers the threshold of gamers to contribute – content creation may even be the only way to proceed. A representative example of a game, in which in-game content creation has a major role, is *Spore* (Maxis 2008). In-game modifications of TTs might also be something to think about, but in this thesis content creation is treated as development work that can be done separately from playing. However, the ideas of TTs that evolve while games are played and possibility of gamers to affect this process are not dismissed. Actually, Publication VI discusses runtime TT generation.

[18]Of course typically it is, nevertheless, expected that companies develop "ready" games that can be played without mods. Some companies have been accused of using the player community as free workforce in order to finish games and make them actually playable. Intentional and transparent "make your game based on the tool set we offer" approach should be clearly distinguishable from "here is a (modifiable) game for you" approach.

3.3 On Scripting

The diversification of game developers has led to diversified code, and scripting languages have been put to use along with the conventional compilable programming languages; technical artists or level designers can create content, such as AI behaviors, with a simple enough scripting language [68]. According to a survey [8], scripting is widely supported by game engines, and scripts are heavily present in commercial games. Scripting is often used as a means to create data-driven programs [5]. One goal of this thesis is to introduce and promote the idea of moving the implementations of TTs into the scripts as well.

There are several scripting languages commonly used in the game industry. The language selection depends on things like personal affections of people in charge, library support, overhead size, and speed. Many general-use languages can be used for game scripting as such or with modifications – for instance, Lua, Scheme variants, and Python are viable choices – but also custom languages designed for particular games are used [68]. The purpose of this thesis is not to specifically promote any single scripting language, since the features that are needed and constraints that have to be applied vary depending on the game. The automated script-generation support offered by the tools created as a part of this study generate code in Lua. This language has been chosen to be used for its speed and because it is easy to interface with and use within typical main programs.

Facilitating the production inside a game-developing company is not the only benefit obtained by using scripts. Decisions such as providing content by scripting and moving descriptions and behavior outside the main binary in easily modifiable form also benefit modders.

3.4 On Artificial Intelligence

Although this thesis focuses more on producing good TTs than actually using them, the subject of use cannot be totally omitted. Not only producing TTs, but also implementing AI agents that can use them in a clever way, should be easy and cost-effective, since technology-advancing AI players have a central role in many typical computer (strategy) games.

In an ideal situation, a game should treat AI players ("synthetic players") and human players similarly [97]. As far as human players are concerned, it is enough to offer a clear user interface (UI) to a TT implementation and let the human brain decide how to use it. Offering an interface for an AI is not hard, but generally producing a good game AI capable of making human-like choices and decisions is.

Therefore, to increase the challenge for human players and to make AI look clever, game rules are not always similar for human players and AI agents. "Cheating" is common [97] – it can even be seen as a "standard tool" in AI programming [53]. For instance, AI players can have access to data (like the state of the whole game world) that human players do not possess, or AIs can be given free units or technologies. Sometimes such methods can be used successfully, but generally human players should be kept unaware of applying different rules to different players – otherwise the game is easily seen as unfair and possibly unnatural.

Creating a clever and efficient AI is often an intriguing and challenging task. However, it should be kept in mind that an AI playing optimally in order to win the game is not always the best one. An AI should not appear too foolish [87], but, nevertheless, applying artificial stupidity [60] suitably can make a game more enjoyable in many cases. The key for creating a good game AI is to make it act as a human being – seemingly quite cleverly, but not unerringly. One of the key issues, when creating game-playing AIs, is the adjustability of the challenge level. Hardcore players may tolerate even clear "cheating", if they in return are able to get enough challenge, but casual gamers and beginners may not be as forgiving. They easily get frustrated, if the AI plays too well.

The most common AI technology in digital games are finite state machines (FSMs) [35]. They have been used in digital game programming basically for the whole existence of these games, but still remain common [11]. FSMs are conceptually simple and easy to understand, and often also able to offer good performance. There are several ways to extend the basic construct [35], and for instance hierarchical FSMs offer certain kind of scalability. However, FSMs are not an effective solution for every possible need.

Goal-oriented action planning (GOAP) [78] and other goal-based approaches have been heavily used in recent digital games. Goal-directed behavior can be applied – not only to NPCs, but also to TT-using AI players [24].

In Publication II, a generic AI approach suitable for GOAP and implementing, for instance, AI opponents for strategy games is introduced, as is a related software tool. Also a tool implemented for creating easily functional FSMs that can be used from within applications is discussed. The tool makes use of visualization and automated code generation.

3.5 Related Work

Games "have generally been regarded as simple and insignificant pastimes for children" for a long time, and studies concerning them have largely focused on history and equipment concerned [16]. However, during the twentieth century, publications of Huizinga [47] and Caillois [16], among others, elevated cultural aspects and the nature of play and games to a topic of study. Since then, digital games have emerged as a distinguishable and important subset of games with special features; computerization has opened up options that make digital games something unprecedented in the history of games.

Also, game theory was solidified as its own field during the twentieth century by numerous publications (such as references [72, 73]), although game-theoretical results had already been known to date back a long time before them. Game-theoretical games are very easy to find in real life, and game theory is applied in several fields of science [22]. Concerning many games, combinatorial game theory (see, e.g., reference [20]) is an important subfield.

Roughly speaking, game theory is about making (optimal) decisions in a given setting, "game". In this dissertation, structures used to offer options in the context of digital games are studied, as is constructing these structures. Once an actual game (including all the necessary structures) has been defined and possibly implemented, game-theorists may apply their methods in order to determine, how the game should be played. However, real-world commercial digital games are extremely complex pieces of software, and because often they are only played for fun, game theory has better application areas elsewhere.

Nowadays, digital gaming is a wide field and suitable environment for approaches of different disciplines and interdisciplinary studies. On the one hand, this offers intriguing possibilities to cooperate and achieve interesting results, but on the other hand, this also may hinder the understanding and lead to misinterpretations and clashes between different views, schools of thought, and colliding research cultures.

The academic world needs time to adapt and form conventions concerning unaccustomed phenomena like digital games. The rift between ludologists and narratologists (see, e.g., references [28, 29]), misunderstandings [33] and the question whether (digital) games need their own discipline at all have stigmatized academic conversation and efforts in relation to digital games.

Games have been used in education for a long time. Already in the 1600s there were war games that can be seen to have been used educationally, and since the 1950s, gaming and simulations have found new educational areas to be applied in besides military planning [40]. Recently, digitalization and computers have opened up new opportunities, and educational games and possibilities to use game elements in education have become popular research topics. There are a large number of publications (e.g., references [4, 32, 110]) available focusing on these issues. Concerning the focus on this thesis, it is particularly worth mentioning that also technology trees used in games have aroused educational interest [102]. Education is by no means the only possible use for “serious games”. On the contrary, the application fields are numerous, and include also, for example, military, health, and government [86].

Health disciplines have also demonstrated interest (see, e.g., publications [55, 56]) in nonserious, conventional games played for entertainment – and especially their effects on players. The questions if there is a relation between gaming and violent behavior in real life and what exactly are the possible effects of violent games have remained popular – and well popularized –

topics for quite a time already. Many different theories and results have been published. However, so far consensus has not been achieved, and the applied research methodology has been problematic. Once again, there are a great number of publications (e.g., references [30, 41, 95]) available for more information. Besides negative effects, also positive impacts of gaming have been studied – see the review by Connolly et al. [18].

Gamification, which can be defined as “the use of game design elements in non-game contexts” [21], has also received a great deal of attention since 2010 [42], and the number of publications on the issue has been growing [45]. Education, once again, is a representative field of application for gamification efforts [100].

The game industry and its evolution and foundations have been studied from different viewpoints (see, e.g., publications [6, 80]). Also gender-related gaming and game character issues have been found interesting, which can be deduced from the multitude of publications on these topics (such as references [13, 54, 75, 113]).

The research projects and publications focusing on digital games have so far been largely about game design, games themselves, and social impacts of gaming. Studies of game design are important, because a badly designed game can never be good. So far the “unified theory of game design” is still missing [91], although there are various game design models and publications on the topic.

However, not even a good design is enough: an implementation is also needed in order to actually have a game. Engineering is considered the hardest part of developing a game [10], so there is a demand for academic studies concerning it too. The author of this thesis does by no means think little of game studies as they typically are. He, however, is currently more interested in practical solutions and views of game development from the implementational point of view than analyzing existing games or their effects or writing more papers on abstract game design. The design is strongly present in this thesis, though, because one of its central themes is automatic implementation based on design and reducing the workload of both game designers and programmers.

Many algorithmic solutions used in digital game development have been borrowed and adapted from fields like AI and computer graphics. Naturally, different more-or-less formal software development models, practices, and methodologies have been applied to game development. In the end, digital game development is software development.

Digital games also create a challenging environment for AI studies, and several AI researchers have been focusing on the problems related to them [82]. Game AI techniques differ considerably from techniques used in other fields [77]. Moreover, traditional AI methods able to tackle problems present in symbolic games do not often scale well to cope with digital ones [67]. Therefore, there is a demand for new methods.

Simulation games with their needs for decision-making within strict time constraints are extremely suitable to be used as testbeds in real-time AI studies [15]. Many games feature autonomous agents inhabiting virtual game worlds, in which they encounter varying AI challenges, and solving these problems has gained attention. Examples of relevant questions are, among others, how to adapt into a dynamic environment and how to plan behavior [71] – and even how to identify interaction possibilities with objects [94]. Digital gaming is known to be "an excellent platform for research in intelligent adaptive agents" [67].

There is a huge number of academic papers focusing on developing machine learning or game AI methods and testing them by implementing them for specific games (e.g., references [52, 64, 81, 101, 111]). Although games differ from each other, their AIs have similarities, and solving AI problems on a conceptual level can help to reduce code duplication and to allow better AI design [88]. There even are general game-playing AIs, such as CadiaPlayer [31], capable of learning to play a wide variety of games. Numerous textbooks (such as references [11, 14, 68]) have been written on the subject of implementing AIs for games.

Also publications of software tools for AI development (e.g., references [23, 34]) exist. AI architectures and middleware tools – such as TIELT [2, 69] – that can be integrated with different game engines have been developed. The approach of this thesis also promotes such portability. Moreover, even possibilities to generate game mechanics and design games automatically by AIs have been studied (see reference [118]).

TTs, the central focus area of this thesis within the digital games, have not been paid much academic attention. So far, publications really focusing on them, such as the article by Ghys [39], are very scarce. The communal pressure to make the "right" development choices in massively multiplayer online games and its effects are briefly discussed in [103]. There are also textbooks, like the one by Adams [1], discussing TTs. However, the understanding of the author is that this thesis and the included publications are unique in tackling the practical matters of implementational aspects of them.

End-user-created content and modifying games have been studied quite extensively, and several publications (e.g., references [26, 89, 99]) considering these issues exist. The TT approach presented in this thesis aims, among other things, to facilitate these activities. Tools usable by game designers without programming skills have been developed for generating scripts before [66]. However, specialization on TTs in this dissertation is something new.

Chapter 4

Separate Functional Modifiable High-Quality Technology Trees and Constructing Them

"Games are the most elevated form of investigation."
– Albert Einstein [d]

This chapter introduces the main contributive ideas of this thesis regarding TTs and producing them. Also, the main content and contributions of the included publications are explained. The suggested approach to facilitate creating good TTs with moderate workload consists of few ideas that are strongly connected and partly intertwined. Nevertheless, they are given their respective sections in order to express the components of the approach as clearly as possible.

In Section 4.1, effects of separating TTs from game engines and main game implementations are covered. It is followed by Section 4.2 about unification of TT development methods and formats. Section 4.3 discusses having functionality in TT implementations, after which Section 4.4 considers the benefits of implementing TTs as scripts. Then, Section 4.5 focuses on the view of the author concerning the preferable creation process of high-quality

[d]J. McGonigal: *Reality is Broken: Why Games Make Us Better and How They Can Change the World*. Random House, 2011.

TTs and combines the elements of the approach in order to give a summarizing view to the whole. Finally, Section 4.6 briefly presents the contents of the included publications in order to clarify their contribution to this thesis.

4.1 Technology Trees as Replaceable, Independent Entities

Based on the facts stated in Sections 3.1 and 3.2, the author proposes clearly separating TTs from other program components. TTs should be basically independent entities bindable to their respective users (main games or other programs) via a suitable interface. Modularity is a generally used principle in software development, and there are no reasons to make TTs an exception.[19]

There are many benefits to be obtained. Separating the TT entities clearly from the other components of a game would, at least,

- clarify the structure of the implementation,
- facilitate distributing[20] TTs,
- make it easier to build tools for manipulating them, and
- help in parallelizing game development.

Provided that the programmers knew the interface via which the main game communicates with a modular TT, they could use standard test TTs or TT stubs while building the game. Concurrently, the content providers (e.g., game designers) could create the actual game TTs using suitable tools created for dealing with such replaceable modular TTs. This possibility to parallelize

[19]BinSubaih and Maddock [7] have also noticed the benefits of enabling the existence of game logic, object model, and game state independently of game engines. Their approach and architecture ("game space architecture") aim to improve the portability of these three (rather generic) aspects and thus to facilitate migrating from a game engine to another. The approach presented in this thesis, on the other hand, does not aim at separating "the whole game" from the game engine, but focuses on TTs and thus is more specific. However, this approach can be used as a part of wider-scope separation schemes.

[20]This includes here, for instance, sharing products among gamers, offering them commercially, and dividing parts of a game implementation into different machines including server-based multiplayer schemes.

development work could be used to save time in the game-producing process. Moreover, the quality of the design could be evaluated early by a design tool (or a separate testing tool) – not only by the playtesters based on the more or less finished product, when it may already be too late.[21] The resulting high-quality TT could then be trivially put into the actual game by simply replacing the placeholder TT by it.

The modular approach would also facilitate modding: as a mechanism, simply swapping TTs with each other is extremely simple – at least, if TT entities are encapsulated in, say, TT files. Also distribution of mods would be easy, if TTs could be distributed as such, without any extra payload.

Multiplayer games connecting people over networks have become popular over the past several years. There has been a great deal of general and also academic interest in these games. This was the case already in 2000 [117], and the interest has not ebbed since. In a multiplayer setting, opening up a gaming system to be modified too much by gamers can easily lead to actions that are considered cheating [59][22]. However, online games with dedicated servers can overcome this problem by storing the official versions of the critical parts that should work similarly for all the players, and using them on the server side.

If TTs were implemented separately from their users, it would be easy to apply this idea to them. Uploading modified versions to be applied for all the players in a session by the governing server – even an "official" one – could also be allowed. This kind of a centralized system would make it possible to guarantee certain fairness. However, adequate computational server capacity and good network connectivity would, of course, be required. In addition, centralized systems are vulnerable to, for instance, denial of service attacks.

[21]Evaluating the quality of a TT programmatically is, of course, not trivial – it is a totally subjective matter that depends, among other things, on the game it is to be used with. However, some general guidelines and heuristics based on easily obtainable measures can be established and used in order to detect possible problems and enhance the design automatically. For more details, consult Publication V.

[22]Although the potential for such detrimental effects related to the modding phenomenon must be acknowledged and constantly kept in mind, generally the benefits that are obtained by allowing modding greatly outweigh the drawbacks. Therefore, it is generally a recommendable idea to allow modding. Moreover, it must not be forgotten that games can also be totally immune to harmful mod-based cheating.

In single-player games, local (or otherwise personal) copies of critical parts can be allowed to be modified and used freely. This is one of the reasons to focus on single-player games in this thesis; the fairness is not an issue, when the sole player is allowed to freely use whichever modifications he or she likes.

4.2 Technology Tree Implementation Format Unification

Modularity and replaceability of TTs serve well, when considering the development of a single game; the developers know all the necessary details, so they can take advantage of the features. However, in order to get benefits on a larger scale, unification of the ways to implement those modular TTs and their interfaces (i.e., standardization of TT implementation formats and interfaces) is needed. In the ideal case, all the game-developing companies and other developers would implement their TTs similarly and use standardized interfaces for communication between a TT and its users.[23]

In practice, of course, getting such a widely used specification for generic TT implementations produced requires the major parties interested in creating and modifying TTs to be able and willing to agree on a common format (or a small set of formats). This may be a challenging goal – traditionally, sharing technologies or interfaces has not been considered beneficial when trying to gain a competitive advantage – but, nevertheless, worth striving to achieve. The format should also be public.

Concerning this standardization, this thesis focuses solely on TTs, but other structural parts of game implementations should also be considered similarly taking into account the aspects of unifiability and possible benefits obtainable by generic tool support for editing hypothetical standard-format structures. The author believes that making game content – at least to some

[23]One might, of course, object the claim of this being the ideal case by stating that innovation is driven by competition, which is a valid argument, per se. However, the matter of optimality depends on viewpoints and objectives. In this thesis, reducing "unnecessary" development time and getting rid of frustrating incompatibilities are considered to be matters of high importance.

extent – more widely available, allowing modding, and facilitating it by unification concerning tools, formats, and methods used by different developers would be beneficial for all.

As already stated, modularization facilitates creating TT-related tools. It is a widely known fact that good software tools are essential in the game development, so it is generally worthwhile to create such tools (although sometimes developers fail to do so). However, unification of formats and interfaces makes it even much more worthwhile to invest in producing sophisticated tools for designing, implementing, and testing TTs. This is the case, because the use of a tool suitable for handling TTs in the hypothetical unified format is not limited to a single game or few similar games. Instead, it can be applied successfully in different game projects – even by different developers agreeing on the format. Hence, the returns of investments are considerably larger than with traditional tools, if good results are achieved.

Therefore, having a clear and easily available specification on a generic TT format and the corresponding user interface would facilitate creating tools to manipulate such generic structures. However, the ease of creation of tools does not mean that there should necessarily be several of them for any TT-related task. On the contrary, a wheel should only be invented once, and in an ideal case there could only be one tool that all the interested parties (including different large-scale commercial game developers and independent developers, as well as individual modders) could benefit from.[24] This means that there would be no need for every developer to implement an in-house version of virtually the same tool, and the game developers could focus on actually developing games.

Using the same (possibly open source, free-to-use) tools for original creation and modding the content after the publication by fans makes sense: there would be no costs for developing special modkits, if the modders could use the same tools as the original developers. This way the developers could

[24]Unfortunately, we do not live in an ideal world, so maybe there will not be any single, perfect tool. However, the public common TT format would make it easy to produce generally useful TT tools, e.g., by open source projects using liberal licences, and game developers could benefit from them without necessarily using much resources on the tool development themselves.

save time and money while letting their products be modded with high-quality tools.

As far as the scenario is considered from a point of view of a single developer or publisher, the quality of the tools naturally matters considerably. No one wants to use low-grade tools voluntarily – and even less keep using them for different products, as would be the case with generic TT tools. On the other hand, the general applicability of the hypothetical tools implemented for creating and modifying various kinds of TTs means that investing resources into making them reality pays itself back manyfold: having a quality tool that can be used after and beyond the immediate need – the game currently under development – is invaluable for game developers. The capability to offer finished tools to modders without great additional efforts is also a huge bonus.

For these reasons, if a format was agreed on, the developers would have a vested interest to get their hands on such tools, even if it took some time and money. Instead of undisclosed in-house projects, these efforts should be put into open source projects, even though this might benefit also competitors, because this would speed up the process of adopting the format by different developers, strengthen the whole idea, and increase the benefits obtained.

Although letting modders change parts of a game (for instance, TTs) freely is generally a desirable goal, some developers may want to limit or restrict modifications (while still allowing some). This should be supported by the tools used in TT development. Such limitations may be necessary, for example, if the main program makes non-standard assumptions. Therefore, the TT implementation format should offer possibilities to impose restrictions and include suggestions. For instance, lower and upper bounds for the safe operating intervals for variable values could be included in order to avoid "mystical" problems. If developer companies would like to protect, for instance, their heuristic functions for tree evaluation, the tools could be extended or modified in-house, but the TT compatibility with the basic versions of tools available to modders should be maintained, nevertheless.

The firm belief of the author of this thesis, based on a prototype tool implementation (see Publications IV and V), is that such generic tools and

TT formats are viable. The challenge now is to get the game developers to endorse the approach (and then to define the exact format and get the actual tools created and used).

4.3 Technology Trees as Functional Entities

TTs could be implemented as augmented graph data structures that store technologies (nodes), dependence relations (edges), and all the required additional data (that could be, e.g., textual, numeric, and pictorial). If such a structure could be edited programmatically via a suitable interface, such trees could be used by main programs. Tools to manipulate such structures easily could be developed, and if the structures were encapsulated to be effortlessly replaced and distributed, they would basically satisfy the requirements for a good TT implementation presented so far.

However, such a solution would essentially be only a relatively simple container, and this view fails to capture the whole essence and nature of TTs. They are definitely more than containers: in games their meaning and importance come from *using* them, and TTs are relevant, because the *development affects* the game. A good TT implementation might offer support for realizing these crucial aspects.

TTs come in various forms in different games. There are, however, functional requirements common to a large portion of real-world TTs. The functionalities could be realized in the main program code, but on the other hand, TT implementations themselves could also be made capable of performing typical tasks.[25] The author thinks that this is the preferable approach, because nowadays, for example, various data structures[26] included and used in different libraries and languages can perform complicated tasks, and the popular object-oriented programming paradigm endorses objects that are able

[25]This idea does not contradict the content of Section 4.1, because game-dependent functionality is not to be moved into TTs, but only generic TT functionality.

[26]One possible angle to TTs is to see them as data structures, and they may be implemented as such.

to act: encapsulating data with operations operating on them is one of the "key aspects of object-oriented structuring" [58].

TTs are typically meant to be used by different software agents as well as human beings. However, often the interaction is simply selecting the next technology or technologies to be developed. It is also possible to include AI components capable of making such decisions into TTs, and in some cases it may even be wise to do so. Because often there are also other – not directly TT-related – decisions to make, it may feel natural to have, for each TT-using software agent, a dedicated program component that implements the needed AI functionality. However, this "main AI" might delegate a part of its work to TTs.

Whether a TT is allowed to make the decisions concerning development by itself or not, it should be aware of the current development possibilities. Even if the decisions are dictated by some other part of the software (as is the case at least when human players make the choices that are then conveyed to the actual TT implementations), it simplifies the matters if the TT can provide information about which (if any) technologies can be developed in the current situation. After all, a TT is the program component that has the knowledge of the technology requirements, and passing unnecessary data within a program should be avoided in order to keep the operation simple and to save computational resources. However, in the name of versatility, for instance, different queries and manipulations concerning statuses of individual technologies via the user interface of a TT should be enabled.

In Figure 4.1, the main idea of a functional TT is illustrated. The user of a TT (main program) is able to get information of the tree and manipulate the state of it directly, but it is also possible to let the tree govern itself, in which case the role of the user is basically to feed the tree with resources to be consumed.

In addition to the basic functionalities such as automatic development, resource bookkeeping, or calculating statistics, there are also other functionality types that TTs can implement. An interesting example of such is the capability to perform self-modifications. A functional TT can be able to build or augment itself (see Publication VI). It could also, for instance, modify its

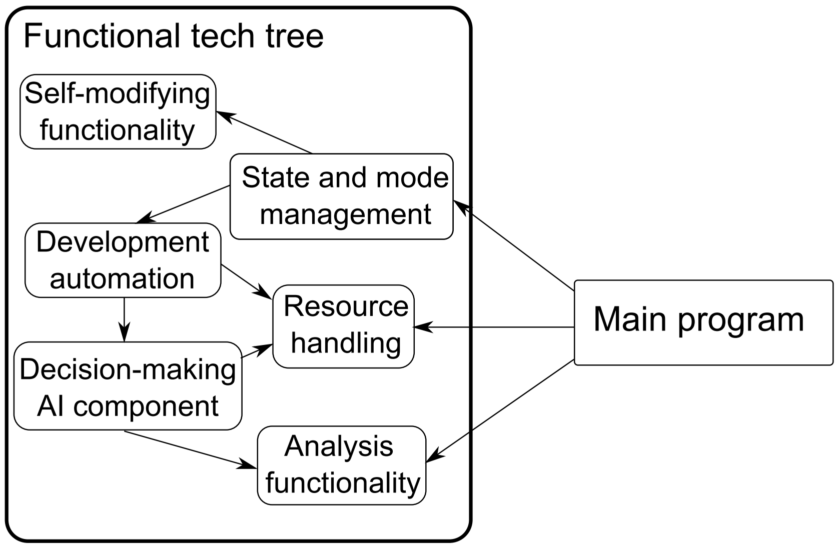


Figure 4.1: A TT with internal functionality to perform TT-related tasks.

properties based on what it learns from the player in order to enhance the playing experience. (So far, such functionality has not been included in the prototype TT format. It is, however, an interesting and natural addition to be made in future.)

4.4 Technology Trees as Scripts

Previous sections have discussed having functional TTs in a unified format separate from their users. Such structures could still be implemented in various ways. This thesis proposes using a scripting language and standardizing a format for TT implementations in it.

Scripting languages have already been used widely in game development, so the approach ought not to come across as exceedingly exotic, but useful, nevertheless. There are at least four good reasons backing scripting TTs:

- **understandability** offered by typical scripting languages makes the code written in them ideal to be edited even manually, if needed;
- **accessibility of the data** by using normal text editors (in an absence of sophisticated TT tools) increases robustness;

- intrinsic **capability to create functionality** of scripting languages facilitates implementing functional TTs using them; and
- some scripting languages offer such **ease to interface with** them that the separation of TTs and other parts of games does not become a burden.

Using scripting languages offers various benefits. For instance, if a script is interpreted at runtime and not compiled beforehand – as often the main program is – modifying and tuning scripted behavior or variable values do not require compilation before testing. This may result in considerable time-savings in the development process.

Conventionally, scripts have been used, for instance, for AI purposes. Some games actually have already implemented also their development processes using scripts. For instance, the PC development of *The Witcher* (CD Projekt RED 2007) is based on Lua scripts [63]. However, scripting formats tend to be specific to the corresponding games. Moreover, possibilities to adjust TT properties using scripts may be quite limited; for instance, restructuring and augmenting an actual TT graph (i.e., adding and removing technologies and connections between them) may be impossible, if the structure and the graphical representation in the GUI are hard-coded.

If one desires to create a game that is modifiable (by modders), content should be kept in an easily editable form in the final product. Typically, however, much of such content is included in the final games in a form, in which one cannot easily modify it – at least not directly with only common general-purpose tools.[27] Modkits can offer access to data and perform format conversions, but the author finds that in order to keep implementations simple and to facilitate tool creation, simple textual formats[28] should be favored as far as they are practical and applicable without major drawbacks.

[27]One typical reason – besides striving to secrecy – may be the need to compress the data in order to limit the (often huge) disk space requirements. As far as TTs are concerned, the typical sizes are quite reasonable, even with, e.g., associated graphics. Therefore, the benefits of using compressed formats to store TTs are rather limited.

[28]Of course, other "open formats" can be used as well, but textual ones have the benefit of being straightforwardly understandable by humans.

This means that using scripting is a good alternative as a means to support modders.

The scripting approach is suggested in this thesis, among other things, in order to achieve freedom to modify TTs easily. The goal of establishing a common implementational format – a language combined to a generally known structure and its semantics – for TTs may seem to be restrictive and against the freedom of developers. However, a good format does not restrict modifying trees, but only fixes interfaces.

Within a TT implementation format of high quality, one can still make the tree behave as desired, and the requirements should not make this overly difficult. Naturally, the (fixed) interfaces between trees and their users must also be versatile enough, so that they do not become a bottleneck limiting creativity concerning TTs. These issues must be taken into account, when pursuing to create such a common implementation format. However, the prototype format, which the tool introduced in Publication IV uses in its TT generation, and the corresponding interface, via which these TTs can be used, show that this is a realistic task. Designing script-based, generally usable TT formats is, indeed, quite possible.

4.5 Creating Technology Trees

The previous sections have revealed – at least on a high level – what kinds of entities TTs ought to be. Besides these principles and characterizations, one of the most crucial contributions of this thesis is a proposal to use specific software tools, built regarding these points of view, in order to actually implement high-quality TTs easily. In this section, the overall tool-facilitated TT creation process is discussed.

Figure 4.2 summarizes the "big picture". On the left-hand side, developers are divided into content providers and programmers. Modders can be included in content providers, but also a separate modder (the upmost actor in the figure) has been depicted to use a special modkit and via it to create or modify scripts. A modkit could also include, for instance, modeling or image manipulation capabilities, so there could be an arrow pointing also to

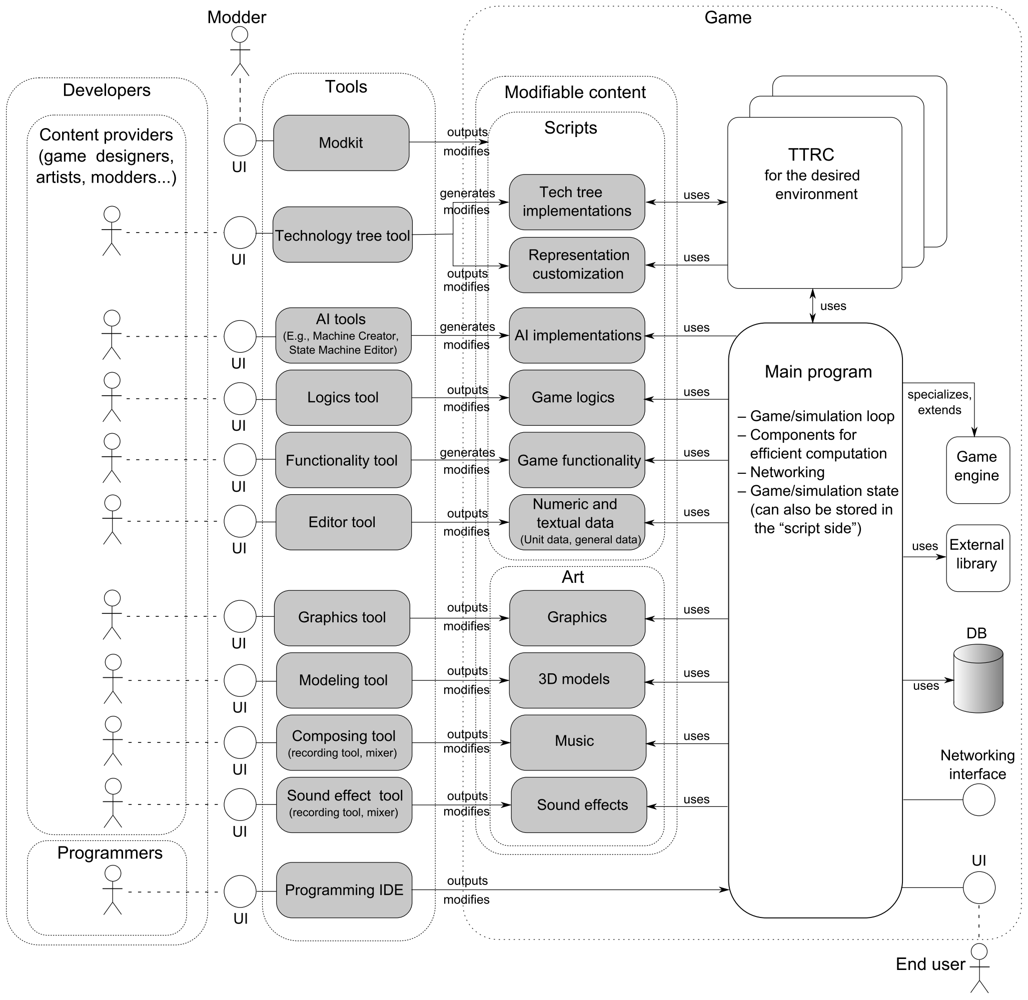

Modder
Game
Developers
Tools
Modifiable content
Scripts
Content providers (game designers, artists, modders...)
UI
Modkit
outputs
modifies
Technology tree tool
generates
modifies
Tech tree implementations
Representation customization
TTRC for the desired environment
uses
AI tools (E.g., Machine Creator, State Machine Editor)
AI implementations
Main program
– Game/simulation loop
– Components for efficient computation
– Networking
– Game/simulation state (can also be stored in the "script side")
Logics tool
Game logics
specializes, extends
Game engine
Functionality tool
Game functionality
Editor tool
Numeric and textual data (Unit data, general data)
External library
Art
Graphics tool
Graphics
DB
Modeling tool
3D models
Composing tool (recording tool, mixer)
Music
Networking interface
Sound effect tool (recording tool, mixer)
Sound effects
Programmers
Programming IDE
End user

Figure 4.2: An example of a digital game (and content) creation process presenting important components, actors, and their relations.

the "art" block besides the one pointing to the "scripts" block. However, as far as TT property and logics modding is considered, the "scripts" block as outputs suffices. Below the modkit, several other examples of tools needed in game creation are shown.

Tools should be suitable for their tasks and effective and efficient in performing them. Programmers ought to have appropriate tools for building main game programs (which, in the approach presented here, depend on the external game content). In order to create high-quality content, good content creation tools are invaluable. By creating and using replaceable content, the content creation can be efficiently parallelized with main program development.

4.5.1 The Trinity of a Game

In Figure 4.2, a game consists of three major parts: modifiable content, TT representation component (TTRC) (or several of them), and main program. Let us consider them next, starting from TTRCs, which are basically ready-to-use program components acting as adapters and enabling the versatile use of the approach in different environments.

TTs often affect the game graphically. They can store pictorial and descriptive data, for example, in order to present the correct unit graphics in a given state of the world and so on. Such data can also be stored separately and used as guided by TTs. Moreover, many games also require a graphical representation of the TT structure itself. On the one hand, offering an overview of the graph structure and possibly also detailed information on particular technologies to players visually is often desirable. On the other hand, typically these graphical representations (the corresponding software components) should also be able to let players guide development by easily selecting technologies to be developed. Visualizing the viabilities of making different choices is a part of this. In other words, graphical interactive TT representations (GTRs) act as parts of the GUIs of games, and the purpose of these parts is to let players interact with TTs easily and meaningfully. Several screen captures in this thesis are examples of GTRs (see, for instance, Figure 2.3).

There are different kinds of graphics solutions used by various games and engines. Therefore, GTRs are somewhat problematic. However, instead of implementing GTR components always from scratch for each game, the author proposes producing easily customizable GTR components[29] that could use the standardized TT implementations via the known interface. The

[29]Such a component could be, e.g., a widget to be used within a graphical main program in order to produce the TT-related part of its GUI.

TTRC part in Figure 4.2 represents such components. The appearance and behavior of them could be specialized and adjusted by scripting.

If TTRCs using standardized scripting and script-using interfaces were implemented for different engines, environments, and platforms, getting a working GTR as a part of a game would be easy. A main program could use suitably selected TTRCs like any other graphical components used in the game-implementing environment, the TTRCs could use TTs implemented as scripts in a standardized format, and such TT implementations could be produced taking advantage of automated code generation offered by suitable software tools. Modifying the appearance and behavior of TTRCs would be easy by scripting, and the necessary scripts could also be generated automatically by the high-quality software tools used to produce TTs.

Main programs can be specialized and extended game engines and they can use external resources like libraries and databases. They also offer the needed UIs for their end users (players) and implement core functionalities that should not be modified or are hard to move to the block of modifiable content, which is the third key game component in the illustration.

The "modifiable content" block is divided in two sub-blocks, "scripts" and "art". The "scripts" block consists of content in script formats (which, in a finalized game, may be either textual or binary). To a considerable extent, important data – for instance, attributes used to define NPC characteristics – and logic can be scripted so that benefits are obtained. However, for some content, other (binary) representation formats may be more suitable. The "art" block in the illustration includes examples of such content.

4.5.2 Automation and its Limits

A TT can be designed using easy-to-use graphical software tools, and the actual implementation can then be generated automatically (see Publication IV). Automated code generation used to produce TTs is one more reason to have functionality in them: when generating TTs (semi)automatically, there is no need to restrict the software to produce only a structure to store the essential data and fill it, but typical TT functionalities can also be generated automatically as parts of the TT implementation without practically any drawbacks or additional costs after implementing the generation capability in a tool.

Automating TT code generation is straightforward as far as the implementation format in a scripting language is known. Therefore, standardization of formats used to represent content makes it easier to implement good generic tools, because the efforts can be focused on few targets instead of many tools for different implementation formats. As mentioned, also creation of TTRC-customizing scripts can be automated.

Improving TTs and monitoring their quality form another important area, in which automation can be successfully applied. The process can be at least partially automated. Publication V discusses these matters. It is possible to find potential design or parametrization problems, and even to improve TTs designed by humans automatically already during the content creation. Considering aspects of evaluating TT designs, computers have their own strengths, when compared to humans. These strengths should be used, and not only in the traditional testing phase of the development cycle, but as early as possible.

Time savings achievable in a game development process by simple automated code generation are obvious. Also, measuring TT properties and making adjustments automatically seems to be a beneficial approach, even though more testing involving real game development is needed. However, it is also obvious that estimating all the interesting features of TTs automatically cannot be done easily. Some features are trivial for human beings to evaluate, but challenging for computers, and vice versa. Therefore, it seems that both are needed in a development process in order to optimize the quality for now.

4.6 Introduction to the Included Publications

This chapter has discussed the proposed approach to TT creation. Partially the ideas have been presented in Publications I–VI. This section briefly summarizes the actual content of the publications in order to crystallize their roles and contributions to the whole.

The descriptions are kept short, because the actual publications are included. They should be consulted in order to get more detailed view of the ideas, methods, implementations, and results.

4.6.1 Publication I: Technology Trees in Digital Gaming

Publication I is an overview on TTs and their roles and usage in different digital games. The paper creates a foundation for the other included publications by introducing TTs generally and by presenting fundamental views of the author on them. Therefore, it is recommended to read Publication I prior to the other publications.

Several possible TT classification criteria are introduced in the paper. The criteria discussed there affect the publications about measuring TT properties (i.e., Publications III and V). Publication I also contains a table representing observations of 17 real computer games with respect to the criteria mentioned.

4.6.2 Publication II: Implementing Artificial Intelligence: A Generic Approach with Software Support

Even though the main focus of the thesis is on creating – and not as much on using – TTs, they should also be usable by artificially intelligent entities. Therefore, Publication II has been included. Although it does not discuss TTs directly, it introduces an approach consisting of two methods and corresponding tools for creating AI implementations. Both the methods and the tools can be used, for instance, to create TT-using AI agents.

The discussed AI implementation approach corresponds to the view of the author on how to implement TTs: it is generally a good idea to separate AI implementations from the main programs and create them in scripting languages using performance-boosting tools. The reasons are similar to the reasons concerning these matters in TT creation. In fact, the whole idea to strive for a unified format and tools to create scripted TTs was born based on the good experiences gained in relation to the AI study that led to writing Publication II.

The article discusses mostly creating AI implementations for NPCs. However, the encountered challenges are largely the same also concerning TT-using AI agents. Proceeding in a TT can appear as a simpler activity than, for instance, a holistic management of the state, goals, and activities of a complex NPC. However, similar qualities are useful for both kinds of AIs; for example, believability, human-like behavior, and ability to plan ahead are important common requirements.

The main scientific contribution of Publication II is presenting a general-purpose AI framework. Technologically, the publication contributes by introducing two software tools created to facilitate implementing AI agents. The tools and the suggested approach are demonstrated to be practical by using them to implement three example game AI scenarios. One of them includes GOAP NPC AI implementations, and the other two are different adversary AIs: one for acting as an opponent in a board game and another for playing as a strategy manager in a RTS game.

The AI approach proposed in the publication is very generic and applicable to various needs. Concerning this thesis, the strategy manager example implementation is interesting, because it demonstrates a cascading and hierarchical way to use "AI machines" in order to achieve an AI that is easy to understand in terms of both structure and functionality. This kind of an approach is suitable and useful also when dealing with TTs; a TT-using AI component could be guided by components governing, for example, attitudes towards other players, economy, and strategic or tactical planning. Such a modular structure of a player AI consisting of several simple cooperating AI modules – including a TT-using AI – with their own responsibilities would clarify the implementation and facilitate parallel development work.

4.6.3 Publication III: Considerations on Measuring Technology Tree Features

Publication III starts considering possibilities to measure properties of TTs in order to be able to automate enhancing them. Three viewpoints to analyzing TTs are highlighted: local, global, and temporal.

Besides the three aspects and general discussion on the problems and possibilities of measuring TTs, the publication contributes by introducing a way to represent a simple TT in a fashion that makes it easier to spot certain kinds of problems visually. This format for TTs is called "time layer topology format". Also, an algorithm for converting a TT into this form is given.

Discussion in the publication is rather abstract and theoretical, and the presented ideas and characterizations are refined in the publications following it. Nevertheless, Publication III functions as a crucial stepping-stone on a journey towards the technological contributions of Publications IV and V.

4.6.4 Publication IV: Facilitating Technology Forestry: Software Tool Support for Creating Functional Technology Trees

Publication IV is about creating TT implementations according to the principles promoted in this dissertation. A model for implementing generic functional TTs, separate from the programs using them, is presented. Moreover, a prototype software tool (Tech Tree Tool, TTT) capable of actually generating such TT implementations as Lua scripts based on the designs created via a GUI is introduced.

The paper highlights the variety of TTs and sketches a generic model that can be used very flexibly in order to create different TTs. TTT serves as a proof of concept demonstrating the feasibility of visual tools able to partly automate TT creation processes. A simple game using TTT-generated TTs is discussed, and the overall approach of creating functional TTs implemented as separate scripts with the help of graphical tools is found viable; the TTT GUI turned out to be suitable for TT creation, the TTs created could be easily used by the main program, and the desired functionality could be achieved without difficulties.

4.6.5 Publication V: Quality Measures for Improving Technology Trees

Publication V continues the discussion on TT property measurement possibilities started in Publication III. While Publication III is basically a short theoretical introduction, Publication V applies and deepens the presented ideas.

The central contribution is introducing and explicitly pointing out several measures that can help one to understand the characteristics of a TT. The article also illuminates, how TTT (see Publication IV) has been augmented in order to automatize the measurements. For a demonstration, TTT is used to measure properties of a real commercial game, *Civ 5*.

4.6.6 Publication VI: Augmenting Technology Trees: Automation and Tool Support

Augmenting TTT, introduced in Publication IV and improved in Publication V, with additional capabilities continues in Publication VI. In this paper, generating technologies automatically (based on a given starting situation) is discussed. A particular approach to model and implement technologies is covered; the idea is to make the design work performed by humans more abstract and to let the computers take care of the details of generating technologies using the given design guidelines.

The automated technology implementation adheres to the principles established in the other publications: TTs are still functional, scripted entities separate from their users. Based on preliminary testing, the approach is found viable.

Chapter 5

Discussion

"Always pass on what you have learned."
– Yoda [e]

In this chapter, general thoughts and issues that occurred during the research work are discussed. These matters are more related to the research process and observations than any single publication. Moreover, some general notions concerning the research topic and the state of the art are presented.

TTs are a surprizingly deep and complex phenomenon. Basically they are very simple structures that may not even seem worth studying, but actually the diversity of real-world samples is overwhelming. The real depth and extent of the subject did not occur to the author before actually starting studying TTs.

Consequently, because there is a huge number of various possibilities to use different kinds of TTs, one might think that the traditional way of creating ad hoc implementations is the correct one. However, despite all the variety concerning functionality, representations, and purposes, the prototype implementation format for functional TTs created during the research process is suitable to be used to imitate all the TTs of the real games that the author has been able to remember, test, or study during the work.

[e] `http://www.yodaquotes.net/always-pass-on-what-you-have-learned/`, accessed in May 2015.

This means that TTs – in all their diversity – seem to have such a common core structure that they can be implemented using fixed (e.g., scripting) formats. Naturally, the expressive powers of such formats must be adequate, but at least based on the prototype, it seems that overwhelmingly complex or large structures are unnecessary. When starting studying TTs, it was unclear, if it was possible to find such a relatively simple format or not.[30]

Research work focusing on TTs has – at least for the time being – some special difficulties associated with it. One major issue has been the lack of prior publications on the topic. In the beginning of the pursuit towards the dissertation, the author could not find practically any peer-reviewed scientific publications focusing on TTs in digital games, and still they are very rare. Hopefully this thesis encourages more people to continue with the topic.

Another issue has been the difficulty to get information on the real commercial game implementations. Lacking actual commercial game development experience so far, the author has had to rely on a variety of sources of varying reliability. He believes that the overall impression established is rather correct, but some more openness by game companies would make it easier to conduct scientific studies. In order to verify the established views and to get more insight into "typical" ways to develop TTs, three companies were asked questions concerning their TT development processes and needs. Only one of the companies actually answered. These answers [17] are in line with the general statements regarding game and TT development in this thesis.

One should be able to compare the new technologies with the alternatives used in the industry [96]. However, in this case, this has not been possible. Because there have been no relevant software or other artifacts available to

[30]Of course, a format suitable for different kinds of TTs could have been defined by taking a union of all the formats defined particularly for each different kind of a TT encountered and thus defining a format with several separate modes. This, however, would not have been very useful; it would be laborious to create tools for such a combined format, and the users of these tools could not benefit very much from their prior experience of working with other kinds of TTs. Moreover, a specification for such a format would be very large and hard to maintain. Therefore, it is a satisfying result that pretty different functional TT implementations can be scripted so that they feature a simple, common interface for using them and a smallish, common set of internal functions. This also facilitates creating generic TT-creating tools.

compare the products of the work with, the evaluation possibilities have been somewhat limited. The constructive and experimental pioneer approach has led only to "proof of concept"–type demonstrations, which are far from perfect evaluations, although common in the field of software engineering [96].

Were there other similar tools available to be compared with the ones developed during this work, one could arrange tests and formally evaluate the results better. Although the scientific value of this thesis is, on the one hand, diminished by its lack of formal evaluation, on the other hand, in future, the created methods and artifacts can be used as references when evaluating new approaches.

The non-quantitative results allow some subjective bias, when interpreting them, but on the other hand, in this case the author does not really have any vested interest to "make the results good". Of course, it is pleasant, if the artifacts will really be useful in real game development some day, but on the other hand, as far as this dissertation is concerned, the results as such do not really matter. Had the author felt that the methods are of no use, that result would have done as well. The author honestly believes that the artifacts are usable and suitable for many games, purposes, and situations.

The lack of prior scientific work focusing on implementing TTs calls the importance of the whole topic into question. It is probably true that generally creating TTs is not recognized as a major problem in the game industry; there are many other demanding tasks to consider as well, and it is easy to select one of them to focus on.

Nevertheless, TTs are important elements present in many games, and the quality of a TT directly affects the quality of the game using it. Software tools are crucial in game development and greatly affect the efficiency of the development process. Also, finding design flaws as early as possible is important, because typically fixing a design becomes more expensive as time passes. For these reasons, the approach presented in this thesis should not be overlooked and ignored. Moreover, besides implementing TTs, some of the ideas can be applied to other tasks and subproblems as well.

In the game industry, there have been some attempts to replace traditional TTs with other kinds of structures. A recent example is *Sid Meier's*

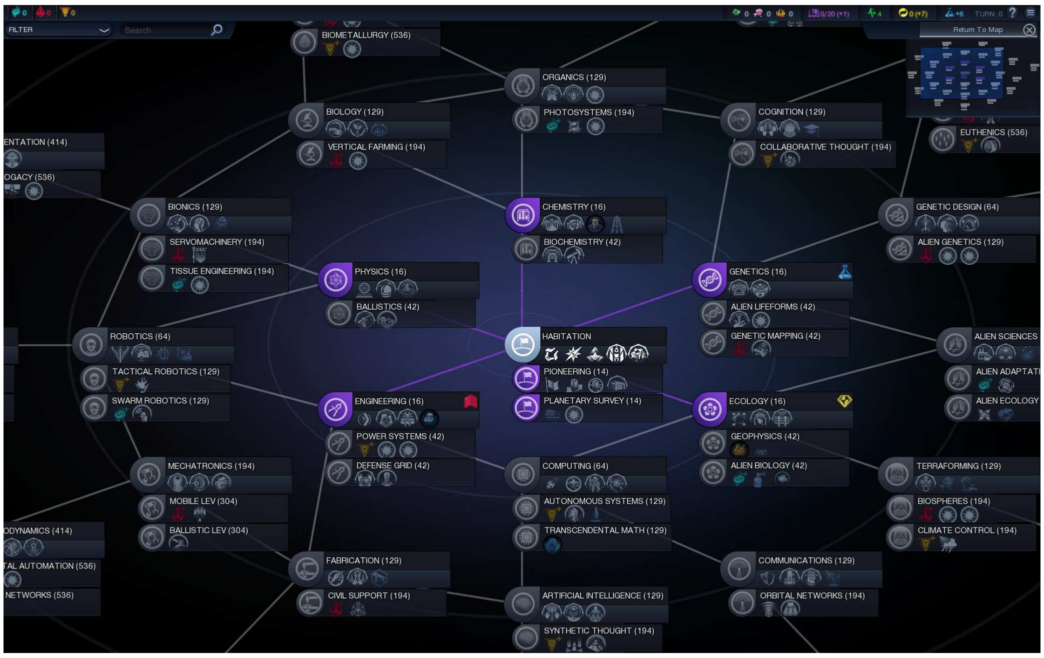


Figure 5.1: In the “tech web” of *Sid Meier’s Civilization: Beyond Earth*, the starting technology is topologically in the center of the graph. A screen capture from the game.

Civilization: Beyond Earth (Firaxis Games 2014), in which the pretty linear, “Civilization style” TT (see Figure 2.3), characteristic to the earlier games in the *Sid Meier’s Civilization* series, was replaced by so-called tech web (see Figure 5.1). There is more freedom to choose the direction of the development, and the graph structure resembles a graph-theoretical tree even less than before, but nevertheless, this structure still fits to the definition of a TT used in this thesis and can be modeled and manipulated with the tools presented without problems.

Another interesting example of a recent game is *The Witcher 3: Wild Hunt* (CD Projekt RED 2015), in which one can select technologies (“skills”) to be developed from different TTs, but only a selected subset of a limited cardinality of these technologies can be active at any given time. During a game, the player can freely alter the subset by inactivating technologies and activating other ones by assigning them to slots defining the technologies currently in use. The number of available slots is slowly increased during the

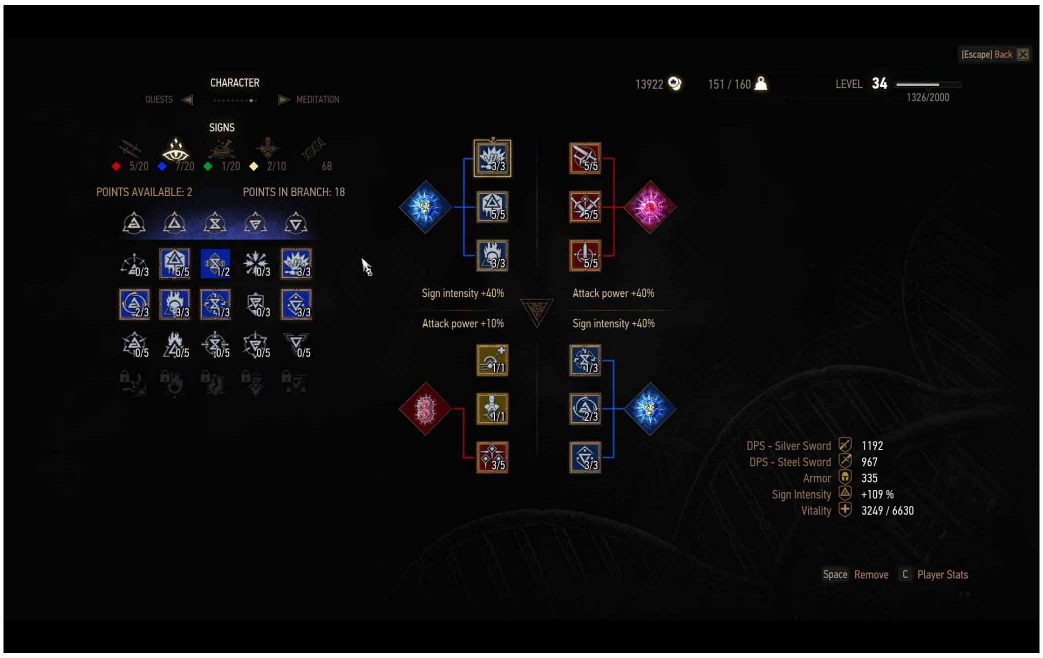


Figure 5.2: A screen capture from *The Witcher 3: Wild Hunt*, which features a slot-based system for activating technologies and mutating them.

game. Moreover, they are grouped in order to allow enhancing the technology effects by mutagens, as can be seen in Figure 5.2. The slot system is an interesting invention. However, the technology development itself works quite conventionally.

It remains to be seen, if genuinely new approaches to deal with development within games, able to challenge TTs as the predominant development option mechanism, will emerge. For now, however, the prototype implementation model for TTs seems adequate and capable of coping with the challenges encountered in the contemporary games. It can also be easily modified, should a need arise.

Chapter 6

Conclusion

"As long as there are games to play it is not over."
– Alex Ferguson [f]

The introductory part of this thesis is concluded in this chapter. To shortly summarize the essential content of the introductory part, it suffices to state that creating TTs for digital games as independent functional entities, separate from their users, and using scripting languages for such TT implementations were discussed and generally found to be a beneficial approach.

The research work that was conducted contributed both scientifically and technologically. During the process, several academic publications on TTs and implementing them were published. Prototype software artifacts were implemented to test the proposed methods.

Section 6.1 revisits the actual research questions and crystallizes the answers found. After that, Section 6.2 discusses possibilities for continuation of the work and illuminates future plans.

[f] `http://www.brainyquote.com/quotes/quotes/a/alexfergus424730.html`, accessed in May 2015.

6.1 Answers to the Research Questions

This section states the central results found in the process of preparing this thesis and conducting the related research work. This is done with respect to the research questions introduced in Section 1.3.

The answers themselves are basically simple and could be given quite concisely. However, as we are not living in a black and white world, some extra thoughts clarifying the situation are also presented, when deemed appropriate.

RQ1: How could the current practices of digital game development (especially from the viewpoint of implementing TTs) be improved?

The typical digital game development process could be improved in several ways. The most – and quite – fundamental change proposed was unifying the habits and formats of important game developers used to implement TTs. A central principle of the TT creation approach presented is also to make the unified format easily modifiable, and the specifications and tools inexpensive and available. The same principles could also be applied in other areas of game development.

The obvious benefits would be numerous. For instance, generally usable development and modding tools could be created, which might lead to higher quality than is common concerning game specific tools. It would become easier to work with several different games, and costs for acquiring development tools would be reduced.

In practice, of course, getting the developers to use the common formats may be very hard (and if these formats are to be easily modifiable, even more so). Cooperation in this matter might, however, be a huge benefit for the participants, if it led to a format suitable for their needs. Of course, the format (and tools) could be created by anyone, but marketing it and getting users and thus "momentum" enough would be challenging.

Another challenge after creating a format with different developers as users would be keeping the users and the format together. Branching to different versions should be, if not totally prevented, at least limited. If all of the companies derived their own (mutually incompatible) versions, the situation would be – at least to a certain degree – returning to what it used to be before. Instead of branching, the main format should be developed and versioned further as needed. Backward compatibility ought to be maintained, nevertheless.

Besides the format unification itself, another – although strongly related – change that can be suggested is making more use of automation opportunities offered by development tools. Specifically, automatic quality monitoring already during the early stages of a development process can save the valuable testing time later for rectifying nontrivial problems.

RQ2: What kind of imperfections and problems are typical with TTs?

There are many different kinds of problems, and Publication V lists many common reasons for complaints. These have been found especially interesting during the research process:

- TTs have typically a rigid structure fixed beforehand without any temporal variations. Even though this is not always a problem – many fixed TTs are suitable for their purposes and work well and reliably – the author thinks that games could often be improved by introducing some temporal variation in their TTs.

- TTs may be poorly balanced temporally or resource-wise. TTs have many roles to play. One of them is to guide and restrict ("gate") the game flow in order to create a meaningful and pleasant gaming experience. If a TT works badly regarding this aspect, the players may feel either being held back artificially or achieving goals too easily. Both cases can lead to negative feelings and frustration.

- TTs may offer too obvious efficient technology development routes to follow. This diminishes the role of the other parts of the TTs – or even makes them irrelevant.

Although the quality of a TT is largely a subjective matter, there are properties that can be generally declared as defects. The listed ones are good examples of these (with the reservation concerning the rigidity).

RQ3: Can automation help with producing better TTs?

The answer of the author is affirmative. Of course, the “goodness” of a TT always depends on many things, and not least on the personal opinions of its users. However, some of the “clear” problems (that can be defined as problems without too many objections – see the answer to RQ2) can be found by measuring TT properties programmatically. Moreover, TTs can be adjusted in order to make them better – at least in the light of the measurements.

Publication V introduces some TT measures that can be applied in order to estimate TT quality and to spot possible problems. As far as computers need human support in improving TTs, different property visualizations are of tremendous value, and suitable software tools can be used to offer them.

It should be taken into account that both automated and manual modifications aiming to fix certain aspects of a TT may always cause other kinds of problems. Therefore, attention should be paid to the risks, and results of corrective steps taken should always be evaluated from several points of view.

RQ4: Can automation help with reducing human workload in TT construction?

It has become clear that many possibilities to save human efforts needed for TT construction are available by the means of using software automation. Especially two of them have been covered in this thesis:

- Automated code generation to create TT structures and their internal functionality can reduce programming work considerably, when compared with ad hoc approaches or reusable but non-automated methods.

- Automated measuring and problem-spotting features in software tools can reduce the need to iterate during the testing and modifying phases of the development, if defects can be found and corrected early in the process because of these features.

Moreover, if improving TTs based on the measurements can also be automated, or at least possible corrections can be suggested automatically by the development tools, the workload is once again reduced. It is obvious that many kinds of automated corrections (or suggestions) can be made relatively easily.

6.2 The Future

Answers for many questions and problems were found during the research process, but several were also still left to be found, and on the other hand, a wide array of new ones emerged. The prototype tools and algorithms created can be improved, and more tests can be carried out. Particularly important would be to get the approach tested with real game development cases, although the applicability seems unquestionable based on the tests conducted so far.

If a common format (and a common tool set) used by several real-world game developers could be established, the effects on the games and game development would be interesting to see. In order to speed up achieving this goal – or at least to promote the idea further – the prototype tools developed for this thesis might be released as an open source project, when they are mature enough. (This, of course, requires rights to do so, and at the moment the rights belong to Tampere University of Technology.) Although the GUIs are designed to be easy to use, some improvements, additions and feature changes should be made prior to any such release. One should also document the tools and the TT implementation format and its user interface thoroughly, because poor documentation may lead to “over-large learning humps” [12] and prevent wide adoption.

TT feature measurement methods should be developed further. One should persistently keep developing and testing new heuristics and means to evaluate and fix TTs. No doubt, there are still meaningful measures to be found and measuring algorithms to be developed. This thesis has only been able to scratch the surface, and what lies beneath it is interesting. The goal in the future work should be to find a set of properties, complete enough to cover the "goodness" of a TT well, and to create algorithms and implementations to measure these properties efficiently. The possible dependencies and relations of the measures should be understood well.

The next step would be to add new techniques into the arsenal of automated adjustment methods. Some of the methods might be trivial tree manipulations targeting tuning particular measurable properties, but other possibilities should be considered, too.

Besides enhancing TTs automatically when creating them, the process could continue during the actual gameplay. Therefore, also runtime functionality (to be standardized) of TTs should be improved. As mentioned in Section 4.3, a TT could contain self-modifying runtime methods. They could improve the TT based on, for example, learning player preferences and observing, how the game is played.

Real-time adaptation – for instance, adjusting game difficulty – based on observing the player during gameplay has already been applied in games [1] and studied [116]. Possibilities to adapt to a player also include generating customized content before the game is played [62]. Until now, adaptations to a player have been applied to several targets including game mechanics, AIs, NPCs, narratives, game scenarios, and quests [62]. The topic of extending customized content generation and in-game adaptations to TTs deserves a further study.

One must, of course, also keep observing closely the trends characterizing the actual games that will be published, and react to possible fundamental changes prudently. However, when writing this, it seems very unlikely that TTs would disappear in the near future. Although they have often been critisized for different reasons, no better solutions have been presented to replace them. Offering players possibilities to choose among different development options (and such) will most likely remain as a central mechanism in digital games; the history shows that people find development to be an interesting aspect in gaming, and computerization has made it possible to use it in the ways that are impractical in traditional games.

To summarize: developing the methods discussed in this thesis should be continued in order to make them appeal to the industry. Developing also the prototype format for implementing TTs and the software tools created during the process further is a natural side-effect. Although the ultimate benefits of the approach can only be achieved, if several developers can agree on implementing their TTs using a common format, which may still seem a distant goal, this should not discourage one from advancing towards this goal and continuing research work with perseverance. The author hopes that this thesis encourages also others to study TTs and ways to make them, digital games, and the world better.

Bibliography

[1] E. Adams. *Fundamentals of Game Design.* New Riders Publishing, second edition, 2009.

[2] D. W. Aha and M. Molineaux. Integrating learning in interactive gaming simulators. In *Challenges of Game AI: Proceedings of the AAAI'04 Workshop (Technical Report WS-04-04)*, San Jose, California, USA, 2004. AAAI Press.

[3] R. Al-Azawi, A. Ayesh, I. Kenny, and K. A. Al-Masruri. Towards an AOSE: Game development methodology. In S. Omatu, J. Neves, J. M. Corchado Rodríguez, J. F. De Paz Santana, and S. Rodríguez González, editors, *Proceedings of the 10th International Conference on Distributed Computing and Artificial Intelligence (DCAI 2013)*, volume 217 of *Advances in Intelligent Systems and Computing*, pages 493–501, Salamanca, Spain, May 2013. Springer.

[4] A. Amory, K. Naicker, J. Vincent, and C. Adams. The use of computer games as an educational tool: identification of appropriate game types and game elements. *British Journal of Educational Technology*, 30(4):311–321, 1999.

[5] E. F. Anderson and C. E. Peters. No more reinventing the virtual wheel: Middleware for use in computer games and interactive computer graphics education. In *The 31st Annual Conference of the European Association for Computer Graphics (Eurographics 2010) – Education Papers*, pages 33–40, Norrköping, Sweden, May 2010.

[6] Y. Aoyama and H. Izushi. Hardware gimmick or cultural innovation? Technological, cultural, and social foundations of the Japanese video game industry. *Research policy*, 32(3):423–444, 2003.

[7] A. BinSubaih and S. Maddock. G-factor portability in game development using game engines. In *Proceedings of the 3rd International Conference on Games Research and Development (CyberGames 2007)*, pages 163–170, Manchester Metropolitan University, UK, Sept. 2007.

[8] A. BinSubaih, S. Maddock, and D. Romano. A survey of 'game' portability. *University of Sheffield, Technical Report CS-07-05*, 2007.

[9] S. Björk and J. Holopainen. *Patterns in Game Design.* Game Development Series. Charles River Media, Inc., 2004.

[10] J. Blow. Game development: Harder than you think. *Queue*, 1(10):28–37, Feb. 2004.

[11] D. M. Bourg and G. Seemann. *AI for Game Developers.* O'Reilly Media, Inc., 2004.

[12] P. J. Brown. Tools for amateurs. In D. Néel, editor, *Tools & Notions for Program Construction: An Advanced Course*, pages 377–390. Cambridge University Press, 1982.

[13] J. Bryce and J. Rutter. Killing like a girl: Gendered gaming and girl gamers' visibility. In F. Mäyrä, editor, *Proceedings of the Computer Games and Digital Cultures Conference*, pages 243–256, Tampere, Finland, June 2002. Tampere University Press.

[14] M. Buckland. *Programming Game AI by Example.* Wordware Publishing, Inc., 2005.

[15] M. Buro. Real-time strategy games: A new AI research challenge. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1534–1535, Acapulco, Mexico, Aug. 2003. Morgan Kaufmann.

[16] R. Caillois. *Man, Play and Games.* University of Illinois Press, 1961. Translation of *Les jeux et les hommes*, 1958, by Meyer Barash.

[17] Civilization Development Team. private communication, Mar. 2015.

[18] T. M. Connolly, E. A. Boyle, E. MacArthur, T. Hainey, and J. M. Boyle. A systematic literature review of empirical evidence on computer games and serious games. *Computers & Education*, 59(2):661–686, 2012.

[19] T. A. DeFanti, M. D. Brown, and B. H. McCormick. Visualization: expanding scientific and engineering research opportunities. *Computer*, 22(8):12–25, 1989.

[20] E. D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In J. Sgall, A. Pultr, and P. Kolman, editors, *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science 2001 (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 18–33, Mariánské Lázně, Czech Republic, Aug. 2001. Springer.

[21] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. From game design elements to gamefulness: Defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments (Academic MindTrek 2011)*, pages 9–15, Tampere, Finland, Sept. 2011. ACM.

[22] A. Dixit, S. Skeath, and D. H. Reiley, Jr. *Games of Strategy.* W. W. Norton & Company, Inc., third edition, 2009.

[23] C. Dragert, J. Kienzle, and C. Verbrugge. Reusable components for artificial intelligence in computer games. In *Proceedings of the 2nd International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques (GAS 2012)*, pages 35–41, Zürich, Switzerland, June 2012. IEEE.

[24] E. Dybsand. Goal-directed behavior using composite tasks. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 237–245. Charles River Media, Inc., 2003.

[25] J. Dyck, D. Pinelle, B. Brown, and C. Gutwin. Learning from games: HCI design innovations in entertainment software. In *Proceedings of Graphics Interface 2003 (GI 2003)*, pages 237–246, Halifax, Nova Scotia, June 2003. A K Peters, Ltd.

[26] M. S. El-Nasr and B. K. Smith. Learning through game modding. *Computers in Entertainment*, 4(1):3B:1–3B:20, Jan. 2006.

[27] Entertainment Software Association. 2014 Essential facts about the computer and video game industry. `http://www.theesa.com/facts/pdfs/esa_ef_2014.pdf`, accessed May 16, 2014.

[28] M. Eskelinen. The gaming situation. *Game Studies*, 1(1), 2001.

[29] M. Eskelinen. *Cybertext Poetics: The Critical Landscape of New Media Literary Theory.* Bloomsbury Publishing, 2012.

[30] C. J. Ferguson. Blazing angels or resident evil? Can violent video games be a force for good? *Review of General Psychology*, 14(2):68–81, 2010.

[31] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 259–264, Chicago, Illinois, USA, July 2008. AAAI Press.

[32] S. M. Fisch. Making educational computer games ”educational”. In *Proceedings of the 2005 Conference on Interaction Design and Children (IDC 2005)*, pages 56–61, Boulder, Colorado, USA, June 2005. ACM.

[33] G. Frasca. Ludologists love stories, too: Notes from a debate that never took place. In M. Copier and J. Raessens, editors, *Proceedings of the Digital Games Research Conference 2003 (The 2003 DiGRA International Conference: Level Up)*, pages 92–99, Utrecht, The Netherlands, Nov. 2003. Faculty of Arts, Utrecht University.

[34] D. Fu and R. Houlette. Putting AI in entertainment: An AI authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, July 2002.

[35] D. Fu and R. Houlette. The ultimate guide to FSMs in games. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 283–302. Charles River Media, Inc., 2003.

[36] T. Fullerton. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann Publishers, Inc., second edition, 2008.

[37] W. O. Galitz. *The Essential Guide to User Interface Design.* John Wiley & Sons, 2007.

[38] A. Gazzard. Unlocking the gameworld: The rewards of space and time in videogames. *Game Studies*, 11(1), Feb. 2011.

[39] T. Ghys. Technology trees: Freedom and determinism in historical strategy games. *Game Studies*, 12(1), 2012.

[40] M. E. Gredler. Games and simulations and their relationships to learning. In D. H. Jonassen, editor, *Handbook of Research on Educational Communications and Technology*, pages 571–582. Lawrence Erlbaum Associates, second edition, 2004.

[41] M. Griffiths. Violent video games and aggression: A review of the literature. *Aggression and Violent Behavior*, 4(2):203–212, 1999.

[42] F. Groh. Gamification: State of the art definition and utilization. In N. Asaj, B. Könings, M. Poguntke, F. Schaub, B. Wiedersheim, and M. Weber, editors, *Proceedings of the 4th Seminar on Research Trends in Media Informatics (RTMI 2012)*, pages 39–46, Ulm, Germany, Feb. 2012. Institute of Media Informatics, Ulm University.

[43] E. A. A. Gunn, B. G. W. Craenen, and E. Hart. A taxonomy of video games and AI. In *Proceedings of the AI and Games Symposium at the*

AISB 2009 Convention, pages 4–14, Edinburgh, Scotland, Apr. 2009. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour (SSAISB).

[44] N. Hallford and J. Hallford. *Swords & Circuitry: A Designer's Guide to Computer Role-Playing Games.* PRIMA TECH's Game Development Series. Prima Publishing, 2001.

[45] J. Hamari, J. Koivisto, and H. Sarsa. Does gamification work? — A literature review of empirical studies on gamification. In R. H. Sprague Jr., editor, *Proceedings of the 47th Annual Hawaii International Conference on System Sciences (HICSS-47)*, pages 3025–3034, Hawaii, USA, Jan 2014. IEEE.

[46] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, Mar. 2004.

[47] J. Huizinga. *Homo Ludens – A Study of the Play-Element in Culture.* Beacon Press, 1971.

[48] J. Iivari. A paradigmatic analysis of contemporary schools of IS development. *European Journal of Information Systems*, 1(4):249–272, 1991.

[49] J. Iivari. A paradigmatic analysis of information systems as a design science. *Scandinavian Journal of Information Systems*, 19(2):39–64, 2007.

[50] Internet Advertising Bureau UK. More women now play video games than men. `http://www.iabuk.net/about/press/archive/more-women-now-play-video-games-than-men`, 2014, accessed Mar. 5, 2015.

[51] P. H. Järvinen. Research questions guiding selection of an appropriate research method. In H. R. Hansen, M. Bichler, and H. Mahrer, editors, *Proceedings of the 8th European Conference on Information Systems, Trends in Information and Communication Systems for the*

21st Century (ECIS 2000), pages 124–131, Vienna, Austria, July 2000. Wirtschaftsunivsitat Wien.

[52] U. Johansson and L. Niklasson. Why settle for optimal play when you can do better. In C. Tse Ning, editor, *Proceedings of the International Conference on Application and Development of Computer Games in the 21st Century (ADCOG 2001)*, pages 130–136, Hong Kong, China, Nov. 2001. City University of Hong Kong.

[53] S. Johnson. The unique challenges of turn-based AI. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 399–403. Charles River Media, Inc., 2003.

[54] H. W. Kennedy. Lara Croft: Feminist icon or cyberbimbo? On the limits of textual analysis. *Game Studies*, 2(2), 2002.

[55] D. L. King, M. Gradisar, A. Drummond, N. Lovato, J. Wessel, G. Micic, P. Douglas, and P. Delfabbro. The impact of prolonged violent video-gaming on adolescent sleep: an experimental study. *Journal of Sleep Research*, 22(2):137–143, 2013.

[56] D. L. King, M. C. Haagsma, P. H. Delfabbro, M. Gradisar, and M. D. Griffiths. Toward a consensus definition of pathological video-gaming: A systematic review of psychometric assessment tools. *Clinical Psychology Review*, 33(3):331–342, 2013.

[57] J. E. Laird and M. van Lent. Human-level AI's killer application: Interactive computer games. *AI Magazine*, 22(2):15–25, 2001.

[58] K. Lano. *Formal Object-Oriented Development*. Springer-Verlag New York, Inc., 1995.

[59] G. W. Lecky-Thompson. *AI and Artificial Life in Video Games*. Course Technology (Cengage Learning), 2008.

[60] L. Lidén. Artificial stupidity: The art of intentional mistakes. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 41–48. Charles River Media, Inc., 2003.

[61] S. Loh and S. H. Soon. Comparing computer and traditional games using game design patterns. In *Proceedings of the 2006 International Conference on Game Research and Development (CyberGames 2006)*, pages 237–241, Perth, Australia, Dec. 2006. Murdoch University.

[62] R. Lopes and R. Bidarra. Adaptivity challenges in games and simulations: A survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):85–99, June 2011.

[63] Lua. `http://djinni.wikia.com/wiki/Category:LUA`, accessed Dec. 15, 2014.

[64] J. Ludwig and A. Farley. Examining extended dynamic scripting in a tactical game framework. In C. Darken and G. M. Youngblood, editors, *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2009)*, pages 76–81, Stanford, California, USA, Oct. 2009. AAAI Press.

[65] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, Dec. 1995.

[66] M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generating scripting code for computer role-playing games. In *Proceedings of the 19th International Conference on Automated Software Engineering (ASE 2004)*, pages 386–387, Linz, Austria, Sept. 2004. IEEE Computer Society.

[67] R. Miikkulainen, B. D. Bryant, R. Cornelius, I. V. Karpov, K. O. Stanley, and C. H. Yong. Computational intelligence in games. In G. Y. Yen and D. B. Fogel, editors, *Computational Intelligence: Principles and Practice*, pages 155–191. IEEE Computational Intelligence Society, 2006.

[68] I. Millington. *Artificial Intelligence for Games*. The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers, Inc., 2006.

[69] M. Molineaux and D. W. Aha. TIELT: A testbed for gaming environments. In M. Veloso and S. Kambhampati, editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1690–1691, Pittsburgh, Pennsylvania, USA, July 2005. AAAI Press.

[70] D. Morris and L. Hartas. *Strategy Games*. The Ilex Press Limited, 2004.

[71] A. Nareyek. Review: Intelligent agents for computer games. In T. A. Marsland and I. Frank, editors, *Revised Papers from the Second International Conference on Computers and Games (CG 2000)*, volume 2063 of *Lecture Notes in Computer Science*, pages 414–422, Hamamatsu, Japan, Oct. 2000. Springer-Verlag, 2002.

[72] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[73] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[74] Newzoo. Global games market will reach $102.9 billion in 2017. `http://www.newzoo.com/insights/global-games-market-will-reach-102-9-billion-2017-2/`, accessed May 16, 2014.

[75] L. Nooney. A pedestal, a table, a love letter: Archaeologies of gender in videogame history. *Game Studies*, 13(2), 2013.

[76] T. Nummenmaa, E. Berki, and T. Mikkonen. Exploring games as formal models. In *Proceedings of the 2009 Fourth South-East European Workshop on Formal Methods (SEEFM 2009)*, pages 60–65, Thessaloniki, Greece, Dec. 2009. IEEE Computer Society.

[77] S. Ocio and J. A. Lopez Brugos. Multi-agent systems and sandbox games. In *Proceedings of the AI and Games Symposium at the AISB 2009 Convention*, pages 70–74, Edinburgh, Scotland, Apr. 2009. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour (SSAISB).

[78] J. Orkin. Applying goal-oriented action planning to games. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 217–227. Charles River Media, Inc., 2003.

[79] PC Gamer. Age of Empires Online interview: Tech trees and civ customization. `http://www.pcgamer.com/2012/05/21/age-of-empires-online-interview-tech-trees-and-civ-customization/`, 2012, accessed July 19, 2012.

[80] M. Picard. The foundation of *geemu*: A brief history of early Japanese video games. *Game Studies*, 13(2), 2013.

[81] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. W. Aha. Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75–84, 2006.

[82] M. J. V. Ponsen, S. Lee-Urban, H. Muñoz-Avila, D. W. Aha, and M. Molineaux. Stratagus: An open-source game engine for research in real-time strategy games. In D. W. Aha, H. Muñoz-Avila, and M. van Lent, editors, *Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games (Technical Report AIC-05-127)*, pages 78–83, Edinburgh, Scotland, July 2005. Navy Center for Applied Research in Artificial Intelligence (NCARAI).

[83] H. Postigo. Of mods and modders: Chasing down the value of fan-based digital game modifications. *Games and Culture*, 2(4):300–313, 2007.

[84] M. Pritchard. Ensemble Studio's Age of Empires II: The Age of Kings. In A. Grossman, editor, *Postmortems from Game Developer*, pages 115–126. CMP Books, CMP Media LLC, 2003.

[85] P. Quax, B. Cornelissen, J. Dierckx, G. Vansichem, and W. Lamotte. ALVIC-NG: State management and immersive communication for massively multiplayer online games and communities. *Multimedia Tools and Applications*, 45(1–3):109–131, Oct. 2009.

[86] P. Rego, P. M. Moreira, and L. P. Reis. Serious games for rehabilitation: A survey and a classification towards a taxonomy. In *Proceedings of the 2010 5th Iberian Conference on Information Systems and Technologies (CISTI 2010)*, pages 47:1–47:6, Santiago de Compostela, Spain, June 2010. IEEE.

[87] R. Rouse III. *Game Design: Theory and Practice*. Wordware Publishing, Inc., second edition, 2004.

[88] F. Safadi, R. Fonteneau, and D. Ernst. Artificial intelligence in video games: Towards a unified framework. *International Journal of Computer Games Technology*, 2015: article ID 271296, 30 pages, 2015.

[89] W. Scacchi. Computer game mods, modders, modding, and the mod scene. *First Monday*, 15(5), 2010.

[90] E. Schaefer. Blizzard Entertainment's Diablo II. In A. Grossman, editor, *Postmortems from Game Developer*, pages 79–90. CMP Books, CMP Media LLC, 2003.

[91] J. Schell. *The Art of Game Design: A Book of Lenses*. CRC Press, 2008.

[92] D. Scimera. The gender inequality in core gaming is worse than you think. `http://venturebeat.com/2013/09/19/gender-inequality/`, 2013, accessed Aug. 12, 2014.

[93] M. K. Sein, O. Henfridsson, S. Purao, M. Rossi, and R. Lindgren. Action design research. *MIS Quarterly*, 35(1):37–56, Mar. 2011.

[94] P. Sequeira, M. Vala, and A. Paiva. "What can I do with this?" – Finding possible interactions between characters and objects. In E. H. Durfee, M. Yokoo, M. N. Huhns, and O. Shehory, editors, *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, pages 5:1–5:7, Honolulu, Hawaii, USA, May 2007. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS).

[95] J. L. Sherry. The effects of violent video games on aggression. *Human Communication Research*, 27(3):409–431, 2001.

[96] D. I. K. Sjøberg, T. Dybå, and M. Jørgensen. The future of empirical methods in software engineering research. In L. C. Briand and A. L. Wolf, editors, *Proceedings of the Workshop on the Future of Software Engineering (FoSE 2007) at the 29th International Conference on Software Engineering (ICSE 2007)*, pages 358–378, Minneapolis, Minnesota, USA, 2007. IEEE Computer Society.

[97] J. Smed and H. Hakonen. Synthetic players: A quest for artificial intelligence in computer games. *Human IT*, 7(3):57–77, 2005.

[98] B. Smith. Poptop Software's Tropico. In A. Grossman, editor, *Postmortems from Game Developer*, pages 137–146. CMP Books, CMP Media LLC, 2003.

[99] O. Sotamaa. When the game is not enough: Motivations and practices among computer game modding culture. *Games and Culture*, 5(3):239–255, July 2010.

[100] S. de Sousa Borges, V. H. S. Durelli, H. M. Reis, and S. Isotani. A systematic mapping on gamification applied to education. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*, pages 216–222, Gyeongju, Republic of Korea, Mar. 2014. ACM.

[101] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. Adaptive game AI with dynamic scripting. *Machine Learning*, 63:217–248, 2006.

[102] K. Squire and H. Jenkins. Harnessing the power of games in education. *InSight*, 3(1):5–33, 2003.

[103] C. Steinkuehler and M. Chmiel. Fostering scientific habits of mind in the context of online play. In *Proceedings of the 7th International*

Conference on Learning Sciences (ICLS 2006), pages 723–729, Bloomington, Indiana, USA, June/July 2006. International Society of the Learning Sciences (ISLS).

[104] T. Tahkokallio. Designers' notes #9: Technology tree. `http://boardgamegeek.com/thread/650731/designers-notes-9-technology-tree/`, 2011, accessed July 19, 2012.

[105] J. Togelius, N. Shaker, and M. J. Nelson. Introduction. In N. Shaker, J. Togelius, and M. J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 1–15. Springer, 2014.

[106] P. Tozour. The perils of AI scripting. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 541–547. Charles River Media, Inc., 2002.

[107] A. Unwin and H. Hofmann. GUI and command-line – Conflict or synergy? In M. Pourahmadi and K. Berk, editors, *Proceedings of the 31st Symposium on the Interface of Computing Science and Statistics: Models, Predictions, and Computing*, pages 246–253, Schaumburg, Illinois, USA, June 1999. Interface Foundation of North America (Interface).

[108] J.-M. Vanhatupa. *Tool Support for Computer Role-Playing Game Programming: Foundations, Guidelines and Applications.* DSc thesis, Tampere University of Technology, Tampere, Finland, 2014.

[109] J.-M. Vanhatupa and T. J. Heinimäki. Scriptable artificial intelligent game engine for game programming courses. In C. Hermann, T. Lauere, T. Ottmann, and M. Welte, editors, *Proceedings of Informatics Education Europe IV (IEE IV 2009)*, pages 27–31, Freiburg, Germany, Nov. 2009.

[110] M. Virvou, G. Katsionis, and K. Manos. Combining software games with education: Evaluation of its educational effectiveness. *Educational Technology & Society*, 8(2):54–65, 2005.

[111] B. G. Weber, M. Mateas, and A. Jhala. Building human-level AI for real-time strategy games. In *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, San Francisco, California, USA, 2011. AAAI Press.

[112] W. White, C. Koch, J. Gehrke, and A. Demers. Better scripts, better games. *Communications of the ACM*, 52(3):42–47, Mar. 2009.

[113] H. E. Wirman. *Playing The Sims 2 – Constructing and negotiating woman computer game player identities through the practice of skinning.* PhD thesis, University of the West of England, Bristol, UK, 2011.

[114] M. J. P. Wolf. Genre and the video game. In M. J. P. Wolf, editor, *The Medium of the Video Game*, pages 113–136. University of Texas Press, 2002. Cited from `http://www.robinlionheart.com/gamedev/genres.xhtml`.

[115] R. Wyckoff. DreamWorks Interactive's Trespasser. In A. Grossman, editor, *Postmortems from Game Developer*, pages 183–194. CMP Books, CMP Media LLC, 2003.

[116] G. N. Yannakakis and J. Hallam. Real-time game adaptation for optimizing player satisfaction. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(2):121–133, June 2009.

[117] J. P. Zagal, M. Nussbaum, and R. Rosas. A model to support the design of multiplayer games. *Presence: Teleoperators and Virtual Environments*, 9(5):448–462, Oct. 2000.

[118] A. Zook and M. O. Riedl. Automatic game design via mechanic generation. In C. E. Brodley and P. Stone, editors, *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 530–537, Québec City, Québec, Canada, July 2014. AAAI Press.

Publication I

T. J. Heinimäki. Technology Trees in Digital Gaming. In *Proceedings of the 16th International Academic MindTrek Conference 2012 (AMT2012)*, pages 27–34. Tampere, Finland, October 2012.

The author is fully aware of the fact that *Dungeons & Dragons* (Gary Gygax and Dave Arneson 1974) (D&D) is a role-playing game, and, as "the first one", an especially important representant of the genre. However, this paper uses the same, rather broad definition of board games that is used throughout the thesis, so in this publication D&D is mentioned as a board game, even though any board is not necessarily used when playing. Using such inaccurately describing genre naming conventions is actually rather common; some prefer referring to D&D-like games as pen and paper role-playing games or tabletop role-playing games, but it is possible to play also without pens and papers or even without a table ...

Technology Trees in Digital Gaming

Teemu J. Heinimäki
Department of Software Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
teemu.heinimaki@tut.fi

ABSTRACT

Technology trees have been used in the digital game industry for over two decades. They are in very common use at least in strategy and role-playing games. However, the academic interest towards the matter seems to be almost nonexistent. Although the basic concept may be rather simple and well-understood as far as the industrial points of view are considered, we believe that it would be beneficial to study the issue also academically. In this survey we discuss using technology trees in digital games and present classification criteria for facilitating further studies.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and Networks; E.1 [**Data Structures**]: Trees; H.5.0 [**Information Systems**]: Information Interfaces and Presentation—*general*; H.5.2 [**Information Systems**]: Information Interfaces and Presentation—*user interfaces*; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*games*; I.2.1 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; I.6.5 [**Simulation and Modeling**]: Model Development; I.6.3 [**Simulation and Modeling**]: Applications; J.0 [**Computer Applications**]: General; K.2 [**Computing Milieux**]: History of Computing—*software*; K.4.0 [**Computing Milieux**]: Computers and Society—*general*

General Terms

Design, Human Factors

Keywords

Digital gaming, gaming, technology trees, techtrees

1. INTRODUCTION

Recently, the game industry has become an extremely important branch of industry. The computer and video game markets are huge, as is the amount of money involved; the U.S. sales grew rapidly from \$2.6 billion in 1996 to \$10.5 billion in 2009 [10]. Hence, several aspects of digital gaming have been studied rigorously. However, some interesting and widely used issues have virtually been omitted. There are some publications about optimizing decision-making policies in strategy games, but as far as we are aware of, the fundamental structures defining the development possibilities in many games – different kinds of progression graphs – have not been studied much. This makes it hard to give meaningful references to related academical work – and thus to carry out research work on the matter.

Technology tree, also known as techtree or tech-tree, is a traditional name for a progression graph defining or depicting the relations and dependencies between *technologies*. Technology trees are often fundamental parts of modern computer games. This is especially the case, when considering games of the genre "explore, expand, exploit, and exterminate" (4X). There are technology trees in most strategy games [17, 28]. They are used for presenting the hierarchical structures of development possibilities of different technologies and keeping track of the achieved – for instance, scientific – development levels. Nowadays, also other game genres use similar progression graph structures, which also can be seen as technology trees with liberal enough definition of the word "technology".

The aim of this paper is to consider different technology tree semantics and usages found in actual games. We present observations on subtypes and possible classification criteria for technology trees. Besides, the overall purpose of this paper is to promote and encourage further research work on the topic.

This paper proceeds as follows. After the situation concerning the related work is illuminated briefly in Section 2, our views and basics of technology trees are discussed further and some terminology used is explained in Section 3. Then, a short overview of the history of using technology trees in digital games is given in Section 4. After that, Section 5 proposes possible technology tree classification criteria. Notions of them, based on an example game selection, are finally presented in Section 6 before concluding the paper with Section 7.

2. RELATED WORK

There is a wide array of literature available considering digital games. Different textbooks cover various topics like game design (see, e.g., [3], [18], or [1]) or artificial intelligence (AI) in games (see, e.g., [13], [7], or [6]). Some of these books mention the existence of technology trees, but usually only with couple of words.

Nowadays there are also a lot of academic game publications. They, also, cover many important topics from finding routes to creating AIs for playing a given game. However, we are aware of only one peer reviewed publication [12] focusing on technology trees so far. In that paper, ideas for analyzing technology trees and measuring their properties are discussed.

Besides that, the most closely related peer reviewed publications seem to be those considering decision-making processes and strategies in real-time strategy (RTS) game settings – settings including technology trees in some form. Also making decisions about constructing buildings – which buildings should be built and when – have been included in this kind of studies. For instance, dynamic scripting has been used for this [23, 24]. However, the relevance of these publications for this paper is somewhat limited; while some kind of a simple technology tree is often used in these kind of studies for fixing the action possibilities, the research focus is typically in completely other matters, and trees are only used as given.

Different AI frameworks and game engines have been implemented and competitions have been organized for improving the quality of digital game AI in general. See, for instance, [8]. However, the main focus aspects seem to differ from the research or technology development process.

3. SEMANTICS AND BACKGROUND

Often the term technology tree refers only to the graphical representation of a hierarchy of technologies. In this paper, however, the term is also used, when referring to the corresponding graph-like information structure or to the abstract idea of the hierarchy with some semantics attached. The meaning should be clear based on the context.

Despite the name, technology trees are not necessarily proper tree structures, but resemble usually directed acyclic graphs[1] (DAGs). Often nodes of different branches have common children. This is the case, when a technology has prerequisite technologies (parents) from different branches. In technology trees there can also be several possibilities for the first technology to be developed[2].

A part of an example technology tree is depicted in Figure 1. There are two possible initial technologies, “Stick” and “Stone Tools”, and the directed edges (dependencies between technologies) are represented as arrows. One may develop

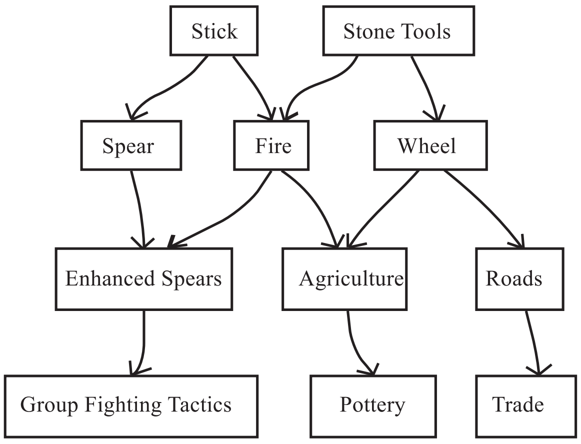


Figure 1: A simple (partial) technology tree.

technologies only after having developed corresponding prerequisite technologies, i.e., “Agriculture” can be achieved only after having knowledge of “Fire” and “Wheel”, and so on.

Technology trees are usually basically dependency graphs. The general requirement for acyclicity is easy to derive for the common cases, in which all the dependencies must be satisfied in order to develop a technology; if there were cycles, they could not be entered. (Cycles can, of course, be allowed, if all the edges of a graph are not interpreted as prerequisite relationships, or if satisfying only some of the prerequisites is enough for developing a technology. However, cyclic structures are not conventional.)

Another misleading convention is to refer to a collection consisting of several separate internally connected DAGs (or other “trees”) as a single technology tree. It would be more clear to use the term technology tree only, when referring to a connected digraph (used to represent development possibility relations with some needed data associated to the nodes and edges). Collections of such separate “trees” could be referred as technology forests, but in this paper we stick with the convention and use the term technology tree. We allow technology trees be arbitrary directed graphs with semantics and possible additional data attached to them. Thus, they may be composed of separate subtrees, not connected to each other. (On the other hand, while such subtrees also fulfill our definition for technology trees, the term does not need to mean always the whole subtree collection, but may refer to a single subtree as well.) The short form “tree” should also be understood to refer to such a technology tree for the rest of the paper.

So-called technologies appear in many flavors. It is hard to characterize, what makes a technology a technology in the structures conventionally referred as technology trees. Moreover, other progression graphs, like “skill trees”, “feat trees”, “perk trees”, or “talent trees” used in character development in computer role-playing games (CRPGs), do not really differ from conventional technology trees. Therefore we choose not to make a distinction between different types of progression graphs; we use the word “technology” in this paper extremely liberally: basically anything achievable by a

[1]Usually there can be at most one edge between two nodes, but directed multigraphs can also be used – with the assumption of edges being individual objects with their own properties. This way, some expressive power can be added to the conventional graph representation – parallel edges can represent, for instance, different processes leading from a technology to another.

[2]It would be tempting to call such nodes roots, but as they are not necessarily roots according to the common definition in graph theory – all the other nodes in a technology tree cannot necessarily be reached via a path starting from this kind of a node – we refrain from calling them roots for avoiding confusion.

research, training, building, or other process, typically with some other technologies as prerequisites, can be seen as a technology. Thus, we use technology trees as a general term covering progression graphs in digital games.[3] For instance the following have been used in real games as technologies in the sense we use the word: Armory, Barracks, Hellfire & Brimstone, Fox Lady, Heaven and Earth, Ninjutsu, Pig Farm, The Five Elements, Way of Chi, Rifling, Replaceable Parts, Flight, Stealth, and Computers [27, 4, 5, 21].

By "developing a technology" we mean any process that aims at achieving the benefits (and disadvantages) of it. Such a process could be, e.g., simply building some advanced building. Usually the "scientific" development process is called research; it is possible, for instance, that virtual researchers contribute producing science points, and when there are enough of them, the technology that is researched can be unlocked. In many 4X games and CRPGs this unlocking mechanism is binary: either a technology has been developed or not. However, this does not have to be the case; technologies can, in some cases, be developed gradually. For instance, in *Space Empires V* (Malfador Machinations 2006), many technologies have several levels [26]. After developing a technology to some level, one may continue researching the same issue in order to obtain higher-level benefits of the same technology. Often different levels of such technologies give the same advantage, with the magnitude depending on the level. For instance, level 1 in "crowd control" could prevent one inhabitant in every city from rising up in mutiny, level 2 could prevent two inhabitants from doing that, and so on.

Of course, for the technology tree system to have any purpose at all, developing technologies should reward the player somehow. In modern RTS games, it is almost always beneficial to traverse the technology tree [2]. New technologies may offer, for instance, access to more advanced units or buildings, government types etc. (For some games it is enough to have building and unit types as technologies.) On the other hand, technologies may also have downsides. Using some advanced technology can, e.g., be expensive, or it can disable the possibility to use some (or all) benefits offered by another technology. Sometimes all the technologies in the tree cannot be developed, but the player may be forced to choose between technologies or branches of the tree to be explored. Thus, developing a technology can close a possibly more suitable path to proceed.

So-called gating is an important concept, when discussing the flow of a game. The term emerged from an assessment (play-testing) program of *Portal* (Valve Software 2007) [20]. The idea was to limit the possibilities of a player to rush forward too fast and thus become frustrated by too hard challenges. Gating can be seen as a wide area of controlling the game flow. There can be many kinds of advance-limiting mechanisms – or "gates". Often technology trees play an important role in this gating process; one cannot advance (at least successfully) in a game without good enough technologies. In addition, developing technologies may open totally new in-game possibilities and make it possible to try and tackle new kinds of challenges.

Technological research work can also be seen as a means of exploration in non-physical domain [17]. If considered as such, it is essential to offer players meaningful and interesting possibilities for conducting technological development; while humans are curious creatures, exploring new frontiers can be intriguing. Thus, a good technology development process – usually meaning good technology trees – can improve dramatically the interestingness of a game.

4. TECHNOLOGY TREES IN DIGITAL GAMES: A GLANCE TO THE PAST

This sections presents a short overview of the history of using technology trees in digital games. For this purpose, some games are mentioned, but many other good examples have been omitted simply for brevity.

Digital games (video games) can be seen as ”natural growth of traditional games into a new medium” [19]. The roots of digital strategy games are in board game domain [17]. Thus, many early digital games featuring technology trees were heavily influenced by board games. While the necessary bookkeeping could be done by the computers instead of human beings, the technology development mechanisms were able to grow more complex than those used in board gaming. Digital games can feature objects having states defined by complex variable sets because of the abilities of programs of tracking the variables. [11].

Technology trees (using our liberal definition) are known to be used with board games (like Dungeons & Dragons (Gary Gygax and Dave Arneson 1974)) in the 1970s, maybe even earlier. The first – at least commonly known – technology tree in the conventional sense appeared in the board game *Civilization* by Francis Tresham in 1980.

In 1991, *Sid Meier's Civilization* (MicroProse 1991) – a computer game inspired, among others, by that board game – was released. It (often called shortly *Civilization* or *Civ*) gained popularity and influenced heavily later digital strategy games. Sometimes this *Civilization* is mentioned as the first digital game using technology trees.

However, technology development had been around in computer gaming domain even before that. As an example, let us mention *Reach for the Stars* (Roger Keating and Ian Trout 1983) – an important game for the birth of the 4X genre. It was published in 1983 and had possibility to develop "ship tech". In 1993, *Master of Orion* (Simtex 1993) was published by MicroProse. A review of it [9] used abbreviation XXXX, from which 4X was derived.

Civilization and *Master of Orion* were early, popular examples of turn-based strategy (TBS) games, in which technological advancement (and technology trees) played a central role. The ideas were soon adopted also to the RTS genre. *Dune II: The Building of a Dynasty* (Westwood Studios 1992) featured building upgrades for developing better

[3]We could, of course, have chosen to use, for instance, term "progression tree" to emphasize the inclusive nature of the term used. However, we wanted to avoid introducing new terminology; technology tree is an established term. It also highlights the presence of some kind of "progression nodes" (even if the name technology may not always be very descriptive), unlike "progression tree".

units (i.e., a technology tree). This scheme was followed by several other representants of the genre, like *Warcraft: Orcs & Humans* (Blizzard Entertainment 1994) and *Total Annihilation* (Cavedog Entertainment 1997). Technology development based on constructing buildings became de facto standard of proceeding in RTS games.

Still other genres adopted the idea of technology trees, most notably CRPGs. In these games, the focus is on the development of individual characters, not in the development of armies or empires. Therefore, the technologies in role-playing game settings typically represent personal abilities, perks, traits, feats or skills – i.e., different personal characteristics. Terms like "feat tree" or "perk tree" are often used, but with our definition for technology, the term technology tree can be used as well. Some examples of CRPGs with technology trees are *Star Wars: Knights of the Old Republic* (BioWare 2003), *Witcher* (CD Projekt RED STUDIO 2007), *Dragon Age: Origins* (BioWare 2009), *Deus Ex: Human Revolution* (Eidos Montreal 2011), and *Skyrim* (Bethesda Softworks 2011).

5. CLASSIFICATION ASPECTS

There are many kinds of technology trees and different ways to use them. Thus, there are also several aspects that could be used for characterizing and classifying technology trees. In following subsections, some of them are discussed.

For identifying these aspects, the usage of technology trees in several games was considered. However, the consideration was not based on any standard game set. In fact, giving any explicit list of games of influence would be misleading, as such a list would almost necessarily be incomplete: the ideas for different aspects are results of observing numerous games over a long time. However, some of these games are listed in Table 1 (see Section 6). The set of games used is definitely biased, e.g., by personal interests and availability. Thus, we do not claim the set of classification aspects to be complete, but rather a set of aspects identified so far.

5.1 Graph Structure and Organizational Semantics

As we use the term technology tree rather liberally, our consideration is not limited to expanding, forest-like technology trees. Although such trees occur often, for instance, in *Civilization*-style strategy games, there are also different structures. For instance, technologies can be organized simply as a set without prerequisite technologies. In other words, it is possible to have a technology tree without edges. Of course, the benefit of thinking technology trees as graph structures is somewhat lost, if there are no edges. *Star Ruler* (Blind Mind Studios 2010) uses a mesh-like "research web" technology collection. Some games use separate – sometimes chain-like – technology trees for different technology categories or research tracks, while some use heavily branching technology trees with intertwining research areas.

In addition to changing the organization and hierarchy between the technologies, one can adjust the semantics of them. Usually edges between technologies are used to control the possibilities to develop successor technologies. There may be exceptions to this rule of thumb, but even in this basic case the semantics of the edges may differ; most importantly, there may be AND prerequisites and OR prerequisites for technologies. By OR prerequisites we mean technologies capable of making it possible to develop a successor technology by themselves, while in the case of AND prerequisites, a high enough level of development (LOD) is required for all of them, if the corresponding successor technology is to be developed. It is not conventional to mix these different kinds of dependencies into a single technology tree, but it is possible. Perhaps dependency type mixing could be used for obtaining more realistic trees; in the real world, some things have several obligatory prerequisites, while others can be achieved with different approaches and backgrounds (i.e., with different prerequisite technologies).

5.2 Rigidity

Usually technology tree structures in games are fixed and invariable. This is a problem, when considering replayability [22]. As the replayability value of a (story oriented) game can be increased by storyline variations based on player behavior [29], similarly variations in technology trees on different times a game is run make the game more interesting and replayable. Having such variable technology trees is, however, extremely rare. Of course, it is "safe" to use only fixed technology trees; play-testing and balancing can then be carried out more reliably.

Although if the technology trees are rigid, they can be different for different game factions. For instance, in *Galactic Civilizations II: Dread Lords* (Stardock 2006), each species has its own technology tree [22]. This way, the replayability value can be increased with some degree – if the players are willing to change factions from time to time (or even for every new game).

Different tricks can be used for making a technology tree appear to be changing temporally. For instance, certain technologies in the tree can be revealed by in-game events. An example of this is making a language translation technologies appear only when the corresponding species is encountered for the first time.

Sword of the Stars (Kerberos Productions 2006) and *Sword of the Stars II* (Kerberos Productions 2011) are examples of digital games with variable technology trees. They randomize their technology trees for each game [14]. However, the randomization means simply leaving some non-core technologies out. So, the "base technology tree" is still a rigid structure, and it is modified by removing technologies. Similar randomization scheme was also used already over a decade earlier in *Master of Orion*.

The variability seems to be more usual among board games, but often they have basically only flat sets of possible technologies to develop, as, for instance, the 4X space strategy board game *Eclipse* (Touko Tahkokallio 2011) does. The lack of complex tree structures simplifies the matter of offering variability.

5.3 Technology Types

Technology trees are used for various purposes. In traditional 4X games, at least following technology types are often used:

- abstract technologies,
- building technologies, and
- movable unit technologies.

Building and movable unit technologies are "physical technologies", of which instances – buildings and movable units – can be produced to perform tasks. Buildings are usually spatially fixed, unlike movable units. Often movable units perform tasks like gathering or producing resources and fighting. These tasks can also be performed by buildings – for instance, cannon tower can attack nearby enemies, oil platform can be used for extracting oil, and library can produce science. One important role of buildings is to provide movable units: for instance, archers may be trained at archery ranges, swordsmen at barracks, and so on. Both movable units and buildings can in many cases be upgraded. Both of them may have some "health point" amounts for keeping track of their integrity. Because of the only fundamental difference between movable units and buildings is usually the ability to move, we use term unit to cover both of these types.

By abstract technologies we mean technologies that do not have "physical" instances in the game. Developing such technologies can lead to possibilities to produce new kinds of units or units of higher quality, or give other benefits. An important role of abstract technologies is often only to make it possible to develop other technologies; thus, they can serve as gating factors, controlling development possibilities. They can be added on a technology tree skeleton and thus the properties of the final technology tree can be adjusted.

Technology development process based on constructing and upgrading buildings (or other units) is traditionally used a lot in RTS genre, while abstract research has been heavily used in TBS games. When considering the unit-based approach, constructing a unit with production or further upgrading capabilities can also be seen as acquiring an abstract technology with the benefits given by the physical unit. Therefore, unit technologies can occasionally be seen – not only as products of having a technology, but as incarnations of technologies themselves. On the other hand, other games offer really abstract technologies – often in a tree – making it possible to produce certain kinds of units. In this case, such units cannot be equated with the technologies.

Technologies in trees used in modern digital games containing role-playing game elements represent typically feats, skills, traits, or perks of game characters. Usually, these kind of trees are used for making development of the main protagonist (and possible party members) possible; often the player may choose new abilities to be unlocked, when gained enough experience and "leveling up". Some skills may also grow better, when performing enough actions related to them – on the other words, through practice – as in, e.g., *Skyrim* or in preceding games of the Elder Scrolls series. For instance, picking a lot of locks may make a game character better in the art of lockpicking.

On the other hand, also some strategy games, like *Total War: Shōgun 2* (The Creative Assembly 2011), use perks or traits that accumulate on units (or their leaders) – thus adding a layer of personalities into the game. The player may not have – at least full – control over such development. Even if units do not have truly individual perks or traits in a strategy game, some kind of an experience bonus system – and technology trees for different units – may be offered. This makes it possible to see development happening in unit level and to distinguish between otherwise similar units. For instance, combat experience may reward a unit with a veteran status and some advantage for the battles to come.

5.4 Size

The size of a technology tree is usually rather small. However, there is great variance between different games. There are games with massive technology trees with hundreds of technologies. An example of such a game is *Warzone 2100* (Pumpkin Studios 1999, nowadays developed as an open-source game by Warzone 2100 Project) with a technology tree over 400 distinct technologies. *Civilization V* (Firaxis Games 2010), on the other hand, features "only" 74 of them in its main technology tree. Skill trees and the like are often even considerably smaller. For certain point of view, this is understandable; this way players do not feel overwhelmed by the enormous amounts of possibilities. Usually technology trees are also planned and realized by human beings with limited time and other resources. Balancing huge technology trees is a very hard task [25]. It is challenging to find and root out all the possible "invincible combinations" in a large set of possible paths of development.

The matter of size is somewhat related to the matter of rigidity: if "intelligent" and efficient ways of generating technology trees automatically will be developed, the sizes will be able to grow. (Of course, players can still be protected from the information overflow, if necessary, by limiting the view of the possibilities or by pruning.)

5.5 Technology Development Process

Many games use simple "binary development process", in which technologies are either developed or not. Occasionally it is, however, beneficial to have more LODs. For instance, in the games of the Space Empires series, there are often several levels to be researched for a single technology. Technologies can be developed either one at a time, as in *Star Ruler*, or the research (or corresponding) can be distributed somehow among different development projects, as in *Master of Orion*.

Time for developing a technology is typically fixed or depends straightforwardly on the resources allocated for the development. However, some games, like Sword of the Stars, try to emulate the uncertainties of the real life regarding schedules and results; the player can see only an estimate of the time needed for (possibly) developing a technology.

5.6 Resources

In some games technological advancement is only a matter of prerequisite technologies and time. Often, however, also other resources besides time – like money or metal – are required for conducting research or carrying out construction work needed for technological advancements. In many strategy games, it is essential to secure sources of resources and

Table 1: **Technology tree properties of some real games.**

		Year	Tree size	Tech type	Edge type	Rigidity	Means of developing techs	LODs	Parallel	Time rand.
TBS	Master of Orion	1993	****	a	or	randomized	research, stealing, exchanging, finding	*	yes	yes
	Galactic Civilizations II	2006	*****	a	–	faction differences	research	1	no	no
	Space Empires V	2006	*******	a	–	faction differences	research, finding	*	yes	no
	Civilization V	2010	**	a	and	fixed	research, wonders, heroes, trade, finding...	1	no	no
Mixed	Sword of the Stars	2006	****	a*	and	randomized	research, salvaging	1	no	yes
	Total War: Shōgun II	2011	**	a	and	fixed	research, training, constructing buildings	mixed	no	no
RTS	Dune II: The building of a Dynasty	1992	*	b	–	fixed	constructing buildings	1	yes	no
	Warcraft II: Tides of Darkness	1995	*	b	and	faction differences	constructing buildings	1	yes	no
	Total Annihilation	1997	*	b	–	faction differences	constructing units	mixed	yes	no
	Warzone 2100	1999	*********	a*, b	and	fixed	researching found artifacts	1	yes	no
	Company of Heroes	2006	*	b	–	faction differences	constructing and upgrading buildings	1	yes	no
	Star Ruler	2010	*	a	or	edges not rigid	research, trade	*	no	no
CRPG	Star Wars: Knights of the Old Republic	2003	**	p	–	class differences	gathering experience	1	–	no
	Witcher	2007	*	p	–	fixed	gathering experience	1	–	no
	Dragon Age: Origins	2009	*******	p	–	class differences	gathering experience	1	–	no
	Deus Ex: Human Revolution	2011	**	p	–	fixed	gathering experience, trading, finding	1	–	no
	Elder Scrolls V: Skyrim	2011	******	p	or	fixed	gathering experience	mixed	–	no

transport lines from them in order to be able to build units. It is rather common to have some kind of a "worker" unit type, capable to harvest and transport resources for storing. A typical example of such a game would be, for instance, *Warcraft: Orcs & Humans.* In *Total Annihilation*, resources are not collected and transported in lumps, but they are "streamed" from their sources with a constant rate.

There are large differences between games, when considering the amounts of separate resource types used. There are games with very few types and others with wide collections of them. This choice seems to be a matter of the focus and the target audience of the game. Some kind of a way to handle economy is usually wanted, but many games take a somewhat simplistic line – and use only few resource types – for keeping the game fun. On the other hand, there are players that enjoy managing a complex economy with several kinds of resources.

For instance, in *Sword of the Stars* there are only money and industrial points in use [15], while *Star Ruler* features eight resource types (six "global" resources and two planetary resources). *Civilization V* divides its resources into three categories: there are 6 bonus resources, 6 strategic resources, and 15 luxury resources [21]. Although the total amount of resource types is rather large, it should be kept in mind that not all of these are equally important for succeeding in the game. The strategic ones are typical resources, needed, for instance, producing certain kinds of units. Other types can be used for gaining additional benefits, for instance, by trading them with other factions.

Different resource types involved in technology development processes could – especially in strategy games – be classified, e.g., into harvestable (often "natural") resources, refined resources, and abstract resources. Traditional harvestable resources are the resources that can be collected and gathered from suitable world locations. Refined ones are the resources that can be obtained by refining harvestable or other refined resources (or combinations of them) by special units, like factories. Abstract resources are resources that are not harvestable or refined ones, e.g., money or science points. They can be produced by different – possibly complex – processes, often by population or facilities.

The traditional resource harvesting process – and resource-carrying workers – can be abstracted away; some games, like *Star Wars: Empire at War* (Petroglyph 2006) and *Company of Heroes* (Relic Entertainment 2006), have effectively replaced the resource harvesting scheme by getting resources by controlling strategic locations.

6. OBSERVATIONS

Based on the presented classification criteria, data of some representative real games have been gathered to Table 1. While the data set is rather limited, it is hard to draw definitive conclusions, but certain notions can still be made. The uppermost games listed are taken from the TBS genre. They are followed by representants of RTS games, and in between there are two games, *Sword of the Stars* and *Total War: Shōgun II*, mixing turn-based strategic game with real-time battles. After RTS games appear games featuring role-playing elements. Inside each of these game classes, games appear in the temporal publishing order. (See the "Year" column for the publishing years.)

The technology tree layouts for most of these games are somewhat similar; the technologies are divided into different categories based on semantics, and there is a subtree for each category. Category amounts and sizes vary a lot. *Civilization V* and *Warzone 2100* can be mentioned as exceptions; the division between categories is not clear, and there are inter-category connections. *Star Ruler* has quite peculiar approach with its grid-like technology tree and possibility to find connections.

Inside the "Tree size" column of the table, every asterisk denotes 50 technologies, with rounding done upwards. Only the sizes of the "main technology trees" are considered. Many games, however, use technology trees for several purposes (so that it is natural to consider them as separate trees); thus it may be hard to define, which one is the main tree. We also only have considered "different" technologies when determining the technology amounts, meaning that the technologies should have, for instance, different names or influences to the game. If each technology level is considered separately as a technology, the total amount of such technologies may be very high in games featuring several LODs. Regarding to the RTS games, we only have considered building types as technologies, even if some unit types could also be seen as such. Comparing the sizes of different trees in different games with different technology development requirements and time steps is extremely hard, so the size of the tree as such is a poor measure, when classifying technology trees. Thus, the sizes, as they appear in the table, should not be considered as hard facts, but rather only as fuzzy indicators.

Generally it seems that RTS games tend to have smaller technology trees than TBS games. (Exceptions occur, of course, like the RTS game *Warzone 2100* with over 400 technologies.) Many RTS games focus on military combat [16]. This may explain the difference in tree sizes; the RTS games generally emphasize battles and tactical aspects, while TBS games put typically more weight on strategic technology development – thus needing bigger trees. It may also be more demanding to construct large technology trees with basically mere buildings and unit upgrades – as is conventional in the RTS genre – than to construct large technology trees with abstract technologies – as is often done for TBS games. Also, in some RTS games consisting of several levels or battles, the technology development may start from the beginning in each of them, which reduces the total amount of technologies needed.

Also different trees of character perks, traits, feats, or talents in games with role-playing elements tend to be rather small in size. One explanation could be the fact that often such games are strongly story-driven and "tube-like". Often they are also rather short. While the plot advances rapidly, there is no need to offer many technologies, as the player would not have possibilities to develop them anyway. If a game is sandbox-like, offering players the freedom to explore the world freely and advance with the (possible) plot with their own schedules, there are often also more technologies to be developed.

The technology types used in the considered technology trees are given in the "Tech type" column. Abstract technologies are denoted by "a". There are some games, in which the technologies can be classified as abstract ones, but many of them represent concrete parts or weapons usable in, e.g., vehicle construction purposes after developing the corresponding technology. These cases are marked with "a*". Buildings and other structures (units) as technologies are denoted by "b". The character "p" stands collectively for perks, talents, skills, and feats, as different sources tend to use these terms in an incoherent way. In *Star Wars: Knights of the Old Republic*, force powers are an interesting addition to "normal" technologies, and in *Deus Ex: Human Revolution*, the technologies are called augmentations.

The "Edge type" column tells the types of dependencies. Using AND prerequisites is a common choice. This case is denoted by "and", and using OR prerequisites is correspondingly denoted by "or". It is also popular to have at most one parent technology, in which case the semantics is clear even without such a classification. These cases are denoted by "–" character. Especially different CRPGs tend to use these kinds of trees, often even with linear progression and branching factor one. Often each different, separate characteristic has its own linear tree of development, in which each level (technology) offers related (additive) perks.

The "Rigidity" column is about offering different variations of the technology trees. Often the technology trees used in CRPGs, themselves, are completely rigid, but different sets of these trees may be offered as the "whole technology tree", based on, for instance, the character class or species of the player. Exactly the same scheme is often used in strategy games: the technology tree may be combined from different trees chosen from a predefined set with similar criteria as within CRPG genre. Some games, like *Master of Orion* and *Sword of the Stars*, offer more variation by randomizing their trees for each game.

As the "Means of developing techs" column tells, each genre discussed have their own dominant ways of developing technologies. In TBS, the main method for acquiring technologies is research, while in RTS games the technological development is mainly driven by constructing units. In CRPG domain, it is generally important to gather experience in order to develop a character. Usually the mechanism is based on rewarding getting experience by some kind of advancement points for purchasing perks from a tree.

The "LODs" column tells if technologies have several possible levels or not. The binary case, in which technologies can simply be either developed or not, is denoted by the number 1, and the case of several LODs is denoted by the asterisk character.

Entries of the "Parallel" column tell, if more than one technology can be developed simultaneously. While typically in RTS games different buildings can be constructed at the same time, and construction is the way of developing technology, RTS games not allowing parallel technology development are rare. While in CRPGs the technologies are usually developed using atomic operations (like using an advancement point for purchasing a new perk), it is not seen to be meaningful to consider the matter of parallelism within this genre.

The rightmost column, "Time rand.", presents information about the predictability of the schedule of a technology development process. In many games, the exact time needed for completing a development project may vary because of different factors, like the availability of resources. However, often there are no possibilities for real surprises in the schedule. Some games, however, apply some randomization, making it possible to complete projects early or late compared to the estimated time of development. Such games are marked with "yes".

Although technology development processes and their ingame presentations have many faces and forms, the concept of technology tree can be used without difficulties with all the games we encountered. So, it seems that the concept is indeed worth further scientific research; the possible results might be widely applicable.

7. CONCLUSIONS

In this paper, different types and properties of technology trees present in several real-world digital games have been discussed. Also possible classification criteria for technology trees have been presented for further use. Moreover, some technology tree–related general properties of different game types have been identified and discussed.

With this paper, we wanted to emphasize the fact that technology trees and technology development processes of digital games in general are an important, yet so far scarcely researched – at least among the general scientific community – area; further studies should be carried out. We have tried to demonstrate that the old concept of technology tree – with a suitable definition – works well with quite different games. Therefore, results for the technology development process of a specific game may be applicable for other cases also.

As future work, we intend to continue studying technology trees. We are to create a generic tool for facilitating implementing technology tree–based technology development processes for games. We also intend to consider automated possibilities for technology generation and tree balancing. It would also be interesting to investigate, how technology tree design relates to other subtopics of the game design field.

8. ACKNOWLEDGMENTS

This work was partly funded by Tampere Doctoral Programme in Information Science and Engineering (TISE).

9. REFERENCES

[1] Adams, E. *Fundamentals of Game Design, 2nd Edition*. New Riders Publishing, 2009.

[2] Bakkes, S., Spronck, P., and van den Herik, J. Rapid adaptation of video game AI. In *IEEE Symposium on Computational Intelligence and Games CIG'08* (2008), IEEE, pp. 79–86.

[3] Björk, S., and Holopainen, J. *Patterns in Game Design*. Charles River Media, 2004.

[4] Blizzard Entertainment. Starcraft. `http://ftp.blizzard.com/pub/misc/StarCraft.PDF`, accessed January 9, 2012.

[5] Blizzard Entertainment. Warcraft II Battle.net edition. `http://ftp.blizzard.com/pub/misc/Warcraft%202%20Battlenet%20edition.pdf`, accessed January 9, 2012.

[6] Bourg, D. M., and Seemann, G. *AI for game developers*. O'Reilly Media, Inc., 2004.

[7] Buckland, M. *Programming game AI by example*. Wordware Publishing, Inc., 2005.

[8] Buro, M. ORTS – a free software RTS game engine. `http://skatgame.net/mburo/orts/`, accessed February 1, 2012.

[9] Emrich, A. Microprose' strategic space opera is rated XXXX! *Computer Gaming World 110* (1993), 92–93.

[10] Entertainment Software Association. Essential facts about the computer and video game industry. `http://www.theesa.com/facts/pdfs/ESA_Essential_Facts_2010.PDF`, accessed August 5, 2012.

[11] Fullerton, T. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, 2008.

[12] Heinimäki, T. J. Considerations on measuring technology tree features. In *Proceedings of the 4th Computer science and Electronic Engineering Conference 2012 (CEEC'12)* (Sept. 2012). In press.

[13] Millington, I. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[14] Paradox Interactive. Manual: Sword of the stars II – Lords of winter. `http://www.paradoxplaza.com/sites/default/files/pdx6105us_sots_ii_onlinemanual_1.1.pdf`, accessed January 30, 2012.

[15] Pitruzzello, J. Sword of the stars: Ultimate collection PC review. `http://www.avault.com/reviews/pc/sword-of-the-stars-ultimate-collection-pc-review/`, accessed February 1, 2012.

[16] Ponsen, M. J. V., Muñoz-Avila, H., Spronck, P., and Aha, D. W. Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning. In *Proceedings of the National Conference on Artificial Intelligence* (2005), vol. 20, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, pp. 1535–1540.

[17] Rollings, A., and Adams, E. *On Game Design*. New Riders Games, 2003.

[18] Salen, K., and Zimmerman, E. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003.

[19] Schell, J. *The Art of Game Design: A book of lenses*. Morgan Kaufmann, 2008.

[20] Schiller, N. A portal to student learning: what instruction librarians can learn from video game design. *Reference Services Review 36* (2008), 351–365.

[21] Sid Meier's Civilization V. `http://downloads.2kgames.com/civ5/site13/community/feature_manual/Civ_V_Manual_English_v1.0.pdf`, accessed January 9, 2012.

[22] Solo, A. Innovative tech trees in space strategy games. `http://www.spacesector.com/blog/2009/07/dynamic-and-specialized-technology-research-in-space-strategy-games/`, 2009, accessed January 18, 2012.

[23] Spronck, P. Dynamic scripting. *AI Game Programming Wisdom 3* (2006), 661–675.

[24] Spronck, P. H. M. *Adaptive game AI*. PhD thesis, Universitaire Pers Maastricht, 2005.

[25] Tahkokallio, T. Designers' notes #9: Technology tree. `http://boardgamegeek.com/thread/650731/designers-notes-9-technology-tree`, 2011, accessed July 19, 2012.

[26] Tech tree (SEV). `http://wiki.spaceempires.net/index.php/Tech_Tree_%28SEV%29`, accessed May 17, 2011.

[27] Total war: Shogun 2. `http://www.totalwar.com/shogun2/`, accessed May 10, 2011.

[28] Tozour, P. Introduction to Bayesian networks and reasoning under uncertainty. *AI Game Programming Wisdom 1* (2002), 345–357.

[29] Vanhatupa, J.-M. Guidelines for personalizing the player experience in computer role-playing games. In *Proceedings of the 6th International Conference on the Foundations of Digital Games (FDG'11), Bordeaux, France* (June 2011), pp. 46–52.

Publication II

T. J. Heinimäki and J.-M. Vanhatupa. Implementing Artificial Intelligence: A Generic Approach with Software Support. *Proceedings of the Estonian Academy of Sciences, vol. 62, no 1*, pages 27–38. Estonian Academy Publishers, March 2013.

Proceedings of the Estonian Academy of Sciences,
2013, **62**, 1, 27–38
doi: 10.3176/proc.2013.1.04
Available online at www.eap.ee/proceedings

Implementing artificial intelligence: a generic approach with software support

Teemu J. Heinimäki[a*] and Juha-Matti Vanhatupa[b]

[a] Department of Mathematics, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland
[b] Department of Pervasive Computing, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland; juha.vanhatupa@tut.fi

Received 18 August 2011, revised 25 November 2012, accepted 17 December 2012, available online 20 February 2013

Abstract. In computer games, one of the eminent trends is to create large virtual worlds with numerous non-player characters. Usually their artificial intelligence (AI) is implemented by scripting, which can be a burden for the application developers involved. This paper suggests an approach facilitating designing AI functionalities and striving to reduce, via software tool support, the amount of hand-written AI script code needed. Our approach is suitable for, e.g., creating autonomous agents with personal characteristics, capable of behaving in a natural manner. For instance, the goal-oriented agent paradigm can be applied easily with the approach. The definitions needed are written using a script language. Therefore, the agent configurations can be tested rapidly. We have extended an existing AI environment and created a script framework for implementing general-purpose AIs. Moreover, we have implemented software tools capable of generating script code for helping in the AI structure design and for simplifying the actual code-writing task. For demonstrating the applicability of our approach, three example scenarios specialized from the framework are presented.

Key words: artificial intelligence, autonomous agents, finite state machines, software tools.

1. INTRODUCTION

Over the recent years, virtual worlds of computer games have expanded beyond measure. They have become complicated constructs with numerous non-player characters (NPCs) and places. Having a huge number of NPCs makes it possible to offer lots of possible opponents, allies, sources of information, and trading partners. Thus, by increasing the number of NPCs, the game can often be made more interesting. However, this also means more time spent on implementing artificial intelligence (AI) for these NPCs. The NPC AI is a crucial aspect of computer games, but achieving sophisticated intelligent behaviour for a very large number of NPCs is not possible using conventional AI methods or engines [19]. The great challenge for the AI is to get the NPCs to behave in a believable, human-like fashion. The player should be able to imagine that interactions between real characters take place. The same facts apply also when considering AI agents (without graphical presentation) for, e.g., playing strategy or board games.

In this paper we propose a novel script-based[1] AI approach for computer game development. It is applicable with several kinds of games, and can be applied also outside the gaming domain. We use a separate AI framework that can be attached to the actual game engine or other program using the AI features. The purpose of the AI framework is to offer virtually all the AI support needed for creating interesting games. Thus, the application developers can concentrate on the real AI problems instead of writing lots of similar scripts

[*] Corresponding author, teemu.heinimaki@tut.fi
[1] Nowadays, game AIs are often implemented by means of scripting; scripts may be used, among other things, for communication dialogues, decision logics, stage direction, and fighting tactics [1,3]. The separation of the AI code and the main program simplifies the development process and makes it easier to modify the AI by different interested parties, like game designers or end user "modders". The matter of scripting AIs is discussed more, e.g., in [5].

for different agents. Scripting is used as a helping method, but most of the actual work is done by the framework. This paper is a derivative of the existing conference article [7], and has been extended with numerous additional considerations, clarifications, and an extra demo scenario. Also the software tool support has been extended and improved since the original paper.

Our approach is based on the famous idea of dividing and conquering: we aim at splitting the problem into smaller ones, and structuring the code accordingly. Moreover, our goal is to facilitate the work of AI developers by making it easier to design the AI structure naturally, in terms of the temporal flow of, e.g., the decision-making process. In addition, our approach is meant to make it easy to take advantage of personalization of the agents and goal-oriented action planning (GOAP) [12] principles. For these ends we have developed a general-purpose scripting framework and software tools for creating specialized AI implementations easily.

Our hypothesis is that using the approach, the productivity of AI programmers can be increased, as implementing AI features requires less work; the tools help in saving time and work in many ways. The ultimate goal of this work is to improve the quality of AIs in digital games by facilitating more efficient use of the AI programmer resources.

The framework was obtained by extending, modifying, and generalizing the AI support part of our existing CAGE game engine [18] and by making it totally independent from CAGE. At the beginning of this work, CAGE supported directly only NPC AIs implemented via state machines, and the application developer had to create more sophisticated AI implementations manually, if needed. Since, CAGE AI framework was extended with GOAP support, before the generalization and separation into our current framework.

The approach of this paper combines the use of finite state machines with the use of new kind of machines for implementing general-purpose AIs. With these machines, e.g., goal-oriented agents can be implemented easily. For testing the applicability of the approach, we have programmed three example scenarios using the framework. For taking advantage of our framework the software tools created for specializing the framework are essential.

The rest of the paper is organized as follows. In Section 2 some background and related work are covered. After that, in Section 3, we present our general AI approach in more detail. In Section 4 we explain the prototype AI framework – a realization of the general idea. Then, in Section 5, the implemented software tools and the benefits offered by them are discussed. In Section 6 we present example scenarios featuring AI implementations created by specializing the framework. The concluding remarks are given in Section 7.

2. BACKGROUND AND RELATED WORK

Almost all virtual worlds of computer games contain less intelligent NPCs in the form of animals and humans with simple behaviour; there are servants, guards, shopkeepers, and so on. State machines are a suitable method to model their behaviour. According to [12], the most of the decision-making systems in current games are implemented using state machines along with scripting. A problem with state machines is their rigidity: when encountering situations that have not been foreseen, the resulting behaviour can be poor [6].

Game worlds may contain also more intelligent agents, whose behaviours should be as complex as those of the player characters (PCs). Alas, in practice, often AI implementations – and thus, the resulting behaviours – of these agents are too simple. It is quite common to just stand still waiting for the interaction initiated by the PC, or to live only to die by the sword of the PC. This kind of simplified behaviour of NPCs can lead to boring, self-repeating, and unnatural virtual worlds. Therefore, our approach aims at facilitating creating intelligent agents of high quality.

One possibility for a basis of creating a sophisticated AI is to model personal characteristics, moods, and knowledge of the agent for inducing its behaviour. Often the AI can be improved by adding in-game interactions between NPCs. (The significance of them is discussed, e.g., in [17].) Also other methods for making games more interesting, developed over several decades by the multi-agent system community, can be used. However, the basic problem remains: often simply too much work is required to implement the wanted behaviour properly with conventional methods. Our approach strives to tackle this problem.

The usual approach for implementing new computer games is to use game engines; modern games are seldom developed from scratch [12]. Instead, a suitable game engine is modified for the new game, which is then implemented using the engine as a framework. Often some scripting language is used in addition to a conventional one. The differences between game engines are huge; their capabilities and features, including AI support features, differ a lot. It is common that the AI implementation of a game is not really supported by the game engine used, but the AI is somehow glued into the main game. On the other hand, many commercially used game engines offer some sophisticated AI support features. However, to the best of our knowledge, none of them use such a personal trait-based[2] approach as presented in this paper.

There is a wide array of publications on solving different kinds of AI problems in game worlds by NPCs and other agents. However, most of them seem to be solving quite specific problems in specific environments. Our framework, on the contrary, aims to be very generic.

[2] Our approach can be successfully used also without personality features, but the original CAGE AI framework extension was designed especially for using them.

Despite the genericity, it is as a rule easily applicable in a given task. The exact easiness depends on the case, but the lack of artificial limitations in the framework and tool support for specializing it help a lot.

The "Scripted Artificial Intelligent Basic Online Tank Simulator" (SAI-BOTS) [2] lets the players script the behaviours of their tanks using the Lua scripting language and then fight each other with them. It resembles, for instance, the CAGE system in the sense that its main programming language is C++ and it uses Lua for scripting. When using our approach, the application developer writes definitions for agents, which calculate their actions autonomously. In SAI-BOTS, on the other hand, the behaviour of the tanks is pre-scripted conventionally. Moreover, CAGE is a prototype game engine, which can be used to develop computer games of different genres. SAI-BOTS, instead, is a specific, scriptable computer game.

The AI structure of "being-in-the-world" [4], a multi-user dungeon (MUD) agent, consists of two asynchronously working modules. The reasoning module includes, e.g., logical reasoning and goal maintenance, while the real-time coping module deals with the world. The basic structure of the solution could be implemented using our approach, while it might not be the best possible way to perform ontological reasoning. The solution of being-in-the-world resembles that of the CAGE engine for goal-oriented agents. While our AI framework uses Lua, being-in-the-world has been written in Common Lisp. The framework presented in this paper is general enough to be suitable for different kinds of games and other purposes, but being-in-the-world is a MUD agent created for surviving in a certain kind of MUD.

Basketball Artificial Intelligence Tool (BAiT) [16] is a data-driven software tool system for implementing AI for a basketball game. It aims at offloading work from programmers by transferring some of it to designers. Our approach strives to offer help for generic AI implementation – not only for implementing a finite set of AI features for a single game (or genre). This, on the other hand, means that we cannot offer as high a level of abstraction as BAiT does; we expect the users of our tools to know as much of the target system and its interfaces as if they were to script the AI conventionally. (Of course, application-specific simplified scripting interfaces can be offered to be used via our tools, if necessary.) BAiT takes benefit of the fact that basketball as an activity is sequential by nature. This is the case in most sports. Also our approach is extremely suitable for creating AIs for such sequential activities.

3. ARTIFICIAL INTELLIGENCE APPROACH

For ensuring suitability for different needs, our approach includes both state machines and more advanced methods intended originally to be used with autonomous agents with personalities. We divide artificial agents into two categories according to the methods used: there are state machine agents (SMAs) and advanced AI agents (AAIAs). Instead of state machines, AAIAs use a new kind of AI machines (AIMs) for behaviour generation. The focus of this paper is on AAIAs, but the support for SMAs is included in our approach, as sometimes trying to mimic complex human-like "thinking" processes for inducing actions can be an overkill. This matter is discussed, e.g., in [10].

In modelling and developing agents, the goal orientation as a paradigm is increasingly recognized [15]. AAIAs support fully creating goal-oriented agents (GOAs) that model and use goals and motivations in their action-generating processes. We consider them as a sub-category of AAIAs. The agent types of our AI approach are depicted in Figs 1 and 2.

In our approach, the application developer describes – depending on the type of the agent – either the agent behaviour as a finite state machine or AIMs for generating behaviour autonomously. (This is the original idea of using AIMs, but they can be applied freely and offer benefits also in other kinds of settings.) The descriptions are given using a script language[3]. Scripting allows rapid application development and testing. Downside of this is the lack of efficient debugging capabilities. However, it is possible to implement the needed machines piece by piece and to test each added code block separately, without any need to recompile the framework during the process.

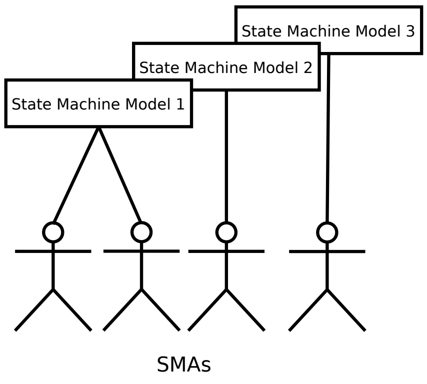


Fig. 1. State machine agents. Several agents may share the same state machine model.

[3] Using script languages or writing scripts are not bad things per se; the important thing is how and when to script. When scripting AI traditionally, the application developer writes the whole AI implementation with the given language somehow. Various methods can be used, including using state machines. This means that we recognize the usefulness of some traditional scripting methods in some cases. However, at least as far as the AAIAs are considered, instead of giving the application developers a language and free hands to use it, we want to reduce their workload by defining a clear structure and making automatic code generation possible without taking away their freedom to apply their creativeness.

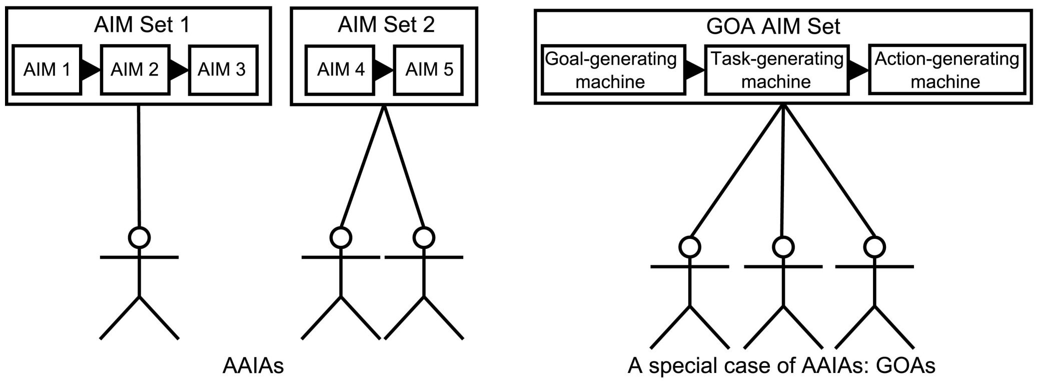


Fig. 2. Advanced AI agents. Goal-oriented agents can be seen as a special case.

Our approach enables the creation of huge virtual worlds with lots of different inhabitants with diverse AI capabilities. Of course, when the number of agents grows enough, making any kind of individual AI calculations for all of them in real time will become impossible. While our approach cannot solve this fundamental problem, it facilitates and speeds up, nevertheless, the creation of a large number of believable AI implementations. The AAIAs are not only suitable for advanced NPCs, but also for believable and seemingly intelligent co-players or adversaries. AAIAs can even be used in implementing different decision-making systems, planners, and so on. Similarly, SMAs can be used for various purposes.

3.1. State machine agents

Although the main focus of this paper is in AAIAs, we have included the support for conventional state machine-based agents as well for convenience. Simple behaviour, after all, is easily achieved using them. Another good side (in some cases) in them is the high predictability of the actions[4]. Moreover, finite state machines are not generally computationally expensive [11].

State machines could easily be implemented using AIMs. However, we choose to use a specific structure for implementing them. They are, after all, rather simple constructs that do not really benefit from being implemented as AIMs.

The application developer describes the state machine models by scripting. This work includes defining the states, state transitions, and inputs capable of firing the state transitions. It is possible to define a separate state machine model for each agent, but several agents having similar behaviour can also share one. This reduces the needed amount of copying and pasting code. An example state machine model is depicted in Fig. 3.

In [7], the general idea was that each agent using state machines has its own state machine instances and

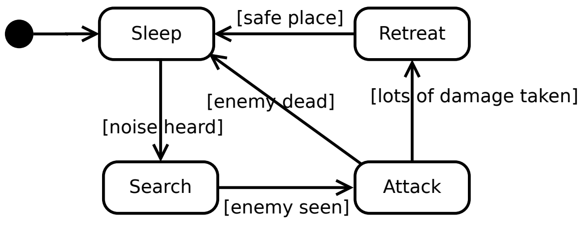


Fig. 3. An example state machine model.

can be in a different state than other agents regardless of whether they use the same state machine model or not. Connecting only one agent to a state machine instance was suggested for simplicity and general implementability. However, currently we are focusing on using shared state machines. This requires keeping track of states of individual agents, either by themselves or by the state machine construct, but in many cases considerable space savings can be achieved by not replicating similar machines.

3.2. Advanced AI agents

In our approach, every AAIA uses a number of AIMs that may modify its personal parameters, possibly adding also new ones. The application developer defines these attributes for the agent, as well as the set of AIMs to calculate new agent parameters, for example, goals and means to acquire them, and to modify the existing ones. The AIM set establishes the desired AI architecture.

The composition of a single AIM is depicted in Fig. 4. Each machine can contain several layers of calculation nodes (CNs). A CN may perform comparisons, calculations, and basically any program code. However, the idea is to keep the logics of an individual CN as simple as possible. The layers are iterated through in a fixed order according to their numbering (in Fig. 4, from Layer 1 to Layer *n*). Inside of a layer, the execution

[4] Of course, there are numerous state machine-based approaches that use probabilities and randomness, but they are out of the scope of this paper.

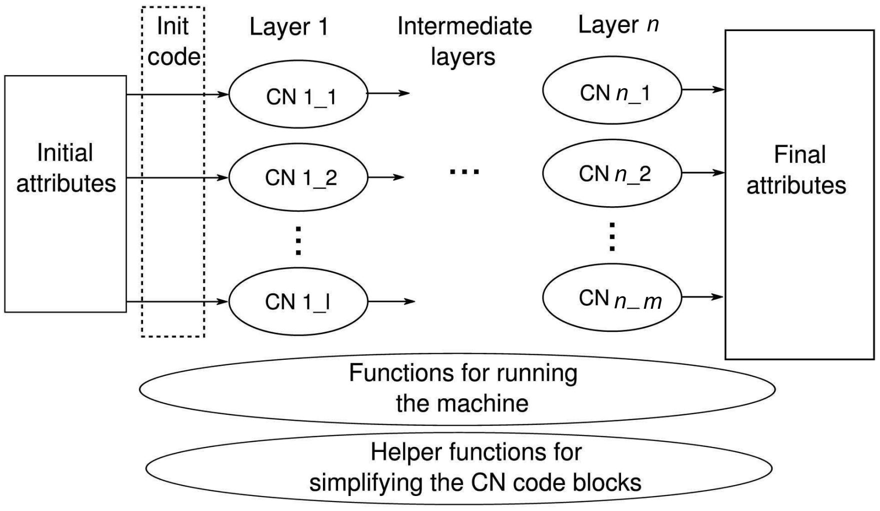


Fig. 4. The components of an AIM.

order of CNs is not fixed[5]. (By executing a CN we mean running its respective code block.) Because of this, the code of a CN must not depend on attributes altered on the same layer. It may only trust that the operations of the previous layers have been performed. Possible common initialization or input filtering code can be run before executing the CN layers. Feedforward artificial neural networks can be seen as special cases of AIMs.

The purpose of suggesting the use of several machines instead one is to simplify the overall AI construction by splitting it into logical pieces. The whole AI functionality is then achieved by chaining the AIMs, as shown in Fig. 5. The arrows pointing downwards in the figure represent the flow of execution. Passing the information from an AIM to another is accomplished by simply modifying the attribute data to which the machines have a shared access. Thus, defining any kind of complex input/output interfaces is not necessary.

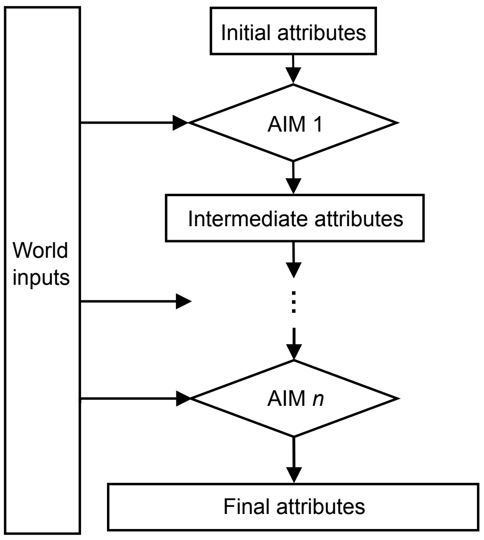


Fig. 5. Overview of the approach. A chain of machines is used to produce new data, like goals and actions to perform. The different rectangle widths depict possible growth of the number of the parameters.

Naturally, the usage of AIMs is not limited into this kind of single cascade structures. They are often viable and sufficient, but if needed, several AIM cascades can be run in parallel, or hierarchical AIM structures can be built.

The initial attributes can contain many kinds of data, including the personal traits and the moods of an agent. These parameters are simple key–value pairs. Data requiring more complex representation, like knowledge, other history-related data, and perceptions, can also be present, represented by suitable data structures. In every stage in the process, the existing attributes can be modified and new ones can be added, so the cardinality of the set of the final attributes may be greater than that of the set of the initial attributes.

The AIMs can be easily used to decide goals, generate action possibilities, and eventually produce a priority list of actions (or sequences of them) to carry out. Thus AIMs are a suitable tool for applying GOAP techniques. In Fig. 6, an example machine architecture, the basic GOAP system of the CAGE framework, is depicted. This architecture implements the general idea of Fig. 5 and uses AIMs at three levels. First, there is a machine for generating the goals based on the personality and the environment. The next machine generates tasks to be performed based on the most important goals given by the first machine. Finally, there is a separate machine for splitting the tasks into short-term actions. Effectively all the machines manipulate the importance values of their respective output parameters so that decisions can be made based on priority lists. For keeping the goals

[5] The purpose of not fixing the order is to make it possible to write parallel implementations easily.

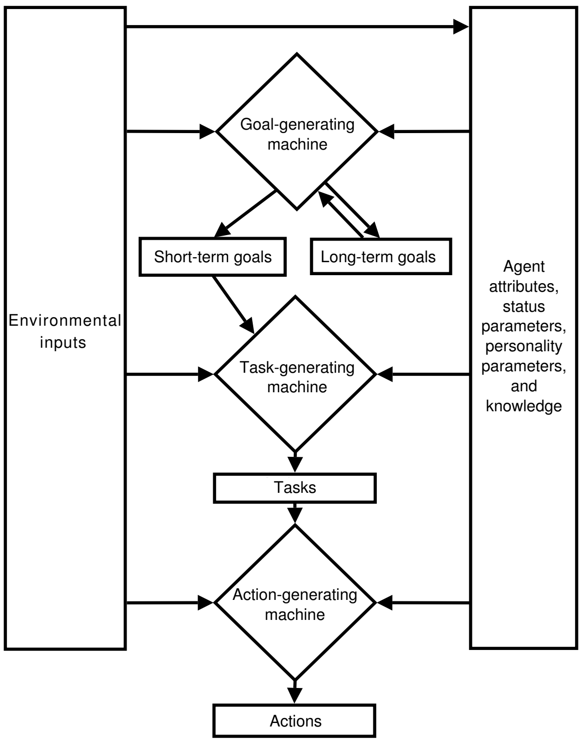


Fig. 6. Architecture of the CAGE GOAP system.

and tasks updated, the machine chain for an agent is run periodically, always when completing a task, and always when important environmental events occur.

4. STRUCTURE OF THE $F_1I_2A_3$ FRAMEWORK

For implementing and evaluating our approach, we defined the structure of the actual code so that the development process could partly be easily automated. The resulting high-level AI framework is called $F_1I_2A_3$. The name comes from "Framework for Individually Intelligent Autonomous Artificial Agents". The framework provides support for SMAs and AAIAs. It is also possible for an AAIA to command SMAs and other AAIAs belonging to its group using a simple message-passing mechanism.

The $F_1I_2A_3$ framework was obtained by extending and generalizing the existing AI framework of the CAGE game engine, so the language of choice is the scripting language Lua [9]. It is a dynamically typed popular language, which has been used in several games and industrial applications [8]. Lua is also very fast [3]. Moreover, Lua is an easy language to bind with other languages. This was an important factor when originally deciding the scripting language to be used in the CAGE AI framework. Thus, $F_1I_2A_3$-based AI implementations can be easily attached to different game engines or other "main programs".

4.1. Implementing state machines

The state machine implementation principles of $F_1I_2A_3$ are basically those of the original CAGE AI framework (although nowadays we use shared state machines actively). For using CAGE state machines, the application developer defines the state machine models consisting of state models, and connects SMAs to them. The state models include the state transitions on suitable conditions, and are written in Lua.

An example $F_1I_2A_3$ implementation of a state model is shown in Fig. 7. The modelled state is an attack state of a simple agent, which could be used, for example, in a first person shooter game. When the agent is in the attack state and sees a PC, it attacks. When there are no more enemies in the vicinity, the agent falls asleep. If injured severely, the agent will retreat.

In Fig. 7, the basic code-level structure of state models can be easily seen: they are implemented as Lua tables with three functions. One is called when entering the modelled state, one when executing corresponding state actions and triggering possible transitions, and one when leaving the state. Besides individual state models, state machine model implementations include functions for creating agent tables, setting agent properties, and running the modelled state machines for different agents.

4.2. Implementing advanced AI machines

Besides CNs, an AIM includes some metadata and functions for running and cascading the machines. The most important function is $run\_X(agent)$, in which X is replaced by the machine name (that serves as a unique identifier (UID)). This function is used for running the code segments of the CNs in the order determined by the machine structure. Using the *agent* parameter, some suitable information about the agent for which the machine is meant to be run is passed for it. The parameter may, for instance, contain only a string containing the name or the UID of the agent, or it can be a pointer to a complex agent class with methods usable from the Lua code. There are also helper functions (achievable from the CN code blocks) for, e.g., sorting and handling priority lists.

An example implementation of a CN of the CAGE Goal Generating Machine is shown in Fig. 8. The CN is responsible for updating the hunger status parameters of the NPCs. Updating takes place each time the game time passes.

```
--Attack state of an agent
State_Attack = {}
State_Attack ["Enter"] = function(agent)
  agent.draw_weapon()
end
State_Attack ["Execute"] = function(agent)
  print("[Lua]: Executing the attack state")
  if agent.hp < 0.2 * agent.max_hp then --lots of damage taken:
    change_state(agent, "State_Retreat")
    return
  end
  if safe(agent) then --enemy probably dead
    change_state(agent, "State_Sleep")
    return
  end
  agent.attack()
end
State_Attack ["Exit"] = function(agent)
  agent.holster()
end
```

Fig. 7. An attack state model implementation. In this case, *agent* is a Lua table that stores necessary properties and functions of an AI agent using the state machine model.

```
-- gc_attributes is a table containing
-- attributes of NPCs, which
-- is indexed with an agent name

function calculation_node_1_2()
  an=agent:get_name()
  gc_attributes[an].hunger =
    gc_attributes[an].hunger +
    gc_attributes[an].hunger_step *
    gc_attributes[an].time_tick;
end
```

Fig. 8. An example calculation node implementation. The variable *agent* that has been given to the machine-running function as a parameter can be used inside the CNs. In this case, *agent* refers to a class instance with a method called *get_name*() callable from the Lua code.

5. TOOL SUPPORT

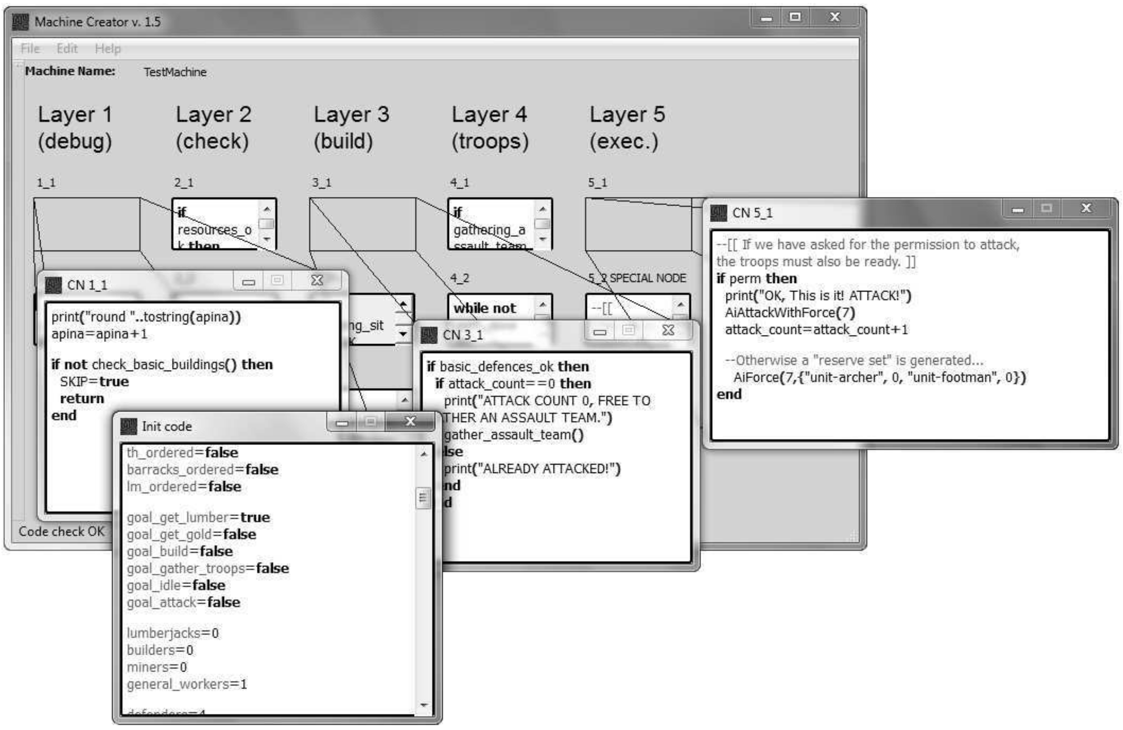
In the case of AAIAs, specializing the $F_1I_2A_3$ framework requires fixing the number and the layer structure of CNs and writing their respective code blocks. It is easy to automate the generation of the other parts of a normal specialized AI implementation. So, to help application developers in the creation of AIMs, we have implemented a software tool, called Machine Creator, for building them. Defining the layer structure of a machine and inserting the required script code for the CNs can easily be done via the graphical user interface (GUI) offered by it. A screenhot of the Machine Creator GUI is shown in Fig. 9.

After the structure of the machine under construction has been defined and the desired content of the CNs has been added, one can order the software to generate the AIM in $F_1I_2A_3$ format. In the Machine Creator version used in [7], the agent parameter value initializations had to be added to the script files by hand, but nowadays, an arbitrary initialization code can be added via the GUI. Everything else is generated automatically. The tool support speeds up the machine creating process considerably.

The Machine Creator tool has been implemented using the Qt framework [14] in order to get an easily portable tool for different platforms. It has been successfully used in Linux and Microsoft Windows environments.

Besides adding the possibility of giving the initialization code via the GUI, we have improved the Machine Creator tool in many other ways since [7] was published. The GUI has been enhanced visually, and alternative ways of performing actions have been added. Nowadays, the Machine Creator can also load the generated scripts from files for further modifications. Moreover, a syntax checking feature has been added: the tool can check the syntactical correctness of each code block. This makes debugging – and in the first place, generating bugless machines – considerably easier. Also Lua syntax highlighting support was added for making code-writing more pleasant and less error-prone.

Our AAIA approach would only have little use without the Machine Creator: GUI, syntax checking, and realization of our framework by automated code generation are the means for trying to achieve our goal of reducing the burden of AI developers. The time and



CN 1_1
```
print("round "..tostring(apina))
apina=apina+1

if not check_basic_buildings() then
  SKIP=true
  return
end
```

Init code
```
th_ordered=false
barracks_ordered=false
lm_ordered=false

goal_get_lumber=true
goal_get_gold=false
goal_build=false
goal_gather_troops=false
goal_idle=false
goal_attack=false

lumberjacks=0
builders=0
miners=0
general_workers=1
```

CN 3_1
```
if basic_defences_ok then
 if attack_count==0 then
  print("ATTACK COUNT 0, FREE TO
THER AN ASSAULT TEAM.")
gather_assault_team()
lse
print("ALREADY ATTACKED!")
nd
d
```

CN 5_1
```
--[[ If we have asked for the permission to attack,
the troops must also be ready. ]]
if perm then
  print("OK, This is it! ATTACK!")
  AiAttackWithForce(7)
  attack_count=attack_count+1

  --Otherwise a "reserve set" is generated...
   AiForce(7,{"unit-archer", 0, "unit-footman", 0})
end
```

Fig. 9. The Machine Creator user interface.

work savings come from multiple sources; the main benefits of using the tool (and our general approach) are

- the possibility of implementing scripts naturally, based on temporal (decision-making etc.) flows;
- the easiness of locating the relevant pieces of code, when necessary;
- the easiness of implementing agent personalization and GOAP features;
- the savings obtained due to automation of implementing logics related to the execution order;
- helper functions offered;
- syntax checking in the code block level; and
- easiness of writing code to be executed truly concurrently[6].

Of course, the actual savings in the amount of work or time used when implementing an AI are highly case-dependent. In some cases, our approach might not offer any benefit, while in other cases, the savings could be considerable. We do not try to make any numerical comparison against conventional scripting, as it is impossible to choose a generally applicable and fair baseline; if some AI feature was fixed to be implemented first conventionally and then with our tool, the results would not mean much, as there are many ways to solve a problem in either way. However, based on our experiments (see Section 6), we claim that at least in some cases using our AAIA approach and Machine Creator is truly beneficial.

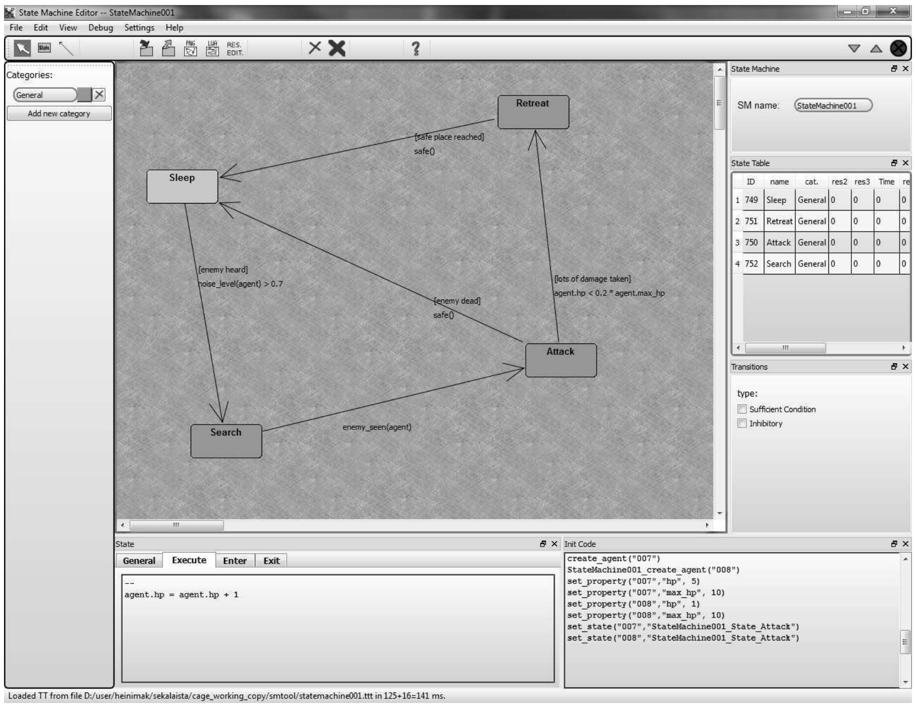
The size of the script code overhead generated for structuring the AIMs and using them grows a (small) constant amount for each CN and a constant amount for each AIM. Compared to the size of the hand-written code, the size of the automatically generated code may be considerable with simple AI implementations with several AIMs and CNs and negligible with complex implementations with only a few AIMs and CNs. Ideally, all the code written by the AI developer should be "effective" and closely AI-related, but the exact number of the lines of the code needed for some implementation depends on the skills of the programmer and the chosen way to use our tool.

We also have implemented another tool, called State Machine Editor, for creating and modifying finite state machines via a GUI and for generating Lua implementations automatically. The GUI is demonstrated in Fig. 10. With this tool, our goals are to

- provide a visual view to the logics and thus help in the design process and
- speed up the development: the time savings gained by generating the state machine code automatically are substantial.

The Qt framework has been used also for implementing the State Machine Editor for the same reasons as it was used in the Machine Creator implementation. Also the State Machine Editor runs at least in Linux and Microsoft Windows environments.

[6] It is trivial to split the generated code to pieces to be run concurrently, and the GUI of the Machine Creator helps with visualization of the code parallelization.



	ID	name	cat.	res2	res3	Time	re
1	749	Sleep	General	0	0	0	0
2	751	Retreat	General	0	0	0	0
3	750	Attack	General	0	0	0	0
4	752	Search	General	0	0	0	0

```
--
agent.hp = agent.hp + 1
```

```
create_agent("007")
StateMachine001_create_agent("008")
set_property("007","hp", 5)
set_property("007","max_hp", 10)
set_property("008","hp", 1)
set_property("008","max_hp", 10)
set_state("007","StateMachine001_State_Attack")
set_state("008","StateMachine001_State_Attack")
```

```
Loaded TT from file D:/user/heinimak/sekalaista/cage_working_copy/smtool/statemachine001.ttt in 125+16=141 ms.
```

Fig. 10. The State Machine Editor GUI.

6. EXAMPLE SCENARIOS

We have implemented three example scenarios by specializing the $F_1I_2A_3$ framework to test our AAIA approach. The first one is a scenario called Gunslinger, in which the PC plays a role of a sheriff in the situation in which a group of bandits has stolen a chest full of gold. The purpose of this scenario was to verify the applicability of $F_1I_2A_3$ for its original intended task, creating NPC AI implementations. (The problem domain analysis considering the role of a sheriff is omitted here for brevity because it is not crucial given our goal.)

The second scenario is called Gomoku. It was implemented to demonstrate that it is possible to create also a board game opponent AI using the framework. The third scenario, Wargus AI, was implemented to test the applicability of $F_1I_2A_3$ for creating hierarchical manager systems. We created a simple strategy manager AI for playing a real-time strategy (RTS) game.

6.1. Gunslinger

In the Gunslinger scenario, initially there are five bandits guarding a chest full of gold. All the bandits use the same AIM, but differ in personal traits. Additionally, one of the bandits is the leader of the group. The leader can command the others, and the commandments are normally obeyed.

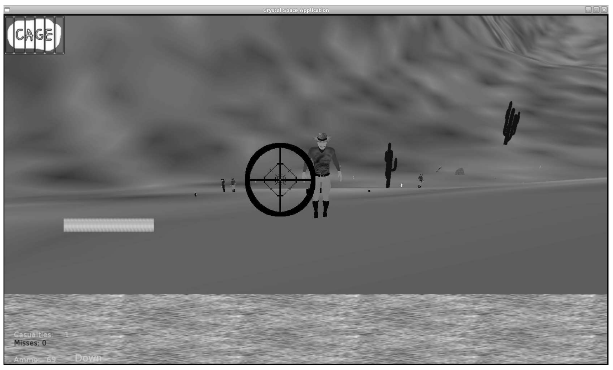
State machines could have been used for this basic operation of the bandits. However, the bandits are meant to model human beings, and human beings have different personal urges. For modelling these, AIMs are handy. So, if some personal need of a bandit to do something, for instance to flee, outweighs the authority of the leader, the bandit can also commit “rebellious acts”.

The personalities of the bandits are modelled by the following parameters: authority, bravery, alertness, tiredness, and greed. The tiredness value grows as time passes. The observations about the environment affect also: the PC can be seen, as can be the gold chest. Moreover, the bandits are aware of the casualties. In this simple scenario, there are five possible actions the bandits can perform: guarding, sleeping, attacking, alarming, and fleeing. In addition, the leader can command the others to perform these tasks.

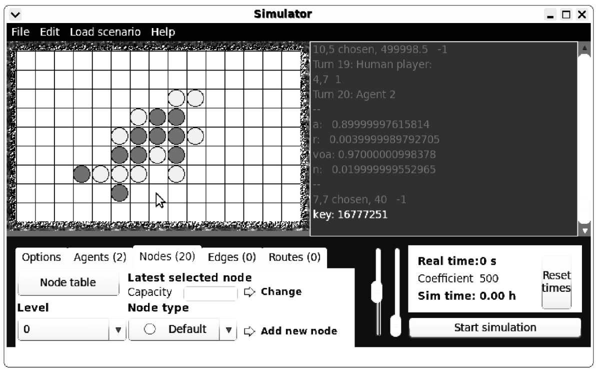
As the game engine (providing the graphics, the game loop, etc.) we used CAGE run in Ubuntu 10.04. Attaching the AIM to it was trivial and required only minor modifications as CAGE already was capable of running Lua scripts. A screenshot, in which the PC is attacking a bandit, is shown in Fig. 11a.

By using the Machine Creator tool, the implementation of the scenario, in which the individual characteristics of the bandits are clearly reflected into their behaviours, required roughly 40 hand-written lines of Lua code for the CN logics. There were a total of eight CNs forming an AIM used by all of the bandits. The scenario implementation clearly demonstrated that the Machine Creator tool and our approach in general could be successfully used for speeding up NPC AI implementations.

(a)



(b)



```
10,5 chosen, 499998.5   -1
Turn 19: Human player:
4,7  1
Turn 20: Agent 2
--
a:   0.89999997615814
r:   0.0039999989792705
voa: 0.97000000998378
n:   0.019999999552965
--
7,7 chosen, 40   -1
key: 16777251
```

Latest selected node
Capacity
Level: 0
Node type: Default

Real time:0 s
Coefficient 500
Sim time: 0.00 h

Fig. 11. Screenshots from example scenarios: (a) the Gunslinger scenario, (b) the Gomoku scenario.

6.2. Gomoku

The $F_1I_2A_3$ framework is meant to be a general-purpose AI framework. Thus, it should be applicable not only in creating the AIs for NPCs, but also in creating AIs that could be used as opponents or co-players. For demonstrating this, an AI for playing Gomoku (also known as Five-in-a-row) was implemented. This scenario was created for a self-made, general-purpose simulator software run in Ubuntu 10.04. A screenshot from a game between an AAIA and a human being is shown in Fig. 11b.

The basic implementation is simple: all the possible places for setting a counter are evaluated each round and the one with the best evaluation score is chosen. The scoring is based on the overall numbers of counters in the horizontal, vertical, and diagonal rows around the potential place and the lengths of continuous rows around it. Immediate victory moves and the rows of three counters with open ends are recognized as the special cases.

For making the AI behave like a human, the evaluation is affected by three parameters: *ac*, *r*, and *asvm*. Their values are evaluated for each ply of the game, using a single AIM, as depicted in Fig. 12. The "attacking coefficient" *ac* tells how strongly the decisions are biased towards considering only the agent's own counters. If the value of *ac* is low, the agent will play defensively considering mostly the opponent's possibilities of winning the game. Randomness coefficient *r* is used for making random mistakes in the counter placement, and the parameter *asvm* represents the ability to spot places leading to victory.

The base parameters for different player agents are absent-mindedness, interest in playing the game, tendency to get nervous during the game, aggression, and general pedantry. By altering these initial values, the playing style of an agent changes to reflect the given characteristics.

The AIM uses only four CNs, each having only one Lua line of code. In addition, some code was needed for binding the parameter values to the actual game-playing code. Still, the amount of the code and time for creating an opponent AI was minimal, and the resulting AI seems to work as expected.

Based on this scenario, the AI approach presented in this paper seems to be suitable also for non-NPC AI implementations. In this scenario, as well as in the Gunslinger scenario, the benefit gained by using the Machine Creator tool was obvious. Without using it, the corresponding AI features would have required much more time to be implemented.

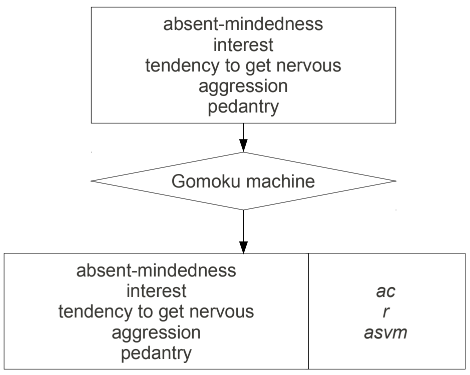


Fig. 12. Generating the playing parameters from the player agent characteristics using an AIM.

6.3. Wargus AI

RTS games are often seen as ideal test-beds for AI development. They offer a wide variety of challenges to be coped with. Wargus is a clone/modification of the well-known RTS game Warcraft II: Tides of Darkness (Blizzard Entertainment 1995). Instead of the original engine, Wargus runs on an open-source engine called Stratagus. The Stratagus engine has been previously used in different AI studies [13]. Therefore it seemed also to be a relevant test environment for our framework and we chose to implement a test scenario for it.

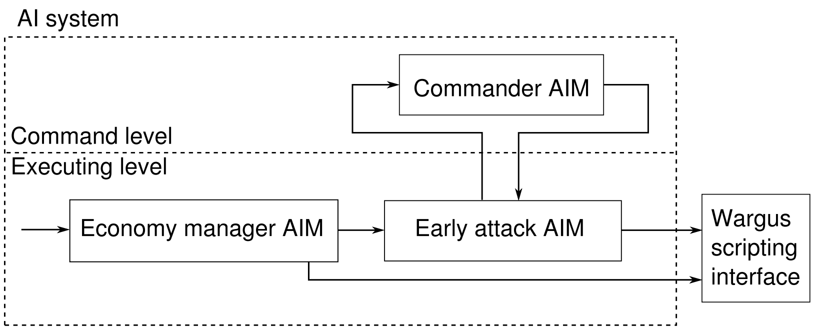


Fig. 13. The structure of the AI system implemented for Wargus using AIMs.

We tested cascading and hierarchical use of AIMs by creating an AI implementation for Wargus 2.2.6. It tries to beat its opponent by gathering an attack force and attacking the enemy relatively early in the game without giving the enemy much time to prepare; i.e. it does not rely on supremacy gained by developing technologies, but on speed. The structure of the AI implementation is depicted in Fig. 13.

The AI is two-tiered: there are command and executing levels. The executing level consists of two AIMS in a cascade. The first of them manages the economy, i.e. recruits peasants for workforce and gives orders about how to balance the efforts between gathering different resources, like lumber or gold. The second one issues orders related to constructing buildings, gathering army, and organizing attacks. This machine, however, is not allowed to work totally autonomously, but it must ask permissions to proceed. The command level consists of a single AIM that may give or deny these permissions.

The three AIMs needed were constructed using the Machine Creator. A total of 13 CNs were defined. The normal approach for implementing an AI for Wargus would have been writing a lengthy Lua script giving different orders concerning different things. We, instead, used such a normal AI script only for running our machines (which were implemented, of course, also as Lua scripts). Although the AI implemented was rather simple, it still took hundreds of lines of code. Writing and organizing it would have been more difficult without using the Machine Creator tool, which let us organize the code block execution with visual feedback, check the syntactical validity of each block separately, and split the total AI implementation into several simple machines without practically any additional work: while working already in a Lua scripting environment, the tool was able to generate all the code needed to run the machines automatically. Due to sharing the attribute data between the AIMs of an AAIA – in this case, our Wargus AI system – it is extremely easy to implement systems requiring message-passing between different components. This was also verified in implementing the Wargus scenario. A screen capture taken along the way of developing the Wargus AI using the Machine Creator is shown in Fig. 9.

7. CONCLUSIONS

In this paper we presented a general AI approach suitable for computer games and different kinds of simulations using intelligent, autonomous agents. The approach can ease the development of computer games containing large virtual worlds considerably. A crucial part of the contribution of this paper was introducing a new kind of artificial intelligence machines.

The approach was organized into a framework. Using the support offered by it, application developers can define their agents and create living virtual worlds with relatively little effort. The applicability of the framework was tested by creating tools for specializing it easily, and finally by implementing example scenarios. The preliminary results are promising: we were able to create believable and naturally behaving AIs quickly and easily.

7.1. Future work

The future work includes creating more different test scenarios and honing the framework structure and the Machine Creator tool based on the observations. We intend also to conduct tests with the State Machine Editor and demonstrate its usefulness. (This issue was left out of this paper, as the focus was on AAIAs.)

ACKNOWLEDGEMENTS

This work was partly funded by the Academy of Finland, partly by the Graduate School on Software Systems and Engineering (SoSE), and partly by the Tampere Doctoral Programme in Information Science and Engineering (TISE). Also the Foundation of Nokia Corporation has supported the work.

REFERENCES

1. Bourg, D. M. and Seemann, G. *AI for Game Developers*. O'Reilly Media, Inc., 2004.

2. Brandstetter, W. E., Dye, M. P., Phillips, J. D., Porterfield, J. C., Harris, F. C., Jr., and Westphal, B. T. SAIBOTS: scripted artificial intelligent basic online tank simulator. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05)*. 2005, 793–799.
3. Buckland, M. *Programming Game AI by Example*. Wordware Publishing, Inc., 2005.
4. DePristo, M. A. and Zubek, R. being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*. 2001, 31–34.
5. Doulin, A. Scripting your way to advanced AI. In *AI Game Programming Wisdom*. Vol. 4. Charles River Media, 2008, 579–591.
6. Fairclough, C., Fagan, M., Mac Namee, B., and Cunningham, P. Research directions for ai in computer games. In *Proceedings of the Twelfth Irish Conference on Artificial Intelligence and Cognitive Science*. 2001, 333–344.
7. Heinimäki, T. J. and Vanhatupa, J.-M. Layered artificial intelligence framework for autonomous agents. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11)*. 2011, 102–113.
8. Ierusalimschy, R., Celes, W., and de Figueiredo, L. H. About. http://www.lua.org/about.html (accessed 20.09.2011).
9. Ierusalimschy, R., Celes, W., and de Figueiredo, L. H. The programming language Lua. http://www.lua.org/ (accessed 20.09.2011).
10. Khoo, A., Dunham, G., Trienens, N., and Sood, S. Efficient, realistic NPC control systems using behavior-based techniques. In *Proceedings of the AAAI 2002 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment*. 2002, 46–51.
11. Lecky-Thompson, G. W. *AI and Artificial Life in Video Games*. Course Technology (CENCAGE Learning), 2008.
12. Millington, I. *Artificial Intelligence for Games.* The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
13. Ponsen, M. J. V., Lee-Urban, S., Muñoz-Avila, H., Aha, D. W., and Molineaux, M. Stratagus: an open-source game engine for research in real-time strategy games. In *Papers from the IJCAI 2005 Workshop on Reasoning, Representation, and Learning in Computer Games*. 2005, 78–83.
14. Qt – cross-platform application and UI framework. http://qt.nokia.com/ (accessed 20.09.2011).
15. Shen, Z., Miao, C., Tao, X., and Gay, R. Goal oriented modeling for intelligent software agents. In *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004 (IAT 2004)*. 2004, 540–543.
16. Snavely, P. J. Custom tool design for game AI. *AI Game Programming Wisdom*, 2006, **3**, 3–12.
17. Sterling, L. and Taveter, K. *The Art of Agent-Oriented Modeling*. Intelligent Robotics and Autonomous Agents. The MIT Press, 2009.
18. Vanhatupa, J.-M. and Heinimäki, T. J. Scriptable artificial intelligent game engine for game programming courses. In *Proceedings of Informatics Education Europe IV (IEE IV 2009)* (Hermann, C. et al., eds). 2009, 27–31.
19. White, W., Demers, A., Koch, C., Gehrke, J., and Rajagopalan, R. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*. ACM, New York, 2007, 31–42.

Tehisintellekti rakendus: üldine lähenemine koos tarkvaratoega

Teemu J. Heinimäki ja Juha-Matti Vanhatupa

Arvutimängude realiseerimisel on tähtis genereerida virtuaalne maailm, keskkond, kus tegutseb suur hulk mitte-mängijatest tegelasi. Nende tegelaste jaoks loodav tehisintellekt (TI) teostatakse tavaliselt skriptidena (stsenaariumidena), mille koostamine on mängu realiseerijatele lisakoormuseks. Käsitsi kirjutatavate skriptide hulga vähendamiseks pakume käesolevas artiklis lähenemise ja vastava tarkvaratoe TI funktsionaalsuse projekteerimiseks. Meie lähenemise abil võib mänguväljale luua näiteks erinevate omadustega loomulikul viisil käituvaid ja eesmärgile orienteeritud autonoomseid agente. Vajalikud mõisted spetsifitseeritakse vastavas skriptikeeles ja kohe seejärel saabki agentide konfiguratsioone kiiresti testida. On välja arendatud TI keskkond ja skriptide kirjeldamise raamistik, mis sobib ka üldotstarbelise tehisintellekti loomiseks ning vastavate skriptide automaatseks genereerimiseks. Artiklis on lähenemise rakendatavust näidatud kolme näite varal.

Publication III

T. J. Heinimäki. Considerations on Measuring Technology Tree Features. In *Proceedings of the 4th Computer Science and Electronic Engineering Conference 2012 (CEEC'12)*, pages 145–148 (orig. 152–155). Colchester, UK, September 2012.

III

Considerations on Measuring Technology Tree Features

Teemu J. Heinimäki
Department of Software Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
Email: teemu.heinimaki@tut.fi

***Abstract*—Given the recently emerged importance and rapid growth of social and economical impacts of the digital game industry, it is crucial to carry out research work for being able to develop better games. Technology trees is a topic covered hardly at all in academic papers so far. This paper discusses measuring different features and properties contained in them. The long-term goal is to facilitate producing good technology trees by the means of measurement-based enhancing and balancing algorithms.**

I. Introduction

The game industry has recently gained importance rapidly, if measured, for instance, by its monetary value. So-called *technology trees* (TTs) have been used a lot for a long time, yet the concept is virtually omitted in the academic studies [1]. This paper focuses on measuring features of TTs, because it is a matter of importance for solving the questions "how to make good TTs" and "how to improve existing ones". Answering these questions is important, because technology development modeled using TTs is an essential part of many digital games. Especially this is the case within the genre of strategy games, but nowadays TT-like structures are used also within other genres.

Naturally, TTs can be designed manually without any specific methods or explicitly given algorithms – by instinct. The resulting trees can be tested in their intended environments and adjustments can be made based on the observations. This is a method used successfully in practice. However, if TTs are large in size, automating the process of making adjustments could result in considerable savings of time (and money) and improve the overall TT quality, as more factors could be considered and more adjusting done within available time.

While probably ideas and methods resembling those we are going to present are used, they are not publicly available, but undisclosed information of game developer companies. This paper discusses thoughts related to work-in-progress, but the main contribution of it is to try and open up the issue of developing TTs of high quality to be considered also among the scientific community. Getting thoughts and ideas exchanged and evaluated critically and more openly – i.e., applying the peer review method to the production methodology – could benefit even the game industry itself.

This work has been funded by Tampere Doctoral Programme in Information Science and Engineering (TISE).

This paper assumes the TTs discussed to be used in a game setting, but similar approaches might be transferred into other kinds of settings using TT-like structures straightforwardly as well. The focus is on offline methods – i.e., the trees are measured as static entities outside the environment they are to be used – but general ideas apply also for developing online methods – measuring (and dynamically adjusting) TTs as players proceed in them.

We proceed as follows: Section II defines the essential terms and notations used in this paper and explains the semantics as needed. Then, Section III discusses different generic aspects for measuring TT features. Finally, related work is briefly discussed in Section IV before the concluding remarks of Section V.

II. Terminology, Notation, and Semantics

In this paper, the word *tree* is used synonymously to TT. TT is a structure defining relations and dependencies between *technologies* possible to develop. Despite its name, a TT might not be a graph-theoretical tree, but we expect (the graph aspect of) it to be a directed acyclic graph (DAG)[1]. In addition to the underlying graph structure, different attributes and values for them can be attached to the nodes – representing technologies – and to the edges – representing (possibly different) relations between them. This kind of interpretation of the essence of a TT is similar to that of [1] and [2].

Usually an edge in a TT represents a possibility (and requirements) of developing a technology (the end node) given another one (the start node) allows, typically by being already developed itself. In this paper, we limit ourselves into this typical usage for simplicity. A partial example TT is depicted in Fig. 1.

The word *balancing* is not used in this paper in the normal tree-balancing sense, when "balancing trees", but carrying the idea of making distinct TTs (nearly) similar in respect to some important properties. For instance, different parties in a strategy game – say, different virtual species – could have their own TTs. This is a good and viable way of making the parties different and thus also making the game more interesting. However, if such trees are not balanced well, some party may have an unfair advantage over others.

[1]A multidigraph could also be used, but in this paper, we consider only structures having at most one edge from a node to another for simplicity.

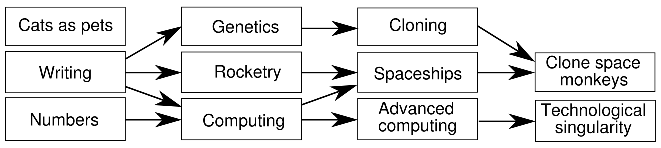


Fig. 1. A part of an example TT demonstrating the basic idea of the concept; technologies can be developed only if their prerequisites are met, and TTs are used to define or depict such dependencies.

We also aim at "balancing" trees internally to prevent the distributions of the calculated estimate values for given properties from changing too steeply when proceeding in them. For instance, the flow of the technological development a tree is limiting and guiding ought to be somewhat steady throughout the whole tree for "applying the science" to remain interesting.

Normally, roots of a digraph are defined to be those nodes, from which there is a directed path to every other node in the digraph. As our TTs are not required to be quasi-strongly connected, however, they do not necessarily have roots. Let us define a somewhat similar concept more suitable for our purposes: let *entrance nodes* be the nodes in a digraph not having any incoming edges, i.e., the nodes n having in-degree ($d^-(n)$) zero. We use the word *leaf* as it is normally defined for (graph-theoretical) trees: the node m is a leaf $\Leftrightarrow$ the out-degree of m ($d^+(m)) = 0$.

The case, in which proceeding in a tree via edge e is allowed by the start node of it, v, is denoted as $\bigodot(e)$. Usually this means that the status of v is "developed" (to a high enough level, if there are several possible levels of development). An edge starting from node a and ending to node b is denoted as $\mathrm{e}(a, b)$.

III. Feature Considerations and Different Aspects to Technology Trees

It would be pleasant and quite sufficient, if there was a simple, well defined measure for "goodness" of an arbitrary TT. Alas, this is not the case. There are several things to be considered when trying to determine how good or fitting a TT is. Moreover, one cannot treat every TT similarly, as they can be used for various purposes in different environments and within different constraints; the definition of goodness must change with the application and desired properties of the tree.

So, for analysis (and adjustments), some features of interest, estimating the goodness in some sense, must be chosen. The estimates for the numerical values of them should be easy to obtain (calculate). Moreover, for making adjusting the tree viable, there should be clear ways of affecting the values in a deterministic (or at least foreseeable) fashion. These features can be rather general concepts, like "flow of development" or "military might", but they can be observed from different points of view. It is useful to consider them at least

- locally – per node or per edge,
- globally – on (sub)tree-wide scope, and
- temporally.

Depending on the tree under scrutiny, there may be several other interesting aspects as well. These three, however, have been chosen to be discussed in the following subsections in more detail because of their generically useful nature.

A. Local Analysis

Local approach is needed in spotting the possible bottlenecks and such problems of a local nature in a tree. When they are identified, surgical adjustments can be made in order to try and make the tree work better. Even if TTs are not usually very large in size, computing the property values for a given node or edge several times should naturally be avoided. (In the future, the applications using larger TTs may be common.) Therefore we strive for being able to calculate the local values going through the tree in a single pass. With conventional, fully-known TTs this goal is also achievable with many interesting features.

So, we are interested in determining an estimated value for a given (abstract) feature for a given node or edge. For simplicity, let us assume that the development status of each technology (represented by a node) is Boolean; a technology either is developed or it is not. Let us also assume that each edge represents the cost of developing a technology "via that edge", i.e., developing the end node technology of the edge with the permission given by the fact of the start node technology being already developed. The cost can consist of several types of *resources*, time among them.

If a feature to be analyzed is of additive and cumulative nature, it is possible to determine an estimated probability distribution of its value for a node only based on the properties of the parents of it. (Of course, a reasonable order of evaluation is required.) The count of parents of a typical node is rather limited, so this leads in practice to linear time complexity of analyzing distribution estimates of a feature for all the nodes in a tree in terms of the number of nodes (the tree size). Some features suitable for this kind of evaluation could be, e.g., overall level of enlightenment (scientific development), military potential, or ability of producing certain kind of a unit.

However, there are also interesting features, for which the local values cannot be determined only based on the parents of a node or the start node of an edge. For instance, if we consider the speed of being able to proceed from a technology to another ("the maximal edge flow"), the situation with other possible parents of the goal technology – and their parents and so on – might affect the distribution.

For some cases it might be enough to extend the considerations to cover only, for instance, the Markov blanket of the node. However, unfortunately, there are many properties of the local nature one cannot evaluate only based on few local nodes or edges. For instance, the size of the subtree – that could be used as a measure of the overall usefulness or "gate factor value" of a node – is such a property (although it happens to be also quite easy to evaluate).

Let us denote a node to be developed g and its set of parent nodes as $P = \{p_1, p_2, ..., p_q\}$. Let us also denote the set of

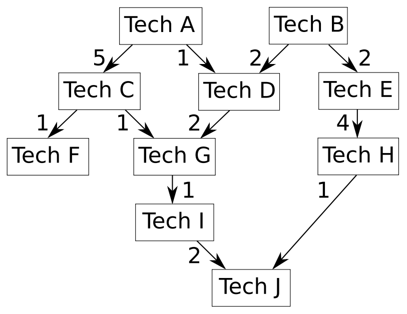


Fig. 2. A conventional TT presentation.

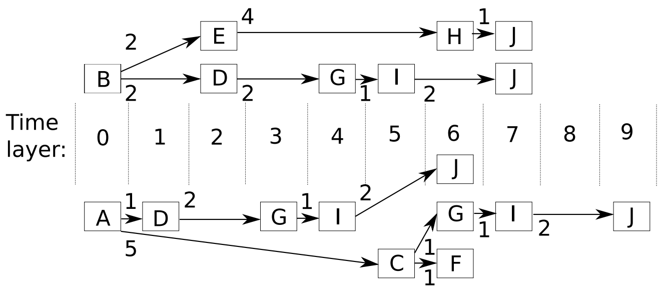


Fig. 3. The tree converted into time layer topology format.

edges corresponding the parents as $E = \{e_1, e_2, ..., e_q\}$, such that $\forall n \in \{1, 2, ..., q\} : e_n = \mathrm{e}(p_n, g)$. There are basically two different possible edge semantics commonly used: either it is required that $\forall e \in E : \bigodot(e)$, or it is enough that $\exists e \in E : \bigodot(e)$ for g to be developed (the development process of g to be started). In the first case, we speak about having AND type edges, and in the latter one about having OR type edges.

While often this kind of a simple classification is satisfactory and sufficient, we suggest a less restrictive characterization allowing different trees to be handled in a uniform way: for each edge, a set of identifiers – that can be, for instance, integer numbers – is given. Let us denote this identifier set of edge e as I_e. Let $U = \bigcup_{e \in E} I_e$. Now, all the edges can be handled as AND type edges: g can be developed $\Leftrightarrow \exists i \in U : \forall n : e_n \in E \bigwedge i \in I_{e_n} \Rightarrow \bigodot(e_n)$. (Of course, the possibility to develop a technology depends often on the resource situation or some other limiting factors, but here we are considering only the prerequisite relationships.)

B. Global Analysis

Analyzing the tree features locally plays a significant role when aiming at improving a tree by local adjustments. However, occasionally it is useful also to try and see the "big picture". Tree-wide analyses can be performed for getting descriptive data on a TT (or its subtree) as a whole. The evaluations of (probability distribution estimate) values for global features may use results of local measurements; for instance, *tree flow* could be obtained by averaging the individual values for edge flows. Another example of a tree-wide analysis would be to solve the distribution of different technology types[2] occurring in the tree under scrutiny.

Analyzing a tree on the global scope can be interesting, but the full value of analyses is obtained only when there are more trees to be analyzed; if the goal is to balance the trees with each other, measurements must be obtained for guiding the adjusting process. In some cases, trees can be balanced with each other successfully (with "rough" adjustment operations – like multiplying the values of some property by a constant coefficient throughout the tree) only based on this kind of general characterizations. Moreover, even if further micromanagement is needed, it might be a good idea to first apply macroscopic fixing, so that the local methods could really be used for their intended purpose – solving local problems.

C. Temporal Layer Analysis

TTs are not (normally) only presentations of technology relations, but structures related to time very closely: they are used for defining the possibilities of changing in time. This aspect cannot be omitted, if the actual technology development possibilities offered by a tree are to be measured.

We suggest analyzing technology development on different temporal layers. In Fig. 2, a typical TT digraph presentation is shown; each technology occurs only once. The required time steps[3] for the development processes represented by the edges are depicted beside the corresponding arrows. The topology of the graphical representation is just "somehow pleasant" without carrying any additional information value.

Fig. 3 presents the same TT differently: the node topology has been altered and some nodes have been duplicated. The idea is to represent different possible paths of development individually. We do it by forming a set of real graph-theoretical trees based on the original TT in a rather simple way.

There are two entrance nodes in the original TT. Each of them serves as a root for the corresponding (real) tree. Starting from them, the possible paths of development are followed: a node corresponding to every child technology is added "on the right layer in time" based on the knowledge of the time required for the corresponding development process (edge). The process is continued until the leaves have been reached. The algorithm is given more precisely in Fig. 4.

In digital games featuring technology development processes, they are usually meant to form a central part of the gameplay. Thus, it is important that there are not too long periods without anything notably happening in this area for a game to remain amusing. Therefore we consider different kinds of "flow" parameters important; possible problems in a tree can be detected by finding the lowest flow values, and the tree might be improved by getting rid of these problematic parts somehow – for instance, by making local adjustments to the properties affecting the flow. From the time layer topology

[2]It s common to have the technologies divided into several distinct types or categories, like military, scientific or economy-related technologies.

[3]Here we assume discrete time, but the numbers could as well represent the durations of continuous temporal intervals. For simplicity, time is also the only resource required for development in this example.

```
//Let T be the TT to be converted and S the
//tree set to be created. A, B, and N are FIFO queues.
S ← ∅; C ← ∅; A.clear(); B.clear()
for s in T.entrance_nodes() do
    t ← new_empty_tree()
    t.add(s.copy()) // as the root node without children
    o ← t.root
    o.time ← 0
    A.push(s)
    while not A.empty() do
        n ← A.pop(); C ← n.children; N.clear()
        for i in C do
            c ← i.copy()
            e ← edge(n,i)
            c.time ← o.time + e.time_requirement
            A.push(i); B.push(c); N.push(c)
        end
        o.set_children(N); o ← B.pop()
    end
    S.add(t)
end
```

Fig. 4. An algorithm for converting a TT into a set of (real) trees tracking possible paths of development in time.

representation it is easy to spot obvious flow problems; nothing very interesting with respect to the TT can happen on the time layers without any technologies. Therefore, several subsequent layers without technologies may indicate a serious design or parametrization problem.[4]

The set of technologies on a given time layer consist of those technologies that can be obtained on that time. Usually, however, all of them cannot be obtained simultaneously, because the research or producing capacity is limited; often the proceeding in a TT is even limited into proceeding via only one edge at a time. However, in such a situation, the structure can be easily updated dynamically based on the development selections made; when time is used to develop some technology, the other (real) trees are moved in time layers accordingly. In other words, a constant time is added to the time information of every technology except those in the tree one is proceeding in.

Another way to use the time layering is to consider the development of probability distributions of having different technologies in terms of time as a Markovian process. For instance the probabilities of having certain technologies in a given point in time could be evaluated this way.

IV. Related Work

To our best knowledge, there are no peer reviewed papers focusing on TT-related issues. There are lots of publications about digital games and even about developing artificial intelligences for real-time strategy games, in which constructing buildings and determining the build order often play a major role (see, e.g., [3] or [4]). TTs could be used to model such relations, but normally they are not even mentioned; the existing academic publications tend to focus on totally different aspects.

In [5], a state lattice for the real-time strategy game Wargus is presented. The states (nodes) represent sets of completed buildings and transitions correspond to constructing new ones. The representation resembles a bit the result of the time layer conversion of a TT presented in this paper. In the general case, the set of technologies on a single time layer can be seen as a state. However, the semantics of this kind of time layer states differ from the semantics of the lattice states of [5].

There are, naturally, lots of different studies on general graph properties and algorithms. TTs being interpretable as DAGs, generic graph and network algorithms can be applied, and ideas for different graph applications modified for using with TTs. At least for balancing, resource allocation, and strategy selection algorithms, also game-theoretic concepts and methods might be useful. (Analogies can be found, e.g., among load balancing tasks discussed in [6]). There are also several case-based reasoning systems (see, e.g., [7]) that could be adapted for TT-related tasks. Generic AI frameworks like [8] might also be useful for implementing AIs using, constructing, or analyzing TTs.

V. Conclusion

In this paper we have discussed generally the problem domain of measuring TT features for being able to compare different TTs and trying to improve them or balance them internally or with each other. We have also presented an algorithm for converting a TT into a format taking the usual time-dependant nature of TTs into account. Besides covering some work-in-progress, the purpose of the paper was to highlight the lack of academic studies focusing on TTs.

We intend to continue working with the issue of measuring TT features. The goal is to produce usable algorithms for the measuring itself and for adjusting trees based on the measurements. Some experiments can be carried out by extending the functionality of the existing software tool for creating functional TTs introduced in [2]. For a decent evaluation of the methods, however, a (game) development project of some magnitude should probably be carried out, or at least an existing open-source application using a suitable TT implementation should be found. Then, comparative results could be achieved by conducting queries among test users.

References

[1] T. J. Heinimäki, "Technology trees in digital gaming," in *Proceedings of Academic MindTrek Conference 2012 (AMT2012)*, in press.

[2] ——, "Facilitating technology forestry: Software tool support for creating functional technology trees," unpublished.

[3] K. Dill, "Prioritizing actions in a goal-based RTS AI," *AI Game Programming Wisdom*, vol. 3, pp. 321–330, 2006.

[4] P. H. M. Spronck, "Adaptive game AI," Ph.D. dissertation, Universitaire Pers Maastricht, 2005.

[5] D. W. Aha, M. Molineaux, and M. Ponsen, "Learning to win: Case-based plan selection in a real-time strategy game," *Case-based reasoning research and development*, pp. 5–20, 2005.

[6] B. Vöcking, "Selfish load balancing," in *Algorithmic Game Theory*, N. Nisan, T. Roughgarden, É. Tardos, and V. V. Vazirani, Eds. Cambridge University Press, 2007, pp. 517–542.

[7] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "On-line case-based planning," *Computational Intelligence*, vol. 26, no. 1, pp. 84–119, 2010.

[8] T. J. Heinimäki and J.-M. Vanhatupa, "Layered artificial intelligence framework for autonomous agents," in *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11)*, Oct. 2011, pp. 102–113.

[4]Naturally, graphical representations are not needed for applying the idea, but they may be convenient, if analyses are performed by human beings.

Publication IV

T. J. Heinimäki and T. Elomaa. Facilitating Technology Forestry: Software Tool Support for Creating Functional Technology Trees. In *Proceedings of the Third International Conference on Innovative Computing Technology (INTECH 2013)*, pages 510–519. London, UK, August 2013.

IV

Facilitating Technology Forestry: Software Tool Support for Creating Functional Technology Trees

Teemu J. Heinimäki and Tapio Elomaa
Department of Mathematics
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
Email: {teemu.heinimaki, tapio.elomaa}@tut.fi

Abstract—**Technology trees are frequently used in digital games, but academic studies considering them are few. This paper puts forward a general approach for modeling functional technology trees and implementing them separately from the programs utilizing them. A scripting language framework created for portable and modular implementations is discussed. Moreover, a software tool for designing technology trees and for generating corresponding script code automatically is introduced.**

I. Introduction

Encapsulation of functional program entities like data structures has emerged as a leading principle in programming and software development. Object-oriented programming has taken this approach to even further. The advantages of separating the implementation of a stack, say, from the application code making use of it are many. Forcing the application to use a standard interface to access the stack, for one thing, reduces the risk of misusing the data structure (programming errors). As a side effect, the stack implementation can be reused in other application programs. Today, standardized program libraries of course take care of most common functional program entities in many programming languages.

Digital game industry today is a fast growing software development area. The rapid pace of advancement has led gaming industry sometimes to overlook the golden rules of software engineering in their design principles. In this paper we want to promote the use of good programming practices in digital game programming. In particular, we provide a way to encapsulate *functional technology trees* as semi-independent entities and to create them easily. Our approach (and the tool we provide) enables reusing technology tree implementations easily and facilitates the actual work of game developers. The main contributions of this paper are evaluating this kind of an approach in general and providing insights based on empirical experimentations on the approach itself and the applicability of our novel software tool for technology tree generation.

The rest of the paper is organized as follows. In Section II, essential terms used in this paper are explained and the role of technology trees in digital games is discussed. The overall approach is presented in more detail in Section III. In Section IV, we shed light on our view to an issue highly related to technology trees – different resource types. In Section V, a general structure for functional technology tree implementations is proposed. A framework based on this structure is shortly discussed and a software tool for facilitating creating functional trees is presented in Section VI. The tool is evaluated in Section VII. Related work is discussed in Section VIII before giving the concluding remarks in Section IX.

II. Technology Trees: the Essence, Importance, and Semantics

Technology tree is an important and widely used concept in digital games. The majority of strategy games use technology trees [1], [2], and it is also common to have similar structures ("skill trees" etc.) in games with role-playing elements. Although representing technological development using tree-like structures is challenging (see, e.g., [3]), technology trees are extremely useful: despite their apparent structural simplicity, they can be successfully used for various purposes. A technology tree can be seen, e.g., as a release mechanism of beneficial entities or abilities or as a narrative tool creating the structure for the game and offering immediate goals for the players [4].

In our considerations, technology trees are directed graphs with suitable semantics, data, and functionality attached to them. They are used for keeping track of the achieved levels of development (LODs) of different *technologies* and presenting and defining the hierarchical structures of development possibilities. "Technology" refers to basically anything that can be achieved or acquired by a development process like research, training, or building. Typically, technologies offer some benefits, and for making it possible to develop a technology, some other prerequisite ones are typically needed. We use the word "tree" as an abbreviation for "technology tree" (although technology trees are not necessarily trees in graph-theoretical sense).

This paper considers technology trees in the context of digital games, and in games there are players. (See, e.g., the definition of game by Salen and Zimmerman [5].) Each player – either a human (through a software interface) or an artificial intelligence (AI) implementation – can be seen as an *agent*. More generally, the word agent used in this paper can be seen to refer to any party using technology trees, regardless, if a game or some other setting is considered.

III. The Approach: Separate, Modifiable, and Functional Technology Trees with a Generic Unified Format

Nowadays *scripting* is an essential method of implementing AI in many kinds of digital games. The separation of the game engines (conventional code) and the behavioral designs (AI scripts) has emerged as the development team sizes have grown and the complexity of the process of producing games has increased [6]. Scripts can also be used for game adjustments and other purposes. Many end users appreciate possibilities for modifying the behavior of the end product, and such an option can easily be offered by preserving the separation used in the development process in the final product and by using an understandable script language.

The traditional way of using technology trees is depicted in Figure 1: the trees are hard-coded. Some parts – like the AI – of the program may be scripted, but technology trees and their logics are not among these. We, however, suggest extending the separation scheme also to technology trees; they should be separated from the software components using them (i.e., the main game and its agents).

The motivations for separating, e.g., AI components from the game engine, are applicable also for technology trees; the separation can simplify the development process and increase the modifiability of trees by different parties. So, the technology tree implementations should not only be separated physically, but also to be easily modifiable.

There are already some real-world applications using separate and even modifiable technology tree descriptions. For instance, *Star Ruler* [7] (Blind Mind Studios 2010) uses technology trees described in text files using an easily understandable description format. However, such real-world formats – and technology tree semantics – differ from each other, so the trees are not interchangeable between different applications, and it may be hard to convert a tree from one application to be used with another.[1] The (possibly existing) tools for creating trees in an application specific format are useless with other applications using different formats. Occasionally, a standardized textual representation format (like XML) is used for describing technology trees. However, the mapping of data into such a format can be made in multiple ways.

For making technology trees reusable, they ought to have a similar structure and communicate with other parties through well-known, common interfaces. This kind of trees could potentially be used by several different programs. The separation of trees from their users and unified interface and structure also facilitate research work on technology trees and implementing tools for generating and modifying user-independent trees.

[1]We do not claim it to be common to want to swap technology trees between different applications, or even to want to take a tree from an existing application and use it as such in a new application. However, it could be efficient to be able to, for example, take the tree of an application as basis for the tree of its sequel or an otherwise more-or-less similar application. The modifications needed could require significantly less effort than implementing the whole tree from a scratch. Also when implementing an existing application for a new platform, having technology trees as separate entities capable to work as such on several platforms could be extremely cost-effective; the technology trees and their functionalities could possibly be transferred as verbatim copies without any modifications.

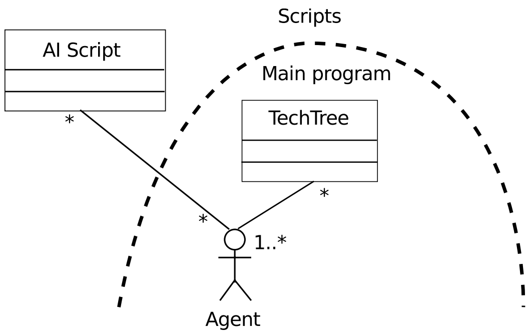


Fig. 1. Traditionally, technology tree implementations are internal and integral parts of the programs using them. There might be, however, some external modifiable parts, like AI scripts.

With the existing formats it is typically only possible in practice to represent the structure of the tree. The approach proposed in this paper, however, is based on functional trees – not only structure representations or data containers. Therefore, in addition to the previous requirements, a tree must be able to contain also the implementation for generic technology tree functionality and offer useful services to the user via the interface. With this idea we aim at reducing the workload of application programmers, which can be achieved, if technology trees in accordance to our approach can be implemented easily. This paper demonstrates – by introducing a software tool for the task – that this really is the case. The tool also offers a higher level of abstraction than just writing all the code manually. This can be beneficial, as often technical (coding) skills of game designers are somewhat limited [8].

IV. Time and Other Resource Types

Resources are a fundamental concept, when considering typical games using technology trees. In this paper, we assume that a game may feature several resource types (e.g., gold, hydrogen, or unicorn horns) with arbitrary names (and other information). Agents may have different sources producing resources, and normal resources can be stockpiled by them. Upgrading a technology requires some (either fixed or calculated) amount of each resource type, and degrading may return some resources to the resource pool of the agent.

The medium of digital games is time-based [9], and controlling the game experience in terms of time is one of the most important purposes technology trees are used for. Therefore, time is a central resource type.

In Section IV-A we consider technology development process possibilities from the angle of using resources. In Section IV-B we explain, how all the different resource types can be handled similarly despite the peculiarities of time.

A. Time, Development Process, and Genericity by Similarity

In the simplest case, a technology development process only requires time[2] without any other costs; technologies can be chosen to develop or be studied without other resources spent. This is the case, e.g., in Total War: Shōgun 2 [10] (when considering the "Mastery of the Arts" trees). In more complex situations, the cost of a technology can consist of several parts; for instance, some science points, some money, and some special metal can be required for developing a technology.[3]

Sometimes the development may not require any time, but happens instantaneously after the development order has been issued. However, often the case is more complicated. Usually the technology development processes are not instantaneous, but it may take several turns in a turn-based game, or several seconds or minutes in a real-time game, for an ordered development process to complete. This models the real-world time demands of research, training, construction, or upgrading tasks.

The waiting phase between issuing an order to develop a technology and actually getting the development task completed is closely connected to the cost in terms of resources like gold or lumber. Often all such needed resources are reserved – subtracted from the total pool of resources available – in the beginning of the development process (right after the command to develop a technology has been given), and after that developing a technology takes a fixed amount of time. However, there are also games, in which the agent can control the development more freely – e.g., put a project temporarily on hold. The player may also be able to control the spending of resources for an ongoing development project, i.e., a technology development task may be completed gradually and be allowed to consume resources bit by bit, without the need to stockpile all the resources beforehand. In some games a number of resources (e.g., science points) are generated each turn (or otherwise over time) and directed to technologies under development. The technologies are achieved or upgraded, or levels of them gained, after stockpiling enough of these cumulating resources. In these schemes, the resource feeds for development tasks naturally affect the respective technology development times.

Clearly, the characteristics of time often differ quite a lot from the characteristics of, e.g., ore, timber, or science points. Therefore, it would be tempting not to consider time as just one resource type among others, but handle it separately. However, flexibility is gained by considering time as a normal resource type – and letting potentially any of the resource types behave as time typically does. Instead of complicating matters, this actually simplifies the resource-handling process.

[2]In another extremely simple resource model basically only gathering *experience points* matters. This is common considering, e.g., so-called skill trees or talent trees. However, when using such trees, an intermediate level of complexity is usually added in the form of "skill points" or something similar. Experience can also typically be gained in various ways, so we consider "time only" model simpler – at least, when time is measured using homogeneous portions, like turns.

[3]In [11], also *opportunity costs* – i.e. what one does have to give up for performing an action – are considered as a cost type. We agree that such things are important to consider, when managing technology tree usage and trying to solve dilemmas of selecting one of mutually exclusive paths to follow. However, opportunity cost estimates can be derived based on the internal basic properties of the tree (as characterized in Section V), and are more related to design and usage than fundamental structure, so this cost type is henceforward omitted.

B. The Means for Uniform Resource Handling

In our approach, one can define two significant quantities of each resource type for each possible development process: *initial requirement* and *total requirement*. The idea is that for initializing the process one must spend the resources indicated by the initial requirements, then during the process one can spend more resources, and when the overall amount of the spent resources matches the total requirements (plus possible side costs occurred during the process), the task is completed and the LOD changed. This way

- there is no need to define explicitly, if a technology can be developed gradually or not,
- all the technologies can be handled similarly, and
- even cases between the two conventional development models can be used: some amount of resources may be needed for starting to develop, but this threshold may be less than is required for completing the task.

Naturally, for the time resource (if present), the initial requirement for all technologies should be zero, as it (at least typically) cannot be stockpiled. All the resource types have also two boolean properties, "omnipresence" and "stockpilability". When a portion of a resource is given to be used by an agent, the omnipresence value defines, if it can and should be "copied" to all the ongoing development tasks as such instead of distributing minor portions of it to different tasks (if applicable). Stockpilability, on the other hand, defines, if a resource can be stockpiled. Time, of course, is a typical omnipresent and non-stockpilable resource type.

Time also is conventionally unstoppable by its nature. Therefore, it can be "wasted" – the total amount of time used for getting a technology developed may not equal the time actively developing it, if there has been pauses during the development process, as time continues to flow also during these pauses. In some scenarios one might want to consider total time of development and effective time of development separately. For being able to handle all the resources – including time – similarly (and for allowing similar behavior to potentially all the resources) we let the technologies start accepting all the resources that are allocated for them, when starting a development task – including the omnipresent resources available for all the ongoing tasks. The development process may be put "on hold" in respect to any resource type; that type is then blocked from being used, but further resource allocations of that type are rather marked as wasted contributions. The process can later be continued in respect to that particular resource type.

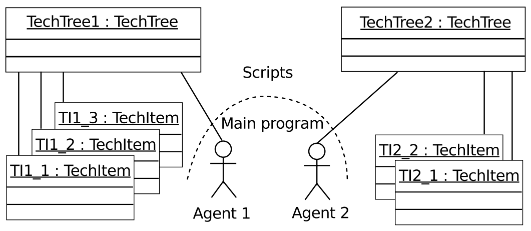


Fig. 2. Every agent uses its own technology tree instance handling its relevant TIs. The trees can be structurally similar or different.

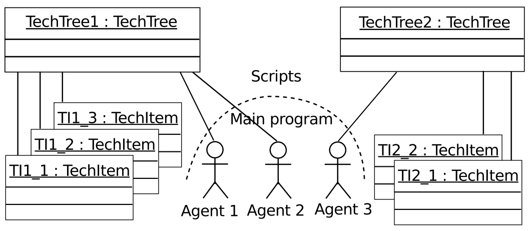


Fig. 3. Agents may share technology tree instances. In this case the tree must be able to distinguish between TI values for different agents. The agent bookkeeping can be implemented on tree level or on TI level.

V. Proposed Technology Tree Structure

A technology tree stores and organizes data. Thus, its implementation is effectively a data structure with some semantics attached to it. Moreover, it must be able to contain its internal functionality and to interface with its users. Of course, many different approaches for implementations are possible.

Although the specifics of a good implementation can depend on the environment and requirements, some general guidelines, like simplicity, ought to be followed. In our approach, technology trees are composed using simple building blocks (effectively forming decorated DAGs offering interfaces for users and having internal management and functionality implementations). In the following subsections we introduce them and their relations to each other.

A. Technology Items

The most fundamental units are called *technology items* (TIs). As the name suggests, they are used for modeling technologies.[4] They can be seen as nodes in the digraph structure defining the tree, augmented with the necessary data.

Typically names and labels are important for presentation purposes. This is the case also with TIs: they represent technologies, and technology names are useful for referring to them. Thus, every TI should store a name of the corresponding technology. The technologies of a game are commonly organized into different categories; there can be, for instance, economic, cultural, and military technologies available. Therefore, each TI also contains a category attribute. The categories may affect different matters in the gameplay – e.g., having some number of technologies of the given category developed may be a prerequisite for advancing, or the cost of a technology could be modified by already having technologies of the same category. Naturally, a TI must also be aware of its development status. Moreover, while it often is the case that developing a technology to some level (upgrading or degrading) may trigger some program code to be executed, the corresponding code, naturally, needs to be known. Similarly, some other code could be executed, say, once in every turn in turn-based games or every x milliseconds in real-time games, given a TI has been developed to a high enough level. While it may in the practice be rather rare to run TI specific code when dropping a LOD or when being not developed above some threshold level, these cases are included for flexibility. In fact, we want to allow different code blocks to be executed for every possible level of development, if necessary. In addition to technology names, e.g., images or more detailed textual descriptions can play representational roles. These "other things" we include as application specific additional data.[5]

So, every complete TI contains information about its

- name (textual),
- category (textual or other identifier),
- LOD for each agent using the tree (generally a numeric value, but quite often a Boolean one is sufficient),
- code to be run when changing the LOD,
- code to be run when on a given LOD, and
- additional data.

Of course, sometimes merely a subset of these is enough. The unneeded data items can be left out or dummy stub entries can be used.

We could choose to instantiate separate technology trees for each agent, as depicted in Figure 2, or allow multiple agents to share trees, as illustrated in Figure 3. Each agent having its own tree instances is a comparably inefficient solution in terms of space needed, if the number of agents is large. On the other hand, this way one does not need to handle the resource situations etc. for each player in each tree or TI implementation. So, if the amount of agents is limited and small enough, it might be a good idea to use separate technology tree structures for each agent for simplicity. If this is not the case (as, e.g., in so-called massively multiplayer games), the necessary agent-specific bookkeeping must be handled by the trees. In this case, tracking data like the LODs of the agents falls naturally to be carried out by TIs. If different kinds of agents use dissimilar technology trees, several tree structure instances are needed anyway, but the number of them is typically significantly smaller than the number of the agents,

[4]Technology as a word is not very illustrative, as it is used very liberally in the context of technology trees in digital games. However, as was the case with technology trees, we stick to using the established term. Technologies can be categorized, e.g., into abstract technologies, building technologies, and movable unit technologies [12], but in this paper, the nature of them does not play any role.

[5]Also "age" or "era" of a technology, present in some games, can be included in the additional data. The limitations etc. to the technology development caused by them can be implemented in the code blocks associated with TIs.

because normally the agents can be divided into few classes, each using one kind of a tree (or a small set of trees).

B. Technology Item Connections

Conventionally, there are some kind of connections between technologies in a technology tree; otherwise, the "tree" would be reduced to a mere set. These connections are essential for describing prerequisite relationships and thus (at least partially) defining, if a technology can be developed in a given situation. They can be straightforwardly included in our technology tree model: let us consider connections between TIs. We treat them as edges of the tree-defining digraph with additional data attached to them. Let us call these augmented edges *technology item connections* (TICs).

The edges in a technology tree graph could indicate the *resources* needed for upgrades [13], but in our scheme, the resources are not handled by TICs. (For information on resource handling, consult Section V-C.)

In Figure 4, a simple example technology tree is illustrated. The boxes represent TIs. The entrance nodes (the TIs to be initially available for development) are on the top of the figure, and children are represented below their parents, as the TIC arrows indicate. (In this example, the only possible level change for all the TIs is from the "undeveloped" state to the "developed" state.)

When considering a customary "normal" connections between a TI and its parents, sufficient development of some of the parents is a necessary condition for the technology represented by the child TI to be developed. It may or may not be a sufficient condition. If it is, the development of the child can be made possible via a single parent (OR prerequisite). Otherwise, a set of parents, typically all of them, (AND prerequisites) must be developed to high enough level to enable the child technology development. For instance, in Figure 4, "Fire" and "Gunpowder" should be AND prerequisites for "Rocketry", as semantically it makes sense that both of them are needed. On the other hand, "Mechanics" and "Electronics" are OR prerequisites for "Computers" – the achieved computers could be either of mechanic or electronic nature, and thus it would be enough to have one of these prerequisite TIs developed.

Normally, the purpose of a technology tree is to make it possible to develop technologies given some prerequisites are met, but occasionally it may also be beneficial to be able to prohibit technology development based on existing technologies. Using inhibiting connections with (or instead of) enabling ones for this could be tempting in some situations. However, in this paper we only consider enabling connections.[6] This means, that by default TIs cannot be developed. To change this, some (or all) arriving TICs are required to "fire", i.e.,

[6]If it is possible to have both inhibiting and enabling connections arriving to a TI, some kind of an overriding priority order needs to be defined, and the big picture becomes unclear. Later on (see Section V-C), we tackle the issue of making it possible to inhibit the development of a TI without any need for increasing the complexity of incoming TICs.

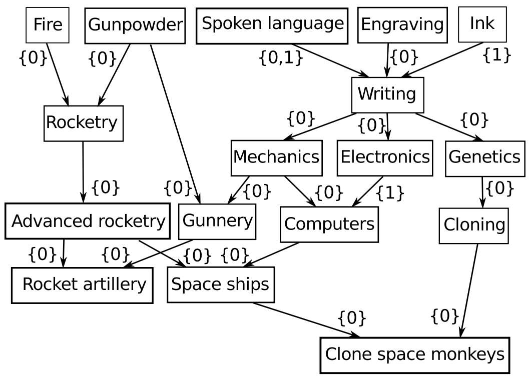


Fig. 4. A simple (partial) technology tree.

to allow the development on their behalf.[7] This means that we are considering TIs as prerequisites for other TIs only indirectly; the firing statuses of arriving TICs are to be used as parameters for the AND or OR operators (depending on the desired semantics), not properties of the parent TIs, when checking, if a technology can be developed or not.

The firing conditions of TICs can basically be arbitrary, but for keeping the tree intuitive, they should mainly be based on the LODs of the corresponding start TIs. Naturally, the firing condition should also take the intended level change into consideration, if there are several possibilities. When applying our terminology to the existing game technology trees, the most usual firing condition is the requirement of high enough LOD of the start TI for increasing the LOD of the end TI (often from the "undeveloped" state to the "developed" one).

Many simple technology trees could operate successfully, even if the firing condition was a property of a TI. In this case, all the TICs having the same start TI would fire as a group. However, for flexibility, it would be beneficial to be able to have different firing conditions for different TICs originating in a single TI; therefore we consider the firing conditions as properties of TICs.

Let us follow the notation used in previous work [14] and denote the case, in which TIC e fires, by $\bigodot(e)$. Let us denote the case, in which TI i is allowed to be developed (in the context under observation, including the intended level change) by its parents, $\triangle(i)$. For $\triangle(i)$ to be true, only some incoming TICs or all of them may be required to allow the development – i.e., fire – depending on the prerequisite type used.

In the real world there may be several distinct ways to acquire a technology. For one to be able to model this properly with technology trees, it must be possible to define several alternative prerequisite TIC combinations for a TI to be developed. For obtaining this kind of improved flexibility we use the characterization suggested in [14]: the central idea is to treat all TICs as AND-type ones and to apply the AND operator for subsets of incoming TICs rather than for all of

[7]For the *entrance nodes* (the TIs without incoming TICs) other means must be used for enabling their development: see Section V-C.

them, when evaluating, if it is possible to develop a TI. These subsets are formed in such a way that for each subset S of incoming TICs with respect to a given TI i to be developed it holds that $(\forall s \in S\colon \bigodot(s)) \Rightarrow \triangle(i)$. Let us call these TIC sets *sufficient and-type connection sets* (SACSes). For instance, in the tree of Figure 4, one SACS could be composed of the TICs starting in TIs "Spoken language" and "Ink" and ending in TI "Writing", as for learning to write, a language and some way to store symbols – for this specific SACS, using ink – are needed.

Now, it is essential to know, if a TIC belongs to a given SACS. A TIC can also be a member of several different SACSes simultaneously. For these ends, we indicate the SACS memberships by letting each TIC have a set of *and bunch identifiers* (ABIs). These identifiers for distinguishing between different SACSes can be, for example, integer numbers. A TIC is a member of a SACS if and only if its ABI set contains the ABI corresponding to that SACS. In Figure 4, example ABI sets are shown besides their respective TIC arrows.

For each TI, ABIs for each SACS among the incoming TICS must naturally differ from each other. However, ABIs do not have to be globally unique, as demonstrated in Figure 4. (In this case, two distinct ABI values are enough: for each TI, there are at most two corresponding SACSes.) The technology "Writing" can be developed, when "Spoken language", and either "Engraving" or "Ink" are developed, and "Computers" can be achieved either based on "Mechanics" or "Electronics". The development of other technologies requires all of their prerequisite technologies to be developed, as all the incoming TICs have only ABI "0" in their respective ABI sets.

After these considerations, we can characterize a TIC as a quadruplet $[a, b, \mathrm{f}, \mathbf{I}]$. In this representation, a stands for the start TI (parent of b), and b is the end TI (child of a). Function f: $\mathbf{A}\times\mathbf{B}\times\mathbf{B} \rightarrow \{\text{True, False}\}$,[8] in which $\mathbf{A}$ is the set of possible LODs of a and $\mathbf{B}$ is the set of possible LODs of b, is the firing condition function, and $\mathbf{I}$ is the ABI set.

C. Development Items

When considering a TI development task using our approach, there are two restrictive layers. The first one is that of requiring approvals from the TICs (and thus indirectly from the parent TIs). The second one is about the matters of time and other resources required for the task. For handling the functionality of this second layer and for modeling the development process from the issued order to the desired change in the LOD of a TI, we introduce *development items* (DIs).

One DI is meant to be able to handle the development processes of one technology, so each TI has a DI associated to it. A DI should be able to perform the calculations needed to update its information content as needed, and not only serve as a data structure. While it might be possible – and maybe in many cases even natural – to implement DIs as parts of their respective TIs, we consider DIs in this paper separately. This is done for clarity. Proper implementation of a DI may require a lot of code, and on the idea level, it is easy to distinguish between the ordinary TI code and DI code dealing with the development process.

DIs contain information about

- arbitrary conditions of development (ACs),
- the target LODs for each agent attached to the tree (the current LOD of an agent meaning the technology is not under development by it),
- phases of LOD-changing tasks (POTs) of the agents for their ongoing tasks (i.e., "readiness" or "health" values of the goal technology levels for different agents, values in the range [0.0, 1.0]),
- amounts of available (unused) resources allocated for their respective development tasks by different agents (including possible working units like constructors or scientists),
- booster coefficients (used for hurrying up or slowing down the development) of the agents for their respective ongoing tasks handled by the DI,
- effective resource usages for different agents, tasks, and different resource types so far,
- total resource usages ("pauses" – i.e., blocked resources – included),
- a progress function for calculating increments for the POTs of an agent (given the resources to spend) and for updating the resource stockpile situation,
- Boolean pause statuses of the tasks for combinations of agents and resource types: paused or not,
- functions for (possibly) returning resources to the resource pool, if the development is canceled,
- functions for (possibly) randomizing the needed amounts of resources for the tasks,
- a function for implementing the degrading model applied, and
- functions for (probabilistic) special events taking place during the development processes.

The POT values keep track of the progress of the current development tasks of an agent. They are initialized to the value 0.0. The idea is to modify the value corresponding to an agent and a task every time the progress situation changes. The correction for the POT value is given by the progress function. The desired LOD change is executed (and the respective TI code run) when POT meets the value 1.0. In this case, the POT value will be returned to 0.0. If, however, the value 0.0 is met again after some initial development without hitting the value 1.0 first, a special function can be run, as one might want to handle this kind of "failures" in some special way. This function can be defined only once by the tree and used by all the DIs. Same applies to other DI functions. However, the DIs

[8]A larger set, like $\mathbf{R}$, could also be used as the codomain. However, in this paper only the binary set is considered for simplicity.

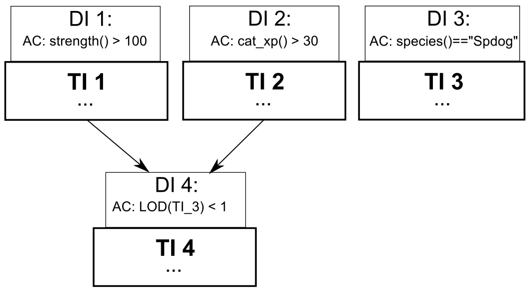


Fig. 5. DIs constraining development. For the entrance nodes TI 1, TI 2, and TI 3, the ACs of their respective DIs define the conditions that must be satisfied in addition to the ordinary resource needs for enabling the development. Developing TI 4 should not be possible after developing TI 3 in this example, and the AC of DI 4 is used to implement this constraint.

should be free to override the default functions by ones suited to their individual needs, when necessary.

One reason to include the possibility of defining ACs is to offer the entrance nodes a way to restrict the development also in other terms than those of resources: they do not have incoming TICs, so an alternative way is needed. ACs can also be used to emulate inhibiting connections discussed earlier. This is demonstrated in Figure 5. One should also be able to connect each AC with an arbitrary code block to be run in the case, in which the technology development is blocked by not satisfying the condition; in this way, ACs can be used for handling different situations and serve different purposes flexibly.[9]

Often gradual technology degrading over time is not needed or desired, but in some development models it is an essential property. For instance, in *Total Annihilation* (Cavedog Entertainment 1997), a project put on hold loses progress slowly, therefore requiring more time and resources eventually to be completed. Therefore DIs need also knowledge about the degrading model to be applied.

Sometimes the estimated development time is not desired to be a hard fact about the actual process, but some randomization offering possibilities to complete the project before or after the scheduled time may be wanted. Thus, we include also functions for implementing the desired randomization scheme.

D. Development Manager

The fundamental technology tree building blocks explained and their mutual relations are depicted in Figure 6. (We find a UML class diagram to be an illustrative and clear way to represent them, but naturally the concepts do not have to be implemented as classes.) The *development manager* (DM) occurring in the figure represents the part of the technology tree implementation managing resources, distributing them for technology development tasks and possibly getting some resources back from the development processes.

In order to fulfil its task, the DM communicates directly with DIs that manage resource consumption and stockpiling for individual development tasks. It also communicates with the set of SACSes (composed of TICs) connecting TIs, because it must be aware of the development limitations posed by them. The task of offering the interface for users also falls naturally for DM, as it is a managing entity "keeping the tree together" and having the necessary connections.

[9]For instance, in some cases one might want to limit the maximal time of a development task. This kind of a time limit can be implemented as an AC, handling the consequences of exceeding the allowed time in the corresponding code. (One can, for instance, get the corresponding TI unlocked (or leveled up) – given enough time, one could overcome the lack of resources. An alternative is to make the agent start the development again from scratch – the resources can be lost because of, e.g., decaying during the long storing time. Probably denying the development totally after an unsuccessful development attempt is generally not a good idea in normal games, but there might be exceptions for this rule of thumb.)

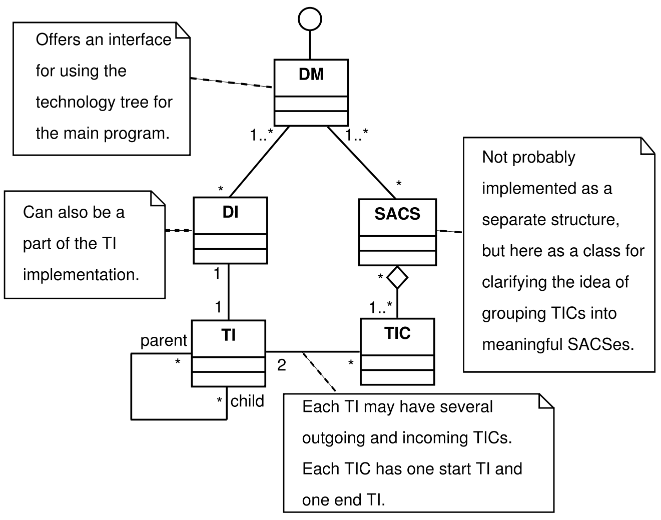

Offers an interface for using the technology tree for the main program.
DM
1..*
1..*
*
*
Can also be a part of the TI implementation.
DI
SACS
Not probably implemented as a separate structure, but here as a class for clarifying the idea of grouping TICs into meaningful SACSes.
1
1
*
1..*
parent
TI
TIC
*
2
*
*
child
Each TI may have several outgoing and incoming TICs. Each TIC has one start TI and one end TI.

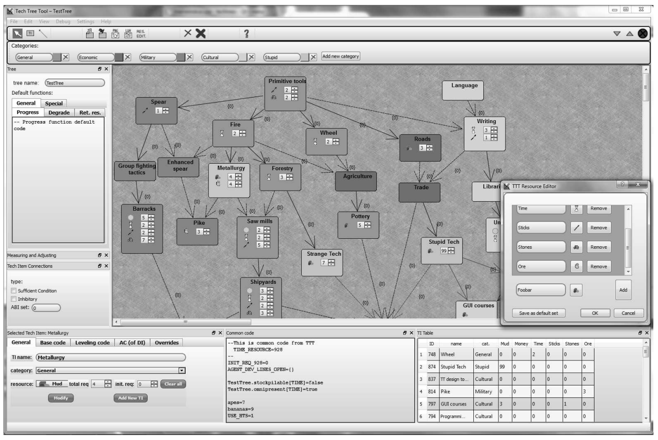
Fig. 6. The technology tree components.

VI. Tool Support

Based on the ideas presented, we have defined a general framework for implementing technology trees – each usable simultaneously by several agents – easily with Lua scripting language. As explained, we wanted the implementations to be portable and usable in modular fashion with different game engines and other applications. Therefore, Lua was chosen to be used: it is easy to bind with other languages and it can be used on different platforms. Moreover, Lua is fast, light-weight and widely used in the game industry [13].

While the structure of the elements in the technology tree presentation of the framework is rather simple, it was possible to automate the technology tree specialization process with reasonable effort. So, we implemented a software tool for creating functional technology trees via a graphical user interface (GUI). The resulting trees can easily be used from the actual game (or other program) code. The tool, called Tech Tree Tool (TTT), was written using the C++ language. The GUI (see Figure 7) was created using the Qt framework (Qt 4.7.0).

TTT serves as a decent editor for designing technology tree structures graphically. One can easily, e.g., add, remove, and move TIs and modify their properties (like categories)



TI name: Metallurgy
category: General
resource: Mud total req 4 init. req: 0

```
--This is common code from TTT
  TIME_RESOURCE=928
--
INIT_REQ_928=0
AGENT_DEV_LINES_OPEN={}

TestTree.stockpilable[TIME]=false
TestTree.omnipresent[TIME]=true

apes=7
bananas=9
USE_HTS=1
```

	ID	name	cat.	Mud	Money	Time	Sticks	Stones	Ore
1	748	Wheel	General	0	0	2	0	0	0
2	874	Stupid Tech	Stupid	99	0	0	0	0	0
3	837	TT design to...	Cultural	0	0	0	0	0	0
4	814	Pike	Military	0	0	0	0	0	3
5	797	GUI courses	Cultural	3	0	0	0	1	0
6	794	Programmi...	Cultural	0	0	0	0	0	0

Fig. 7. The TTT user interface.

and resource requirements, as well as define ACs and default functions for different purposes. Default functions can also be overridden for specific technologies, if needed. Defining prerequisite relations can be done easily by drawing edges between TIs (and by adjusting ABI sets, if needed). Created technology tree models can be saved and reloaded.

These features combined with the most important property of TTT – the ability to generate Lua code automatically based on the created tree models – makes the tool quite useful. It is possible to create technology trees with TTT and then use them easily from other applications via a simple interface (containing, at the moment, 37 functions). Prototyping – as well as producing final versions of – technology trees is easy and quick using the tool.

TTT is still work-in-progress, but most of the ideas presented in this paper are already present in the generated Lua code (that relies heavily on using Lua tables). However, some simplifications are made for now. The TI contents discussed are included, but the additional data is restricted to be textual for now. It is also assumed that the LOD information of a TI is binary – that a technology either is developed or not – and increasing the development level is always the goal, not reducing it.

TICs with their corresponding ABI sets are encoded into the Lua scripts to make it possible to make dependency inquiries. Our intent is to include functions for making these inquiries also into the Lua interface offered to the main program, but at the moment, the TIC knowledge is used only internally on the Lua side. Other interface functions still to be implemented include functions for setting weights for allocating resources for developing different technologies (if it is possible to develop several technologies simultaneously), as well as functions for directing the automated goal technology selection of the development manager and for setting (long-term) goals explicitly.

The aim of the interface implemented so far is to make it possible to initialize the starting situation (regarding to technology levels and resource stocks of different agents), and after that simply to tell the development manager, when it should act. (This could be once in each turn in turn-based games.) When agents gain or lose resources, time passes, or when new technologies to be developed must be set automatically or are chosen by players, the DM can be informed using simple functions.

For the main application and the technology tree scripts to be able to communicate with each other using common terms, we use unique identification numbers for every TI and agent. If several different technology trees – i.e., technology trees generated separately using TTT, or by other means – are used, the difference between them is made by their respective names (adjustable in TTT).

Naturally, there is still room to expand the code generation abilities significantly and to create more extensive set of interface functions to be used in real applications. However, based on our preliminary tests with the current interface, we believe it is a good core set of functions to start with.

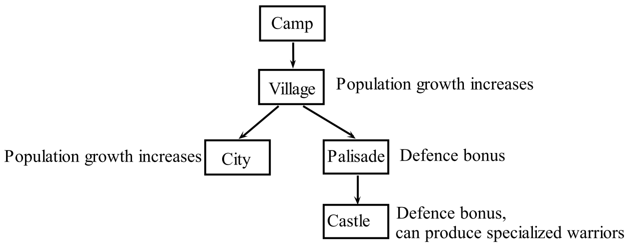
VII. Evaluation

TTT compiles and runs without problems at least in Ubuntu 11.04, Ubuntu 12.04, Microsoft Windows XP Professional, and Microsoft Windows 7 Enterprise environments. For confirming the suitability of TTT to its intended use, some test were conducted. Two different desktop PC systems were used as test environments. First of them (System 1) was BlackStorm X6 running Ubuntu 12.04, having AMD Phenom II processor and 8 GBs of RAM. The other one (System 2) ran Microsoft Windows 7 Enterprise and had Intel Core 2 Duo E8400 processor and 4 GBs of RAM. Such systems were chosen, as they could be seen as somewhat ordinary and common PC environments at the time of writing this paper.

A. *Efficiency Test*

A technology tree with 200 TIs was designed via TTT user interface. This quantity was believed to be high enough for typical use, as the main technology tree of Sid Meier's Civilization V – the latest product of a game series that is fundamental regarding using technology trees in digital games – features "only" 74 technologies [15]. Often technology trees are even smaller. Ten distinct resource types were used. This number was believed to be higher than in typical cases (see, e.g., [12]). Editing the tree was considered to be easy and fast. Of course, these are vague and subjective terms, but at least it can be said for sure that the GUI sped up the planning and editing process considerably, when comparing to the alternative – writing all the code by oneself.[10]

[10]Of course, one could question the benefits of using a GUI. It might also be fast and easy to edit, for instance, a well structured text file and add and modify technologies by copying and pasting text blocks within such a file. However, this kind of an approach requires good knowledge about the application-dependent tree representation format, while our approach aims at providing a unified view to the problem of tree creation and modifying on a higher lever of abstraction. We claim that in the general case the tree structure is easier to understand based on a graphical representation, than a textual one (given that the graphical representation is suitable for its intended purpose).



Camp
Village
Population growth increases
Population growth increases
City
Palisade
Defence bonus
Castle
Defence bonus,
can produce specialized warriors

Fig. 8. The development tree of settlements in the demo game.

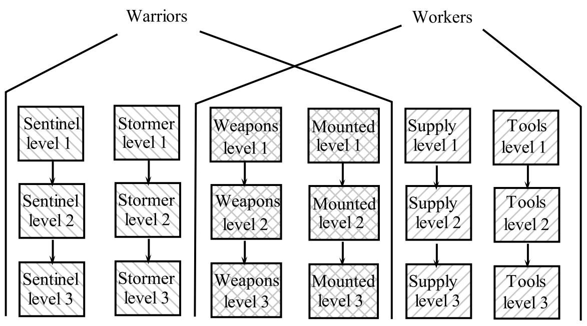
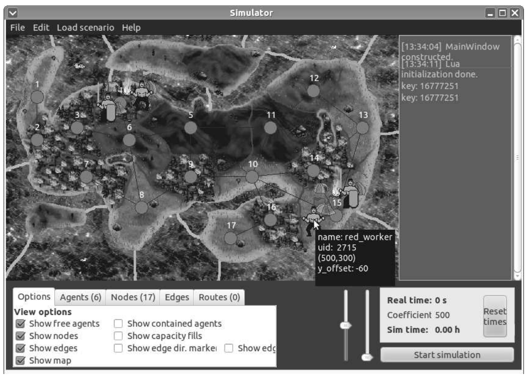


Fig. 9. The specialization tree of warriors and workers in the demo game.



[13:34:04] MainWindow constructed.
[13:34:11] Lua initialization done.
key: 16777251
key: 16777251

name: red_worker
uid: 2715
(500,300)
y_offset: -60

View options
- [✓] Show free agents
- [✓] Show nodes
- [✓] Show edges
- [✓] Show map
- [ ] Show contained agents
- [ ] Show capacity fills
- [ ] Show edge dir. marke
- [ ] Show edg

Real time: 0 s
Coefficient 500
Sim time: 0.00 h

Fig. 10. A screenshot of the demo game.

The average branching factor of the test tree was 1.5, which we believed to be close to typical cases. TTT had no problems whatsoever with handling the test tree. Editing, adding or moving TIs or defining their properties did not induce any inconvenient delay. Loading such a tree from a hard drive, however, required in System 2 approximately 3060 ms of time. This was not seen overly problematic, as loading a tree is a relatively rare operation to be performed. System 1 could handle the loading in 730 ms. (The saving operation seemed to be approximately 10–30 times faster than loading.)

B. Demo Game

For testing the automatic Lua code generation and the interface offered by the generated code, a demo game was implemented. It is a turn-based strategy game for human players, written in C++. There are two movable unit types: warrior groups and worker groups. The world is divided into regions, and there are five different terrain types: plains, hills, forest, mountains, and water. Each region has a designated terrain type. Each turn, the units having *movement points* (MPs) left can be ordered to move by designating an adjacent target region. This way, a player can append regions to the path, until the MPs of the unit are used up. The MP amount required to move from a region to an adjacent one depends on the terrain types of the regions, the type of the unit, and the movement bonuses affecting it. The amount of MPs available for a unit each turn depends on its type and specializations (i.e., technological development).

Besides the movable unit types, there are fixed units: different kinds of settlements. Each player commands a faction with a number of units. Workers are capable of building camps and upgrading them into more advanced settlements given enough time, which depends on the terrain type of the region. The main purpose of warrior groups is to execute war. The worker groups are also capable of fighting, but they are not even nearly as effective as equal-sized warrior groups.

Each unit type has an associated technology tree. The tree for settlements is sketched in Figure 8. A camp can be upgraded into a village, and that can be converted either into city or a palisade that can be upgraded even further into a castle. The (additive) bonuses given by different settlement types are presented roughly in the figure. The upgrade requirements are not shown, but they can be given in terms of time, population amount and worker amount in the region. The population of each movable unit grows (or shrinks) in the beginning of each turn. The growth is affected by the unit type, possible other units present in a region, and (pseudo)randomness.

Warriors and workers share a common technology tree. There are several different specialization or equipping categories or “lines” in the tree, as depicted in Figure 9. Some of them are reserved for warriors only, some of them for workers, and some of them can be chosen to be developed by either type.

Each warrior or worker group can only develop technologies of at most two of them. There are *sentinel* technologies that give defensive bonuses, *stormer* technologies that boost attacking abilities, *weapons* technologies that can give general combat bonuses, *mounted* technologies giving movement bonuses, *supply* technologies affecting the morale of the units in the same region and making it more probable that wounded units recover, and *tools* technologies that improve the working speed of the workers.

A screen capture of the game is presented in Figure 10. Due to the lack of space, the details of the game rules, combat mechanics etc. are omitted, as they do not affect the evaluation of the possible benefits of using TTT.

Based on the demo game implementation, it seems that with TTT the workload concerning implementing technology tree structures is reduced significantly, if compared to conventional implementations. TTT was found to be useful even with these tiny trees, and we strongly believe that in more complex cases with increased amounts of TIs and rules, the benefits offered by a GUI and automated code generation manifest themselves even better. The functional Lua technology trees generated by TTT were found to be easily usable by a somewhat typical kind of a technology tree–using program.

VIII. Related Work

The publications focusing on technology trees are scarce. Some textbooks, like [16], present notions on them, but it is extremely hard to find academic, peer reviewed papers on the issue. A recent article [4] discusses, among other things, the deterministic nature of technology trees and compares trees found in four strategy games with each other. A conference paper [12] presents an overview of using technology trees in digital gaming and offers some classification criteria. In another one [14], matters related to measuring features of technology trees are considered. There are also different articles on gameplay optimizations for different games (see, e.g., [17]), but they are related only indirectly and rarely even mention technology trees.

Different tools are available for creating and altering application specific trees. Different undisclosed tools are used in the game industry. As far as we know, there are, however, no existing publications or tools with a goal of unifying the ways of implementing technology trees. TTT strives to be a generic tool, offering automated generation of universally usable and platform independent technology tree implementations.

IX. Conclusions

We have presented a general approach of modeling technology trees, a widely used concept in the game industry. We have also presented a way to use the model in real applications by structuring the approach as a framework that makes it possible to easily create modular and functional technology tree implementations outside the main program. For creating such implementations in an automated fashion, a software tool, TTT, has been created, and empirical experiments have been carried out. Our approach aims at providing a unified way of implementing technology tree structures for better reusability and interchangeability.

Exhaustive testing has not been conducted yet, but the results of preliminary experiments are promising. The ideas behind the tool seem to be applicable for creating functional technology trees. The tool itself is capable to facilitate the work of designing and implementing technology trees significantly: the actual code generation is automated and the design work needed can be carried out using a GUI designed for the task.

It is worth noting, that TTT is not only a software offering a GUI for editing a simple data structure, but it also generates generic technology tree functionality, thus saving time considerably. While the functional technology trees are generated as Lua scripts, they can be easily further modified by hand, if necessary. Using Lua also allows the resulting technology trees to be used on several platforms without any modifications.

As the interface for using technology trees from external (main) applications is clearly defined and simple (although adjusting and augmenting it continues for now), attaching TTT-generated technology trees in modular fashion to games is easy. Trees can be created quickly with TTT, and main programs can use them straightforwardly. This approach removes the burden of implementing technology trees as internal structures of the programs needing them, and thus helps keeping things simple. The independence of technology trees of their respective users may also help in parallelization of software development processes.

As future work, we intend to use technology trees generated by TTT with more realistic applications. We are going to conduct tests in different settings and to modify the tool and the ideas behind it, should a need arise. The automatically generated Lua interface offered to the main application will be extended with more functions. The existing functionality will also be augmented, for instance, by implementing proper ready-made planning functionalities. Performance enhancements will also be made, especially concerning time usage of loading technology tree models from a disk.

Acknowledgment

This work has been funded by Tampere Doctoral Programme in Information Science and Engineering (TISE). Also the Foundation of Nokia Corporation has supported the work.

References

[1] P. Tozour, “Introduction to Bayesian networks and reasoning under uncertainty,” *AI Game Programming Wisdom*, vol. 1, pp. 345–357, 2002.

[2] A. Rollings and E. Adams, *On Game Design*. New Riders Games, 2003.

[3] E. Watrall, “Chopping down the tech tree: Perspectives of technological linearity in god games, part one,” http://www.gamasutra.com/view/feature/131570/chopping_down_the_tech_tree_.php, 2000, accessed July 19, 2012.

[4] T. Ghys, “Technology trees: Freedom and determinism in historical strategy games,” *Game Studies*, vol. 12, no. 1, 2012.

[5] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. MIT Press Ltd, 2003.

[6] I. Millington, *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[7] Blind Mind Studios, “Star ruler,” http://starruler.blind-mind.com/, accessed February 1, 2012.

[8] P. Tozour, “The perils of AI scripting,” *AI Game Programming Wisdom*, vol. 1, pp. 541–547, 2002.

[9] A. Gazzard, “Unlocking the gameworld: The rewards of space and time in videogames,” *Game Studies*, vol. 11, no. 1, Feb. 2011.

[10] “Total war: Shogun 2,” http://www.totalwar.com/shogun2/, accessed May 10, 2011.

[11] V. Harmon, “An economic approach to goal-directed reasoning in an RTS,” *AI Game Programming Wisdom*, vol. 1, pp. 402–410, 2002.

[12] T. J. Heinimäki, “Technology trees in digital gaming,” in *Proceedings of the 16th International Academic MindTrek Conference 2012 (AMT2012)*, Oct. 2012, pp. 27–34.

[13] M. Buckland, *Programming game AI by example*. Wordware Publishing, Inc., 2005.

[14] T. J. Heinimäki, “Considerations on measuring technology tree features,” in *Proceedings of the 4th Computer Science and Electronic Engineering Conference 2012 (CEEC’12)*, Sep. 2012, pp. 152–155.

[15] “Sid Meier’s Civilization V,” http://downloads.2kgames.com/civ5/site13/community/feature_manual/Civ_V_Manual_English_v1.0.pdf, accessed January 9, 2012.

[16] E. Adams, *Fundamentals of Game Design, 2nd Edition*. New Riders Publishing, 2009.

[17] C. A. Paul, “Optimizing play: How theorycraft changes gameplay and design,” *Game Studies*, vol. 11, no. 2, May 2011.

Publication V

T. J. Heinimäki and T. Elomaa. Quality Measures for Improving Technology Trees. *International Journal of Computer Games Technology, vol. 2015, article ID 975371*, 10 pages. Hindawi Publishing Corporation, April 2015.

V

Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2015, Article ID 975371, 10 pages
http://dx.doi.org/10.1155/2015/975371

Research Article

Quality Measures for Improving Technology Trees

Teemu J. Heinimäki and Tapio Elomaa

Department of Mathematics, Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland

Correspondence should be addressed to Teemu J. Heinimäki; teemu.heinimaki@tut.fi

Received 23 October 2014; Accepted 22 March 2015

Academic Editor: Yiyu Cai

The quality of *technology trees* in digital games can be improved by adjusting their structural and quantitative properties. Therefore, there is a demand for recognizing and measuring such properties. Part of the process can be automated; there are properties measurable by computers, and analyses based on the results (and visualizations of them) may help to produce significantly better technology trees, even practically without extra workload for humans. In this paper, we introduce useful technology tree properties and novel measuring features implemented into our software tool for manipulating technology trees.

1. Introduction

Skill progression of player characters (PCs), acquiring new talents, achieving perks, and developing available technologies are essential parts in many popular digital games. Mental, physiological, and material upgrades significantly affect the narration of a game. They also partly establish the game logic and make it visible and understandable for the player. When players are able to grasp the intended logic, a feeling of more logical consequences follows. It leads to more profound user immersion and increases the enjoyment of play. The upgrades can also be used to partition the game horizon into separate stages or eras. Furthermore, the accumulating abilities set short-term goals for the player.

Technology trees (or *tech trees* for short), as they are called especially in strategy games, are structures used routinely to model and implement these aspects of gameplay. They describe and define the dependencies of technological (in a very broad sense) progress followed in the game. In other game genres, tech trees are called with different names such as *skill trees* or *talent trees*. Also within the genre of strategy games there is some variance in the terminology; for instance, the term *tech web* is sometimes used. In this work, however, we categorize all such structures under the common concept of tech trees.

Despite the central status of tech trees in modern (real-world, commercial) games, our experience and a survey show that they suffer from many deficiencies in practice. In this work, we demonstrate that weaknesses in tech trees can be automatically screened and detected by using suitably defined quality measures. That opens up the possibility to rectify the defects that might have been introduced to tech trees unintentionally. Our vision is that in the future automatic improvement possibilities offered by generic tools will be available for tech trees failing to meet the given quality criteria. The primary reasons for striving for such automation are our aim to improve efficiency and the desire to overcome the human limitations and guarantee a high game quality. One should also remember the fact that a game designer is not necessarily a programmer [1]. Different easy-to-use software tools are already used for many purposes in real-world game development. They have often a considerable positive impact on the quality of the end products, so why not create and use tools also to construct, measure, and adjust tech trees?

The main contribution of this work is the introduction of several measures for monitoring the quality of technology trees. In addition, a general-purpose software tool called Tech Tree Tool (TTT) [2] is improved by implementing quality monitoring. In order to empirically qualify the measures, we determine them, as applicable, for *Sid Meier's Civilization V* (Firaxis Games, 2010) (Civ 5) and discuss the results.

Even though we use Civ 5 as our default example, the results and observations are by no means restricted to this particular game or even the genre it represents. We could as well have used, for instance, *Kerbal Space Program* (Squad, Beta Version 0.90.0 2014) as an illustrative example; it is a quite different game: a space flight simulator. However, in order to expose the illustrations to as wide an audience as possible, we stick to Civ 5 as our running example.

Next, in Section 2, we focus on the nature of technologies and characterize tech trees. We continue in Section 3 by pointing out some typical problems within them. After that, we introduce the measures that we find usable as indicators in quality assurance, first for technology trees (in Section 4) and then for individual technologies (in Section 5). Thereafter, Section 6 sheds some light on our automated measuring implementation, which is then tested with Civ 5 in Section 7. Finally, Section 8 discusses briefly the matter of adjusting and improving tech trees based on measurements, and Section 9 concludes the paper.

2. The Nature of Technologies and Technology Trees

Tech trees have become essential structures forming crucial mechanisms in digital games. There are different kinds of tech trees, and they "fulfill various strategic and narrative functions" [3]. Traditionally, technology trees have been associated with strategy games, of which most feature them [4], but effectively similar constructs are used also within other game genres. A classical example is *Diablo II* (Blizzard North, 2000) that is often mentioned as the game that truly introduced such structures within the genre of computer role-playing games. The idea of the "character skill tree" of the game was based on technology trees of strategy games [5]. Adoption to different genres and subgenres has led to variance in terminology; a tech tree-like structure may come under the disguise of a different name, but the difference is essentially only in the terms used for the items (e.g., "technologies," "perks," or "talents") that can be selected to be developed or purchased. The targets of their effects also vary; for instance, "talents" typically affect individual characters and "technologies" larger entities such as tribes, nations, or species. However, the basic idea of the structures remains the same. In this paper, we consider all these variants as technology trees. Correspondingly, those selectable items are henceforth called *technologies* or shortly *techs*.

A tech tree can be seen, for instance, as "a structure that controls progress from one technology to a better technology, enabling the player to create better facilities or more powerful units" ([6], page 141), or simply "a flow-chart showing the dependencies of upgrades and buildings" ([6], page 72). These views reflect the main two sides of the coin called technology tree; tech trees are typically seen either as (1) structures (mechanisms) for defining and controlling development (upgrading), based on dependency relations of technologies, or (2) flow-chart-like presentations on these dependencies. In this paper, we concentrate more on the former point of view.

Tech trees are not usually trees in graph-theoretical sense but can take forms of different, typically acyclic and weighted, (multi)digraphs. It is natural to treat technologies as nodes of a graph and define their relations using weighted edges. (In order to be able to handle different technology trees using a general characterization, we use a derivative of this basic approach: we basically consider weights to be associated with sets of edges. Sections 4 and 6 illuminate this more.) In addition to these basic components of any graph, additional data, for instance, information related to presentation or effects of the technologies, are typically needed.

Games come in various forms and flavours, and so do technologies used in them. For instance, personal properties, perks, traits, feats, and talents of game agents (e.g., PCs) can be seen and modeled as technologies, if the agents can learn or otherwise acquire them. Such techs are often found in games featuring roleplaying elements. For instance, a thievery-oriented PC might be improved by making it learn *Basic Lockpicking* or, already having acquired, say, *Improved Stealth*, advancing it to *Stealth Mastery*.

In strategy games, on the other hand, the players typically act as commanders, "controlling things from a discrete distance" [7]. This is reflected on the "traditional tech trees" of these games: the effects of technologies target typically larger wholes than individual characters.

Even when discussing these original tech trees, the word technology is used quite liberally: not all "technologies" found in strategy games are technical in nature at all. A player might, for example, develop *Banking* in order to bolster the economy of the controlled nation, invent *Free Time* to raise the general morale, or let the citizens learn *Ghuli'Xi'ulian Language* to be able to communicate with a neighboring game faction. These are *abstract technologies* [8]; their manifestations do not correspond to visible unit instances "on the board." Abstract techs can represent, for instance, cultural or administrative innovations and achievements. Various subtypes, like forms of government, religions, ideological movements, monuments, and pieces of art, fall into these categories. However, abstract techs can also contain technologies, technical devices, and technological inventions in the conventional sense (such as *Printing Press* or *Jet Engines*).

Sometimes, technologies represent different unit types (like samurai, healer, or dragon) used in the game, or, more accurately speaking, the abilities to produce units of the corresponding types. For instance, developing a technology called *Fluffy Bunny* could let the player produce fluffy bunnies (visible unit instances of the type "fluffy bunny") within the game world (given that other possible prerequisites, e.g., being able to pay the production costs and having necessary production facilities, are met). A single technology may also have several effects of possibly various natures; it is possible to have techs enabling unit production and giving simultaneously also other abilities [9].

Similarly, building types (if buildings are not treated as units) may have their specific construction-enabling technologies, and technologies can, in addition to their other effects, allow constructing buildings of certain types. Especially in real-time strategy games, the capability to construct buildings often depends on the types of the buildings already built. Buildings may enable "developing technologies" in them, but also the dependencies between building types can be modeled by technology trees.

Technology trees serve many purposes. According to Sid Meier, undoubtedly the most famous computer game designer in the western world, the addictiveness of *Sid Meier's Civilization* (MicroProse, 1991) is at least partially due to "interesting decisions" [10]. A tech tree is certainly one of the most obvious ways to provide a player with possibilities to make such decisions. In addition to functioning as an upgrading system, tech trees can be used to represent technological history, as release mechanisms, and as narrative tools [3]. The decisions made by a player are often goal-oriented: the overall goal is typically victory, but acquiring technologies sets also short-term goals. Even if a technology does not have any significant impact on the gameplay, having it developed may, nevertheless, be something to brag about (see the article by Gazzard [11] concerning different rewards). As Huizinga ([12], page 50) put it, "in all games it is very important that the player should be able to boast of his success to others."

3. On the Demand for Monitoring Tech Tree Quality

Various properties affect the observed quality of a technology tree, and many common defects may reduce it. In order to find features typically considered problematic, we have been conducting an informal (unpublished) survey since 2012 by observing several WWW discussion forums. Because of the nature of the data, they may be biased; they have been acquired from a (somewhat arbitrary) set of forums, some of which concentrate on specific games (the sites we have been monitoring include, e.g., http://www.quartertothree.com/game-talk/, http://forums.2k.com/, http://forums.civfanatics.com/, http://www.gamespot.com/forums/, and http://forums.steampowered.com/forums/). Occasionally, also interpreting and categorizing the actual problems based on web publications may be problematic. Because of these difficulties, we do not announce any exact results, but only claim that by analyzing hundreds of writings (by various authors) we have found several problems that seem significant based on the frequency of them being mentioned or otherwise. We leave it as future work to validate the importance of the identified defect types.

Based on this survey, our experience, that of some of our fellow gamers, and WWW articles found using search engines, the most typical problems concerning existing tech tree implementations in real games seem to include

(i) too small or too large number of technologies,

(ii) too few requirements for developing technologies (allowing bypassing techs in an undesirable way),

(iii) too obvious *strong paths* (not encouraging to explore the tree, but only to exploit the known efficient strategy),

(iv) poor balancing (temporal or resource-wise),

(v) too limited possibilities to explore the tech tree during a game,

(vi) typically fixed and rigid (predefined) structure without any possibility of temporal variation or, for instance, variation between different game instances,

(vii) meaningless technologies (with only minor effects not motivating to advance in a tech tree),

(viii) requirements that do not make sense semantically, thematically, or historically.

The list presented here is not intended to be generally comprehensive. Grievances concerning visualization, aesthetics, and interaction issues are omitted (although common), because our focus lies on the structure and functionality, not on user interface (UI) details.

Of course, the quality of a tech tree is a highly subjective matter, and adjusting tech tree properties is all about making tradeoffs. However, in order to make justified adjustments to any direction, one must be able to evaluate tree properties. Common concerns serve as a good guideline for deciding which features are important. Balancing tech trees in terms of time and (other) resources is a generally meaningful problem. In this paper, we focus on such balancing, since somewhat objectively measurable tech tree features affecting the issue can be found. Balancing can be either *internal balancing* (within a tech tree) or *balancing between different tech trees*.

One typical central function of a technology tree is to control game flow with respect to time; tech trees are used for so-called gating and for controlling the narration or the overall "big picture game experience." Therefore, the temporal aspect cannot be overlooked, when analyzing a tech tree. By internal balancing we mean a process of modifying a tech tree for making it behave internally in a more favourable, balanced manner. If two technologies are supposed to be relatively on the same requirement level in the tech tree (e.g., they could belong to the same historical period), more or less same amount of efforts ought to be required for achieving them.

Typically, such efforts manifest themselves as gathering and spending *resources*. Time can be a resource type itself, but also cumulating assets, like *science points* or *energy*, can be used. In a more general characterization, of course, also noncumulating assets besides time can be allowed. However, in practice, such are seldom (if ever) used.

If a player is free to develop a technology, the time consumption for obtaining it depends often quite directly on the other resource requirements and the resource income at the corresponding game situation; a certain amount of resources must be accumulated and allocated for the development task in order to acquire the desired tech. Some additional time can be spent in carrying out the development task itself, in particular, if all the needed resources must be collected in advance.

The means and methods of gaining resources vary. Resources can be, for example, won in war, collected by peasants from mines, produced in factories, or gained by research work. However, the resource income can often be estimated as a function of game time, regardless of the exact earning method. When proceeding in a balanced tech tree, the requirements for developing each technology should be in accordance with the desired temporal flow (or "resource

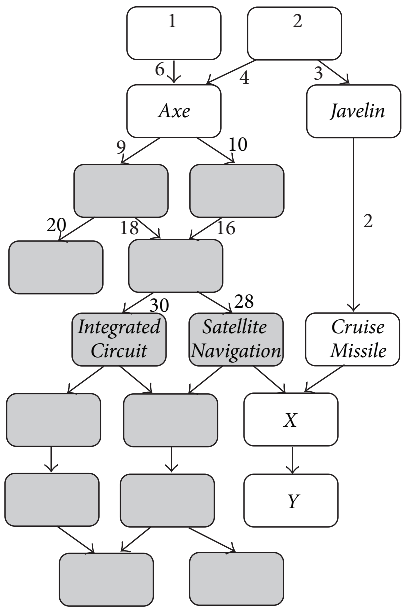


Figure 1: A problematic technology tree. Rounded rectangles depict technologies. They are of Boolean nature: either one possesses a technology, or not. The dependencies (marked with arrows) are of type OR. The numbers indicate costs in *gold nuggets* for advancing in the tech tree along the arrows. The costs for obtaining *entrance nodes* (ENs, the nodes representing the technologies that are initially available for development) are marked inside the corresponding rectangles.

flow”) so the game proceeds smoothly and is gated as intended.

Consider the illustrative tech tree of Figure 1. Both *Axe* and *Javelin* are meant to be easily achievable primitive technologies. The minimum costs for obtaining them are of the same magnitude (six and five gold nuggets, resp.). Also *Integrated Circuit*, *Satellite Navigation*, and *Cruise Missile* are intended to be mutually “on the same level”: they represent rather advanced technologies of the same historical era. All the three have powerful game effects, although this is not directly visible in the figure. However, via *Javelin*, *Cruise Missile* can be developed by using only seven gold nuggets. On the other hand, the minimum cost for getting *Integrated Circuit* is 62 nuggets, and that of *Satellite Navigation* is 60 nuggets. These two costs are relatively close to each other, as they should be, but the cost for obtaining *Cruise Missile* is significantly lower. Such magnitude difference implicates that the design is faulty. In this case, the problem is, naturally, the overly inexpensive technology *Cruise Missile*, which would, based on its achievability, belong to the same technological level with *Axe* and *Javelin*, but thematically, and based on the effects of the techs, this does not make much sense.

The problem with the design is also a flow problem; one should not be able to achieve an advanced technology like *Cruise Missile* too early; not only would such a feature be narratively awkward, but it might also manifest itself as a problem of strong paths, clearly superior routes to follow. Given the chance to obtain a powerful technology at an early stage via a certain route, probably that route is practically always chosen. The strong path problem can be alleviated by good internal balancing. If different techs are supposed to represent the same requirement level, the corresponding costs for obtaining them should be close to each other. Hence, a more balanced tech tree (in this sense) can be obtained by smoothing out differences between the costs of such techs.

In a balanced tech tree, the problem may only arise if the original assumption of the suitable requirement level for a tech is wrong. Looking from another perspective, effects of a tech should always be suitable for and on a par with the intended requirement level. Therefore, adjusting a tech tree in order to fix the problem requires estimates for game impacts of technologies involved. These estimates should be based on proposed requirement levels and offered game-dependent benefits.

So, internal balancing deals with a single tech tree, trying to make it better. By balancing between different tech trees, on the other hand, we mean a process aiming to make different tech trees behave similarly enough. This kind of balancing is important, for instance, when creating various tech trees to be used by different factions (e.g., nations) competing against each other. The common reason to use individual tech trees for the factions is the desire to clearly distinguish between them. Therefore, the dissimilarities between the tech trees should not be only cosmetic (e.g., different tech names): there should be actual variation in advantages, disadvantages, options, and meaningful strategies. However, the feeling of fairness in a game is important [13], and normally all the playable factions should be able to win. The playing should be enjoyable and meaningful. Therefore, none of the tech trees ought to be strongly underpowered or overpowered despite their differences.

4. Indicators for Characterizing Technology Trees

To make it easier to discuss measuring tech tree features, let us define some quality (and other) indicators. Let T be a tech tree with n distinct technologies. For simplicity, we assume here that there is only one resource type of interest, r, to be considered at a time. If there are several resource types affecting the situation, the indicator values can be defined separately with respect to each of them.

Useful global indicators describing the properties of T as a whole are at least

(i) its *size*, which we consider in terms of distinct technologies (and not the number of edges, as is the graph-theoretical convention), so here it would be n techs,

(ii) *tree requirement* (TR), the minimum amount of resources of type r needed to get all the n distinct technologies in T developed,

(iii) *average resource consumption for acquiring technologies* (ARCFAT),

(iv) *expected final tech coverage* (ETC),

(v) *average branching factor* (ABF) of T.

In order to define ARCFAT, a measure of local nature is needed. We call it *resources expected to be needed for acquiring a tech locally* (REAL). ARCFAT is the arithmetic mean of the REAL values of all the techs in T.

The REAL value concerning the development of technology t is the arithmetic mean of its local costs (in r). These costs correspond to the parent combinations that can allow (and are here assumed to allow) the development. In a simple (deterministic) case, a tech has only a single fixed local cost value, and the averaging over alternative parent combinations leading to the tech is unnecessary. In a more complicated case, the REAL value could be affected by, for example, randomization.

An interesting special case is REAL in respect to time (that we consider here as a resource among others). In this case, the REAL value for technology t is effectively the average of the expected development time requirements corresponding to the possible parent technology combinations allowing t to be developed. Such a measure can be straightforwardly used, for instance, to facilitate planning and constructing a tech tree, when targeting for some desired (expected) game duration.

The ETC value for T in a game featuring sequential technology development and technologies of Boolean development statuses can be defined as $e/(m \cdot n)$, where e is the expected gain of resource r of a game and m is the ARCFAT value of T. The ETC values are useful, when striving to tackle the problem of too limited possibilities to explore the tech tree during a game, or more generally, when trying to provide the players with the possibility to explore as much of a tree as desired in a typical game.

Knowing the exact ABF of a tech tree is usually not needed in time or space complexity analyses of the algorithms manipulating tech trees due to their modest sizes and the fact that the ABFs are typically small. However, one can draw conclusions on the structure and nature of a tech tree based on the ABF value of it. If the value is high, one probably has to really keep making choices frequently, but in the case of a low value, the tech tree probably has almost linear paths, and the focus is more on selecting the path(s) to follow and changing it when appropriate.

The values of the global indicator measures can be used for rough comparisons between tech trees. They are also useful in improving a single tech tree or designing such (and related game properties) from scratch. To clarify this, let us assume that we are creating a tech tree T and have formed a prototype with the following known (measurable) indicator values: ARCFAT = 350 r, TR = 5000, and size = 20 techs. We want ETC to be around 90 percent. Resources of type r are somehow produced or gathered during the game, and we want to define the details for the resource system. A simple estimate for the total amount of resource type r that one should be able to gain during a game can be obtained simply by taking the TR value and multiplying it by the desired ETC. In this case, the estimate is $0.9 \cdot 5000\,r = 4500\,r$.

Typically, the first technologies to obtain are relatively inexpensive, and the costs increase when advancing in a tech tree. Hence, technologies which are eventually not obtained in a game probably include many of the most expensive ones. Therefore, the estimated value might be more than enough. However, generally using the ARCFAT for calculating the lower bound in order to define the desired resource gain is a better solution. In our example, there are 20 techs, and on average each of them requires 350 r to be developed. Hence, an ARCFAT-induced lower bound in this case would be $0.9 \cdot (20 \cdot 350\,r) = 6300\,r$ (during a typical game).

Often, it is a good idea to give players some extra resources to let them have more freedom to proceed via nonoptimal routes; all the players are not interested in optimizing exactly, when playing. Too scarce resource supply may lead to players never proceeding to expensive techs, thus rendering them unnecessary. On the other hand, of course, if there is an upper tech coverage limit defined that is not to be exceeded, resources should not be granted too generously, or the number of achievable techs should be limited in another way.

Consider another example. Let T and U be tech trees intended to be used for the same purpose by two different mutually competing factions (one tech tree for each of them). If the expected curves of income for these factions (in r, as a function of wall time or play turns) are similar, one can assume that the total amounts of r gained by the factions during a game are near to each other, as long as the game starts and ends at the same time for both players. Typically, the sizes of T and U should be about the same. This, of course, assumes that ARCFAT values and the desired ETC values are also near to each other. In some rare cases, this assumption might not hold, but nevertheless, the measures are useful and help in making desired adjustments. The ARCFAT values should be adjusted suitably for maximizing the enjoyment of advancing in a tech tree. It is frustrating to have to wait for a very long time for even simple technologies, but on the other hand, a game most probably has also other contents besides making tech choices, so all the gaming time cannot be used for that. To summarize, as a rule of thumb, the sizes and the ARCFAT values of T and U ought to be similar. If this is not the case, adjustments to at least one of the tech trees are probably needed.

A simple, but quite powerful, way to roughly adjust the global properties of a tech tree is to choose a suitable coefficient, c, and then, for each tech t, simply to multiply the needed resource requirements by c (for all the possible parent technology combinations allowing the development of t). This affects directly the REAL values and thus also ARCFAT measures. Moreover, via ARCFAT, the effect of the operation reaches to ETC. Game length can also be manipulated this way via REAL and ARCFAT values for time resource.

5. Indicators for Characterizing Technologies

Besides REAL, there are also other important indicator measures of a local nature to be found. Let t be a technology in a tech tree T with a total number of n distinct technologies, and let Q be the set of those technologies in T, into which there are nontrivial paths from t. Interesting local indicators for the single node t include at least

(i) *gating indicator number 1* (GIN1) = $a/(n-1)$, where a is the cardinality of the set Q,

(ii) *gating indicator number 2* (GIN2) = $b/(n-1)$, in which b is the number of technologies in Q that require t to be developed prior to their own development (i.e., there are no alternative routes leading to them and bypassing t),

(iii) *optimal cost* (OC) = the minimum amount of resources of type r needed to get t developed without any prior development in T,

(iv) *requirement ratio* (RR) = OC/TR (assuming TR differs from zero).

These local indicators can be used, for instance, to estimate which technologies should receive extra attention. As an example, a high value of GIN2 for a technology t indicates that it should be possible, maybe even easy, to obtain t, because otherwise a large portion of the existing tech tree is effectively not needed for anything.

The tech tree of Figure 1 has 19 nodes in total, so that $n - 1 = 18$. From node *Axe* 14 nodes (the grey ones plus X and Y) can be reached. Thus, GIN1 for *Axe* is $14/18 = 7/9$. The value is relatively high, as expected: it is easy to check visually from the figure that only a small portion of the tech tree is inaccessible via *Axe*. Technologies X and Y could be developed without *Axe* via *Javelin* and *Cruise Missile*. Therefore, these two technologies are not counted to number b when calculating GIN2 value. The 12 grey nodes in Figure 1 represent technologies that require *Axe* prior to being developed themselves. Hence, the indicator GIN2 has the value $12/18 = 2/3$, which is less than the value of GIN1 but still rather large; the grey nodes alone make a large portion of the tech tree, and *Axe* is a mandatory node in all the possible paths leading to them.

OC values were already calculated for the argument of *Cruise Missile* being too cheap to achieve in the example of Figure 1. For instance, the OC value for *Satellite Navigation* is 60 gold nuggets by minimization over alternative routes leading to its development. In order to demonstrate TR and RR, let us use even smaller example tree illustrated in Figure 2.

Let us determine the TR (with respect to r) assuming OR-type dependencies. This problem resembles closely the graph-theoretical minimum directed spanning tree problem, but with tech trees instead of one root there can be several ENs. To obtain A, B, C, D, E, G, and I, there are no choices to be made: the cost for getting these nodes developed is $(1 + 2 + 5 + 8 + 3 + 10 + 24)\ r = 53\,r$. We can consider them to have been paid for and developed. Now, the optimal choice for developing F is via D using $7\,r$, and for developing H (after developing F) it is via F using $15\,r$, so that TR = 53 + 7 + 15 = 75. Now, the OC value (still with respect to r) for, say, F, is $1 + 5 + 9 = 15$, so RR value for F is $15/75 = 1/5$. On the other hand, the RR value of I equals $(1+8+10+24)/75 = 43/75$. This value is considerably larger than $1/5$, which indicates that I is more expensive than F. Therefore, I is suitable to be the more advanced technology of the two.

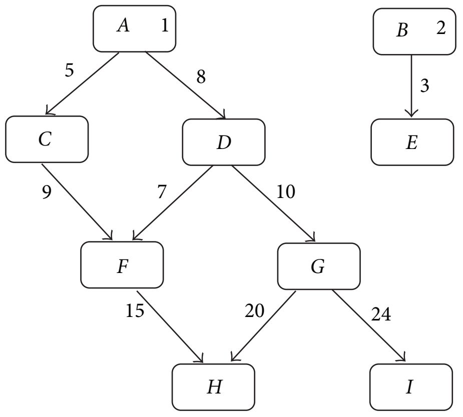


Figure 2: A simple example tech tree depicted using the conventions established in Figure 1. The costs are expressed in resources of type r.

6. Measuring the Indicators: Implementation

In a previous work [2], we proposed a generic approach for implementing technology trees to obtain diverse benefits. We also introduced a software tool for creating technology trees easily and partially automating the process of tech tree creation. We have now augmented our software tool, TTT, with a capability of measuring important tech tree properties.

Our previous paper [14] discusses temporal layer analysis. The basic idea is presented in Figure 3. A technology tree is converted into a forest consisting of real (graph-theoretical) trees by duplicating nodes (representing technologies) as necessary. Then, the nodes, having visual representations, are arranged topologically in a way that makes it easy to analyze tech tree properties visually and find problems. In Figure 3, there are six different time layers corresponding to specific points in time, and there is a node corresponding to a technology on a layer if and only if it is possible to get the tech developed at the time represented by the layer. Time is assumed to be the only resource type in use, the development status of each node is Boolean, and having one developed parent is a sufficient precondition for developing a tech node in this setting (that is, the dependencies between technologies are treated as OR-type ones). The progress is assumed to be sequential and without any slack time.

We implemented a modified version of this conversion in TTT. The idea is still mostly the same, but in order to limit the amount of nodes to be generated and drawn, we only consider possible paths leading to a given technology, not necessarily the whole tech tree. On the other hand, the topologies are presented for all required resource types, not only time. We call this approach *generalized time layer approach* (GTLA).

Resource demands (of the resource types under scrutiny) are accumulated into each node along different routes starting from the ENs. The resource requirements of each node are resolved for each requested resource type, and the values are "pushed" top-down to (new) nodes corresponding to the children in the original tech tree. Then, these newly created nodes are processed similarly, and the process continues, until all the nodes have been exhausted.

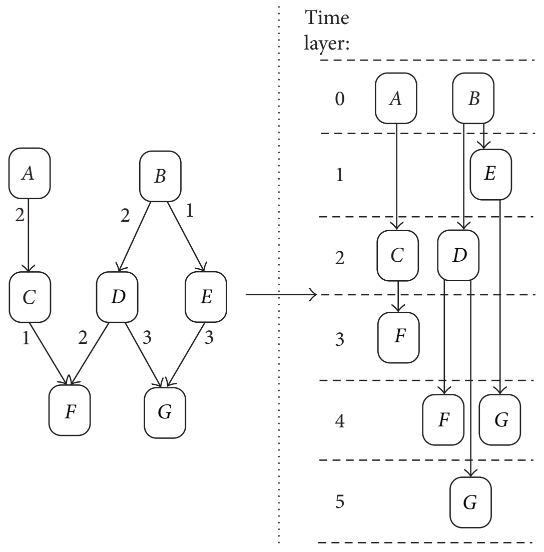


Figure 3: Generating a forest (on the right-hand side) based on a tech tree (on the left-hand side) and representing the optimal times to achieve technologies via different routes topologically. The numbers next to the tech tree edges are corresponding time requirements.

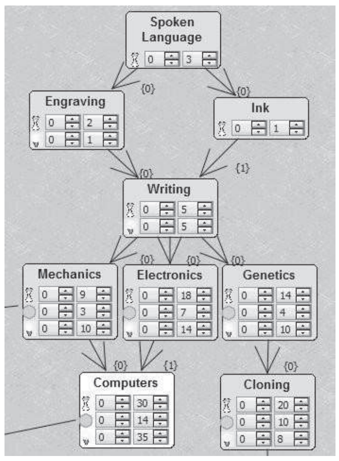
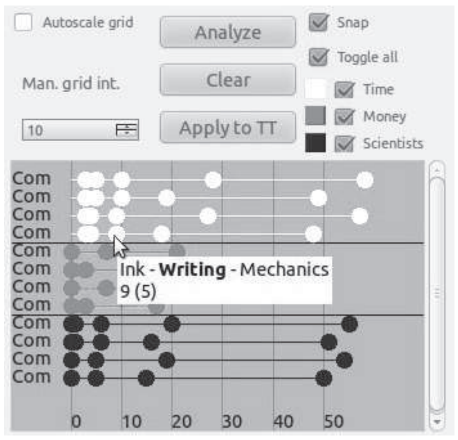


Figure 4: A part of a simple example tech tree. A partial screen capture from TTT.

Consider the tech tree structure of Figure 4 and applying GTLA to obtain possible paths leading to *Computers*. In Figure 5, the graphical presentation of these paths (in terms of different resource type requirements) is shown, as presented in the TTT UI.

From the graphical representation, it is easy to detect, for instance, the paths leading to optimal total resource consumptions in terms of the resource type of interest as well as the OC values themselves. Even simpler representation, with fewer plot points to consider, can be obtained by aggregating different resource types into a more general resource requirement plot. Different weights for distinct resource types can be used.



Autoscale grid
Man. grid int. 10
✓ Snap
✓ Toggle all
✓ Time
✓ Money
✓ Scientists

Ink - **Writing** - Mechanics
9 (5)

Figure 5: Possible technology paths to *Computers* in the tech tree of Figure 4 plotted for three resource types. The paths are depicted as horizontal successions (from left to right) of circles, each representing a technology. A partial screen capture from TTT.

So, GTLA can be used, among other things, to determine OC values. However, the method in its simplicity is only suitable for tech trees, in which any technology can be developed, if allowed by any parent technology. This is (typically) true for tree-like tech trees, in which a technology has at most one parent, and for tech trees based on OR-type dependency relations, like the one in Figure 4. In the example tree there are two technologies, *Writing* and *Computers*, with two parents each. However, having a single parent technology developed suffices to develop the corresponding child. In the TTT UI (see Figure 4), this fact is visible as the identifier sets attached to edges starting on different parents, {0} and {1}, containing different elements.

The idea is that the identifiers, *and-bunch identifiers* (ABIs), of the edges arriving from the parent nodes determine sufficient edge sets that allow a tech to be developed. In order to develop it, for an ABI present in the ABI set of some incoming edge, it must hold that all the incoming edges having this ABI in their ABI sets must *fire* (accept the development) simultaneously [2]. In this paper, we assume for simplicity that an edge fires, whenever the technology corresponding to its start node has been developed. This is typically the case with real-world tech trees with binary tech development statuses.

In order to illustrate the idea behind ABIs, let us modify the example tech tree of Figure 4 a bit. In the tech tree of Figure 6, there are AND-type dependencies involved. *Spoken Language* is not anymore a prerequisite for *Engraving* or *Ink*, and to be able to develop *Writing*, one must have in addition to *Spoken Language* either *Engraving* or *Ink* developed. The freedom to determine several sufficient AND-type parent groups for a technology (and the ABI set characterization) adds considerably to the overall flexibility of tech trees and facilitates using thematically sensible dependency structures.

GTLA proceeds locally from ENs towards the leaf techs without considering all the necessary requirements (parents) when used with this tech tree. Hence, it gives too optimistic

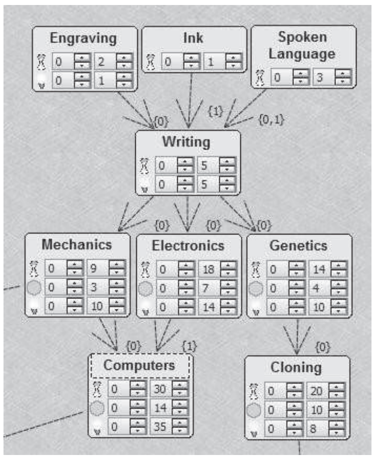


FIGURE 6: A modified example tech tree. A partial screen capture from TTT.

```
(1) function approximate_OCs():
(2)     $C \leftarrow$ the set of ENs
(3)     while $|C| > 0$ do
(4)         $N \leftarrow \emptyset$
(5)         for each $n$ in $C$ do
(6)             handle_node($n$)
(7)         swap($C, N$)
```

ALGORITHM 1: A function for going through a tech tree appropriately. Executing this results in having OC estimates for each node, as well as estimated optimal ancestor sets and optimal ABIs.

OC results in this case. Therefore, a more general method for coping with this kind of more complex tech trees and estimating their OC values and optimal routes was developed and implemented in TTT. Now, each tech may have several different parent subsets allowing its development, and each such subset may have its own respective resource requirements. In other words, a technology may have several alternative price tags, with the actual cost to be paid depending on the developed prerequisite tech combination. The OC estimation algorithm, usable with acyclic tech trees, is presented in Algorithm 1 as the pseudocode function approximate_OCs. The function handle_node, used by approximate_OCs, is given in Algorithm 2.

The functions do not aim at plotting the possible paths leading to a tech, since their number in nontrivial tech trees can be large because of different possible orders of developing prerequisites. Instead, for every technology in the tech tree, the OC value is estimated and a corresponding ABI set, via which one probably achieves the tech optimally, is determined. Estimated optimal route for developing the technology of interest is available in the form of the set of optimal ancestors (*opt_anc*) for each node after executing approximate_OCs.

To simplify the presentation in Algorithms 1 and 2, the resource type under scrutiny has been omitted in the pseudocode, and it is assumed that all the relevant operations

```
(1) function handle_node($n$):
(2)     $n.counter \leftarrow 0$
(3)     for each $a$ in $n.abi\_sets$ do
(4)         $R \leftarrow \emptyset$
(5)         $abi\_solved \leftarrow$ true
(6)         for each incoming edge $e$ of $n$ with ABI $a$ do
(7)             $p \leftarrow e.start\_node$
(8)             if $p.solved$ then
(9)                 $R \leftarrow R \cup \{p\}$
(10)            else
(11)                $abi\_solved \leftarrow$ false
(12)        if $abi\_solved$ then
(13)            $n.counter \leftarrow n.counter + 1$
(14)            $A \leftarrow \bigcup_{p \in R} p.opt\_anc$
(15)            $s \leftarrow n.cost(a) + \sum_{v \in A} v.cost(v.opt\_abi)$
(16)            if $s < n.opt\_cost$ then
(17)                $n.opt\_cost \leftarrow s$
(18)                $n.opt\_abi \leftarrow a$
(19)                $n.opt\_anc \leftarrow A \cup \{n\}$
(20)    if $n.counter = |n.abi\_sets|$ then
(21)        $n.solved \leftarrow$ true
(22)        for each $c$ in $n.children$ do
(23)            if not $c.solved$ then
(24)                $N \leftarrow N \cup \{c\}$
(25)    else
(26)        $N \leftarrow N \cup \{n\}$
(27)        for each $p$ in $n.parents$ do
(28)            if not $p.solved$ then
(29)                $N \leftarrow N \cup \{p\}$
```

ALGORITHM 2: A function used by approximate_OCs for handling an individual node and making the necessary additions to the next node layer.

and variables are with respect to it. The functions could, of course, take resource type parameters and use them in bookkeeping.

The basic idea is to start with the ENs as the active set (layer) C of nodes to be handled. For each node in C, the function handle_node is executed. It forms the layer N to be handled during the next round, and the execution proceeds this way a layer after another, until the values have been solved for the whole tech tree. The algorithm stops, because eventually all the nodes in a tech tree of a finite size will be solved, and thus N will be the empty set after the loop of the lines (5) and (6) of approximate_OCs. Each call to handle_node either marks a node solved and adds its children to N or adds its unsolved parents to N, which prevents looping infinitely.

The obtained values are only estimates, because *v.opt_abi* (handle_node, line (15)) is not necessarily the optimal ABI with respect to n but only v itself. However, in practice, the estimates obtained this way for real tech trees are typically accurate, and making the simplifying assumption of global optimality of the determined optimal ABIs reduces the required computing time and space usage into a sensible level. The optimal ABI values can be used for determining the costs

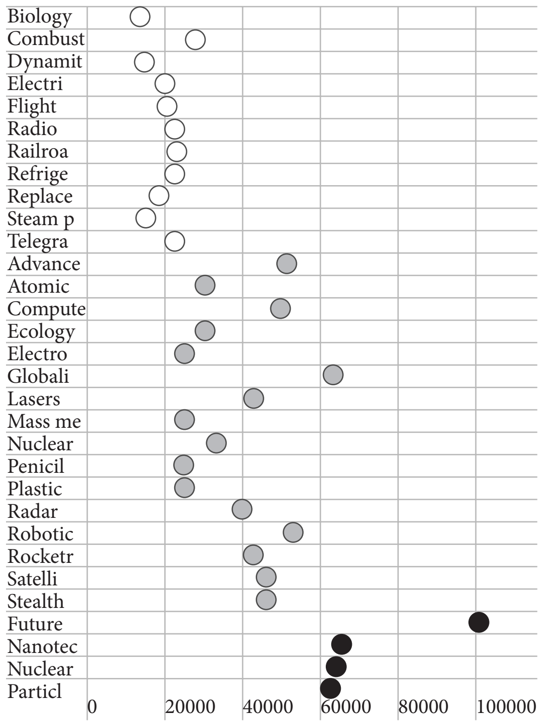


Figure 7: Minimum science requirements for the technologies of the last three eras in Civ 5, as plotted by TTT. (In order to improve the image quality for publication, the figure was redrawn based on a TTT screen capture.) Distinct techs are represented on their corresponding rows, and the horizontal locations of circles (their midpoints) represent the OC values of the corresponding techs. OC values increase linearly along the horizontal axis from left to right, as indicated.

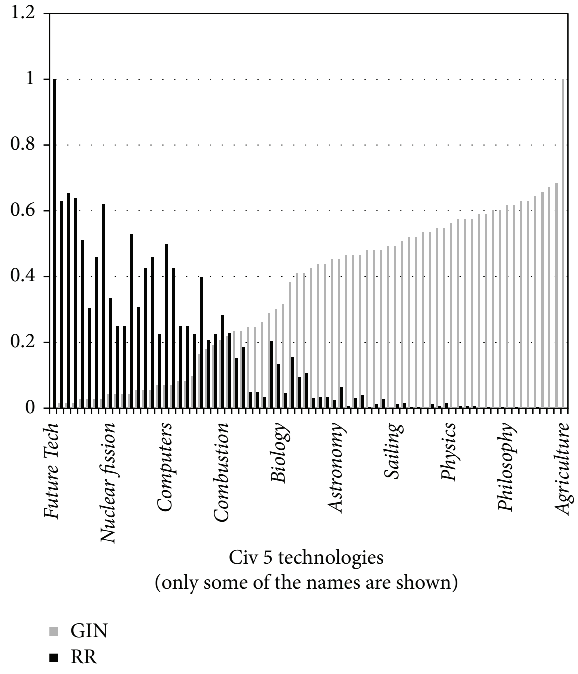


Figure 8: GIN (= GIN1 = GIN2) values and corresponding RR values of the technologies in the Civ 5 tech tree.

for individual techs, when estimating the TR value for a tech tree.

7. Measuring a Real-World Tech Tree

To test our tool improvements, we created a TTT representation of the tech tree of Civ 5, because it is a good representative of technology trees used in contemporary digital games. The game is rather recent, but the technology tree is used in a conventional fashion. The Vanilla version with *science* cost estimates taken from a web source [15] was used. We ran the algorithm of Algorithm 1 for the tech tree. Because of the structure of it, the exact OC values were obtained. The operation took time of 16 milliseconds with a standard desktop computer (Intel Core i5-3470 running at 3.20 GHz, 16 GB RAM, 64-bit Microsoft Windows Enterprise).

In Figure 7, the *science point* (or "*beaker*") OC results have been plotted for the technologies of the last three eras of the game: Industrial Era (white), Modern Era (grey), and Future Era (black). These kinds of plots are beneficial for checking visually that technologies are as easy or hard to obtain as they should be or spotting problems concerning internal balancing. As the eras in Civ 5 represent distinct intervals in the (temporal) continuum of overall technological development, it would make sense for technologies categorized to belong to the same era to have minimum science requirements of the same magnitude.

As can be seen in Figure 7, the requirements for the Industrial Era technologies do not differ much from each other. Modern Era has more variance in this sense, and the technology *Future Tech* of the Future Era seems to be clearly an outlier that should be checked carefully, if the technology tree was under development. In this case, however, the huge amount of science points required is due to the unique nature of *Future Tech*; it can only be developed after developing all the other techs.

Besides OCs, we let TTT also compute other interesting (local and global) indicator values for the technologies of the Civ 5 tech tree. There are no surprises in the results. GIN1 values are equal to corresponding GIN2 values, because Civ 5 tech tree does not offer alternative routes to achieve technologies. In Figure 8, the technologies are ordered into an increasing order based on their GIN1 values (= GIN2 values), marked simply as GIN in the figure. Also, the corresponding RR values are shown in the same figure.

When GIN values increase, RR values tend to decrease. This makes sense, because near ENs there are (typically, and not only in this case) inexpensive technologies, via which one has to proceed in order to access the other parts of the tech tree. On the other hand, the final technologies typically require lots of resources and they limit access to only few even more advanced techs.

The TR value for beakers is 100,487, the ARCFAT value is 1,357.93, and the ABF is approximately 1.49. TR and ARCFAT do not tell very much in a case of a single tech tree without anything to compare to, but based on the modest ABF value and the additional fact that the tech tree is connected, one can conclude that it is also rather deep.

It is worth highlighting that the indicator values obtained might be rather different, if measured from a patched version of the game (possibly augmented with downloadable content packages), as the tech tree properties have changed since the Vanilla version. The fact that the tech tree has been modified several times demonstrates that it is really an important part of the game. As mentioned, used costs are also only approximations; the exact in-game costs depend, for instance, on the number of cities the player has.

8. On Correcting a Tech Tree

Whenever measurements indicate problems, taking corrective steps can be either easy or tedious. With TTT, adjusting the tech tree structure and modifying local properties, like resource requirements and dependency relations of a single technology, are easy, since the tool has been created for effortless technology tree manipulation. The novel features make such manual adjustments even easier. Especially worth mentioning is the fact that the GTLA view (see Figure 5) allows (imposing necessary restrictions) the user to drag technologies along their respective paths and to commit the corresponding resource requirement changes into the actual tech tree presentation of the program, from which functional technology tree code is generated automatically, when desired. This way the user can see the effects of planned changes to other technologies visually before actually applying them.

As far as global adjustments (affecting the characteristics of a tech tree as a whole) are considered, our tool so far supports setting the desired TR value, based on which the system is capable of adjusting the tech tree multiplicatively. The modification is performed simply by determining and applying a suitable multiplier for all the technology costs in the tech tree.

9. Conclusion

In this paper, we have introduced indicator values and discussed algorithms and our implementation for analyzing technology tree features for proper adjustments. The implementation was also tested with a real, popular computer game, and thus its capability to produce and visualize data, which we strongly believe to be useful, was verified.

As future work, more general, important, and measurable tech tree features should be pointed out, and corresponding measuring and correcting procedures ought to be implemented. Moreover, the automated analysis features currently present in our software should be improved. Also, a considerable number of real-world tech trees ought to be analyzed in order to find good practices and typical tendencies to guide in the further development and fine-tuning of adjustment automation procedures.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] N. Hallford and J. Hallford, *Swords & Circuitry: A Designer's Guide to Computer Role-Playing Games*, Stacy L. Hiquet, 2001.

[2] T. J. Heinimäki and T. Elomaa, “Facilitating technology forestry: software tool support for creating functional technology trees,” in *Proceedings of the 3rd International Conference on Innovative Computing Technology (INTECH ’13)*, pp. 510–519, London, UK, August 2013.

[3] T. Ghys, “Technology trees: freedom and determinism in historical strategy games,” *Game Studies*, vol. 12, no. 1, 2012.

[4] P. Tozour, “Introduction to Bayesian networks and reasoning under uncertainty,” in *AI Game Programming Wisdom*, S. Rabin, Ed., pp. 345–357, Charles River Media, 2002.

[5] E. Schaefer, “Blizzard entertainment’s Diablo II,” in *Postmortems from Game Developer*, A. Grossman, Ed., pp. 79–90, CMP Books, CMP Media LLC, San Francisco, Calif, USA, 2003.

[6] D. Morris and L. Hartas, *Strategy Games*, Ilex Press Ltd, Lewes, UK, 2004.

[7] M. Barton, *Dungeons and Desktops: The History of Computer Role-Playing Games*, CRC Press, 2008.

[8] T. J. Heinimäki, “Technology trees in digital gaming,” in *Proceedings of the 16th International Academic MindTrek Conference (AMT ’12)*, pp. 27–34, October 2012.

[9] T. Owens, “Modding the history of science: values at play in modder discussions of Sid Meier’s CIVILIZATION,” *Simulation & Gaming*, vol. 42, no. 4, pp. 481–495, 2010.

[10] R. Rouse III, *Game Design: Theory and Practice*, Wordware Publishing, 2nd edition, 2005.

[11] A. Gazzard, “Unlocking the gameworld: the rewards of space and time in videogames,” *Game Studies*, vol. 11, no. 1, 2011.

[12] J. Huizinga, *Homo Ludens—A Study of the Play-Element in Culture*, Beacon Press, 1971.

[13] E. Adams, *Fundamentals of Game Design*, New Riders Publishing, 2nd edition, 2009.

[14] T. J. Heinimäki, “Considerations on measuring technology tree features,” in *Proceedings of the 4th Computer Science and Electronic Engineering Conference (CEEC ’12)*, pp. 145–148, Colchester, UK, September 2012.

[15] List of technologies in Civ5, 2014, http://civilization.wikia.com/wiki/Technologies%28Civ5%29.

Publication VI

T. J. Heinimäki and T. Elomaa. Augmenting Technology Trees: Automation and Tool Support. In *Proceedings of the 7th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games 2015)*, pages 68–75. Skövde, Sweden, September 2015.

Augmenting Technology Trees: Automation and Tool Support

Teemu J. Heinimäki and Tapio Elomaa
Department of Mathematics
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
Email: {teemu.heinimaki, tapio.elomaa}@tut.fi

Abstract—**In this work we tackle the rigidity of predefined technology trees common in many digital games. Technology trees have been a well-understood concept in the industry for approximately two decades. Unfortunately, so far they have mainly been considered as fixed structures. Some attempts of adding ostensible temporal variability to them have been made before by the means of simulating the effect. Our approach, on the other hand, aims to create a stepping stone towards true runtime technology generation for (both serious and leisure) games in order to improve them. We present potentially useful ideas, constructs, and methods to achieve this goal. Initial observations on our test implementation are also presented.**

I. Introduction

Technology trees, also known as *tech trees* (or *skill trees*, *talent trees*, etc.) are the predominant way of providing development alternatives within digital games. These structures guide and limit possible player-made choices and the overall flow of the game. Moreover, they provide short-term goals for players. Choices offered to the players are known to be essential in meaningful play [1].

In computer role-playing games, usually the protagonist (player character) is improving its skills, piling up money, attaining arms, gaining in knowledge, or developing in some other way as the game progresses. In strategy games, development typically concerns a whole tribe, a nation, a species, or even a larger entity, like a galactic alliance. Concerning the apparent distinction between these development graph structure ("tree") types in different genres, it is basically just a question about whose operation and development does a particular tree guide. Therefore we find it justified to use the term technology tree for all such development-guiding structures.

A tech tree explicates what are the requirements for advancing in technical capability, engineering abilities, scientific knowledge, military power, financial wealth, or such. The tree models and keeps track on the overall development status using distinct *technologies* (also known as *techs*). Technologies can be developed individually, and the tree dictates their relationships and requirements for starting the tech development processes.

The downside of usual tech trees is their rigidity. They are preprogrammed without any ability to adapt to the player and without any variation in time or between different game instances. This paper advances and promotes the possibilities to rectify this issue.

We introduce our approach called CATFAT (Constructing Automatically Technologies For Augmenting Technology trees) to make tech trees more flexible and more interesting by automated content generation. The approach consists of specific kinds of classifications and representations of technologies and a tech tree augmentation method that actually uses them.

The main contributions of this paper are (i) presenting the general principles of CATFAT and (ii) demonstrating the practical usefulness of it, when implemented in suitable software tools. Secondary contributions are (iii) pointing out the common rigidity problem of technology trees and the difficulties related to tackling it and (iv) raising general awareness of the importance of technology trees as a topic for further studies.

We continue by a bit of more background and by explaining our motivation in Section II. A short review to the related work can be found in Section III, after which we introduce our approach in Section IV. Section V illuminates tool support implemented so far, and Section VI describes initial experiments that have been conducted. Section VII discusses different aspects of possibilities to generate technologies based on our observations. Finally, conclusions are drawn in Section VIII.

II. Background, Motivation, and the Problem Domain

Technology trees are essentially data-containing acyclic digraphs that are used to offer players possibilities to develop technologies.[1] Tech trees are used to guide and limit possible choices and the overall flow of the game and to offer goals for players. (The term technology tree can also be used to mean a representation of such a structure, e.g., in a pictorial form.) The idea of a tech tree is demonstrated in Fig. 1. Initially, there are two technologies, *A* and *B*, available to be developed. Technology *D* can be developed after obtaining *B*, but prior to developing *C*, both *A* and *B* are required.[2]

In practice, technology trees are designed manually, and once they have been implemented for a game, they do not change, barring possible corrections and modifications introduced in patches and expansions to the game. Even if a game

[1]We use the word technology quite liberally here. In this paper, the word is used synonymously with any node belonging to any technology tree.

[2]This paper only considers tech trees with such conjunctive prerequisites. In general, however, obtaining all the prerequisite technologies is not always necessary in order to develop a tech, but only some of them may suffice.

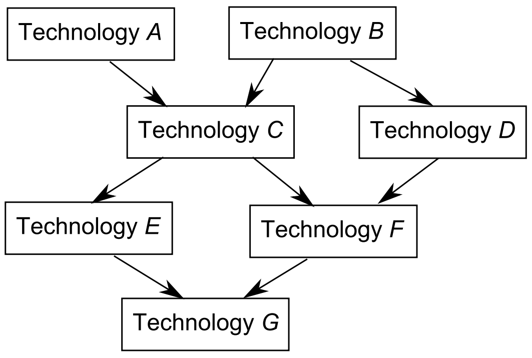


Fig. 1. An illustration of the basic idea of a tech tree.

uses a data-driven approach and defines trees in, say, text files instead of hard-coding them – thus making it easier to alter the game – tech trees are still essentially fixed between the modifications. Making tech trees easy to modify is a good practice, but does not remove the fundamental problem.

Rigidity is a problem because of its negative effects to the replayability and interestingness of a game [2]. Moreover, a fixed tech tree also implies having only a limited amount of technologies to be offered during a game. At some point, there simply may not be any technologies left to be attained.

Unfortunately, the solutions applied in practice are far from perfect and tend to restrict games severely. For instance, the number of turns in a game may be limited in order to force the tree to suffice. It is also possible to offer some generic "future tech" to be researched after all the "real technologies" have been developed, as is done in the Civilization games (*Sid Meier's Civilization* (MicroProse 1991) and its numerous sequels). In Fig. 1, *G* could be such a technology, researchable over and over again. However, this approach may lead to uninteresting gameplay after a critical point, because the central element of making the player to choose the next technology to be developed among several candidates is effectively removed.

There are, of course, lots of cases, in which there is no need to expand the tree to make it larger at the runtime. For instance, in strongly story-driven games the length of a game is determined by the narration, and trees of finite – and often quite modest – sizes can be built to fit the arc of the story. Therefore we acknowledge that good games of certain types can be implemented even with rigid trees.

Nevertheless, especially strategy games featuring large traditional tech trees and so-called sandbox games with considerable freedom to explore the world and proceed in the possible story would benefit from expandability of tech trees. The degree of actually using dynamic tech tree content generation might naturally vary – all the technologies could be generated during runtime, a fixed tree could be augmented only when necessary, or anything in between. In any case, it would be generally appealing to be able to generate techs on runtime and keep discovering specific, meaningful technologies even after exhausting the predefined tree. This is the motivation behind this work.

If one desires to produce better games, also capability to create temporal variance in tech trees is an important aspect to consider. Even if the tree size is fixed, technologies do not have to be. If the set of available technologies changes for each game, one cannot keep playing the game similarly, making the same choices time after time.[3] The whole tech tree does not necessarily have to be replaced; temporal variability in varying degrees can be offered by changing, adding, or removing available individual technologies in a "basic tech tree". Such tech tree–modifying operations can also be performed during the actual gameplay – not only when initializing the game. Combining the idea of temporal variability and runtime content generation means introducing some (pseudo)randomness into the generating process.

Often technology tree structures and technologies in them are fully known to players so that they can easily plan their technological advancement beforehand. If, on the other hand, one is interested in dynamically "inventing" possible areas of research always based on the pre-existing knowledge (the state of technological development) during the game, real runtime generation – instead of revealing preconstructed content – is an intriguing way to proceed, since this way the virtual process of development corresponds to the real one.

A central problem in automated technology generation is natural language: it is rather simple to generate technologies with somewhat arbitrary properties within some guidelines, but typical use of technology trees requires the technologies to have descriptive names (and possibly even more detailed descriptions of some length). Another major issue is also about representation; often technologies have corresponding visible instances, like moving units or buildings, in the game. This paper mainly focuses on the former problem, as it applies also to the abstract technologies without those "physical" instance requirements. Sometimes also graphical images are used in visual technology representations. Images might be generated, for instance, by combining elemental images corresponding to "basic natures" of techs and their property bonuses etc., but this is out of the scope of this paper.

III. Related Work

Procedural content generation (PCG) for digital games in its various forms has been recently studied rigorously. A recent survey [3] gives a decent overview to the field. Our approach can be seen as a PCG method specializing to generate technology trees, and it touches upon at least two PCG subfields identified in the survey: those concerning game systems and game design.

The approach presented in this paper fits in the category of domain specific language approaches to PCG, because the content generation in this case is based on a model defined by a language developed specifically for this purpose. Using the taxonomy of methods in PCG [3], our approach can be seen as one using generative grammars. The basic idea of creating content by "expanding" smaller description is typical to PCG algorithms [4].

[3]Making each game different by changing tech trees for each game instance would add considerably on the replayability value. Naturally this would, however, also annoy some players, so probably the feature should be optional.

Until recently, technology trees have been overlooked as interesting subjects for academic studies. As far as we know, this is the first paper about generating content to technology trees. The fixed structure and the assumption of technology trees or technological paths existing as such – without changes due to outside influence – have been criticized (see, e.g., reference [5]). However, practical remedial suggestions are hard to find. Typically tech trees are treated as totally deterministic constructs. Sometimes determinism is even included in the definition [6].

As the basis for our implementation of the approach put forward in this paper we use a prior software tool, Tech Tree Tool (TTT) [7], [8], created for, e.g., building, manipulating, and analyzing tech trees. The tool facilitates the process of defining the required constructs by offering an easy-to-use graphical user interface (GUI) and generating code automatically.

One possible way to create temporal variability to tech trees is to leave some technologies out of a large predefined technology tree (more or less randomly, often preserving essential "skeleton" technologies). This approach has actually been used in real commercial games – for instance in *Sword of the Stars* (Kerberos Productions 2006). An alternative way – the approach discussed in this paper – proceeds from the opposite direction and tries to generate new technologies (with some random factors involved) and thus expand a skeletal tree.

The problem of managing and directing a game based on user-made decisions has been discussed in previous papers, and frameworks have been presented (see, e.g., reference [9]). Our approach does not alter the game plot as such, but belongs in the wider scope of improving player experience by automation. We also acknowledge the possible benefits to gaming experience obtainable by making runtime changes to the game played based on the observations about the player (see, e.g., reference [10]). For instance, there are numerous ways to adjust the difficulty of a digital game dynamically [11]. At the moment, the suggested method to generate technologies in its simplest form does not explicitly include such functionality. However, it is easy to add considering the modeling and algorithmic choices made.

There have been several publications on generating game mechanics automatically based on user-generated content. For instance, a recent conference paper [12] discusses this topic and introduces a game-generating system. Our approach has a more strict focus on technology trees, but in some sense it also generates mechanics by defining a set of technologies with effects to be chosen at a given time via the tech generation. Our method is also designed to operate during runtime, though it can alternatively be used to help with planning tech trees beforehand when designing and creating a game.

Numerous studies focusing on existing games, their effects, and game design have been carried out during recent years. As nontechnical by nature they are only weakly related. In addition to (merely) entertainment applications, so-called serious games have gained considerable attention. They have various application areas [13], [14]; for example, games can be used in education [15] or in rehabilitation [14]. Our examples in this paper hint towards a leisure setting, but the CATFAT approach can as well be applied in "serious", non-leisure contexts using tech trees.

IV. The Proposed Approach

The main idea of the CATFAT approach is to generate new technologies based on the existing ones in order to make the game more interesting and replayable (see Subsection IV-C). Technologies can be generated either dynamically when running the game, when initializing a game instance, or beforehand when implementing the game.

Properties and categories are attached to technologies, since they guide the technology generation process (see Subsection IV-A) and thus are crucial. To be able to communicate the method clearly, the technologies are also divided into several types (see Subsection IV-B).

A. Categories, Properties, and Non-tech Modifiers

Developing a technology usually manifests itself in a game by enabling the corresponding player to use (or create), e.g., physical abilities, spells, units or other such (possibly abstract) concepts. Let us call these, in the lack of a better term, *products* of their respective enabling technologies. To clarify: technologies are goals for (technological, scientific, equipment-related, or some other kind of) development in a game, and products are the manifestations of the developed technologies.[4]

The CATFAT method attaches *properties* to technologies and their products. These properties reflect, e.g., physical measures, abilities, roles, and effect types in the game world to which the technologies or the products are directly related.[5] The properties are basically name-value pairs having typically numerical or Boolean values, via which they inflict functional effects in the game. There may also be faction-wide or global properties affecting the game more widely.

For instance, each faction in a game could have the property *loyalty* featuring, say, a value between zero and one indicating the general tendency of units to defect to a competitive party in situations offering possibilities to do so. In addition, each unit might have its personal loyalty modificator property, and the actual probability for a defection of a single unit occurring in a given game situation could be calculated based on these two property values.

The relation between properties and technologies is simple: developing technologies modifies property values, and property values define and modify constraints and effects for, e.g., technologies. For instance, achieving some popular form of government (a tech) in a game might boost the global *loyalty* value, and developing an institute for funding research work might increase the personal loyalties of scientist units.

In addition to using properties, another key element in our approach is categorization of techs and products based on an

[4]It is not always necessary to make a distinction, and sometimes the term technology is used to cover also products, but we find this a bit confusing habit and use two distinct words in this paper for clarity.

[5]In computer role-playing games, it is typical to use numerical statistics in order to track character development [16]. Here we are applying the idea more widely: measurable properties can be attached to other entities also, not only characters.

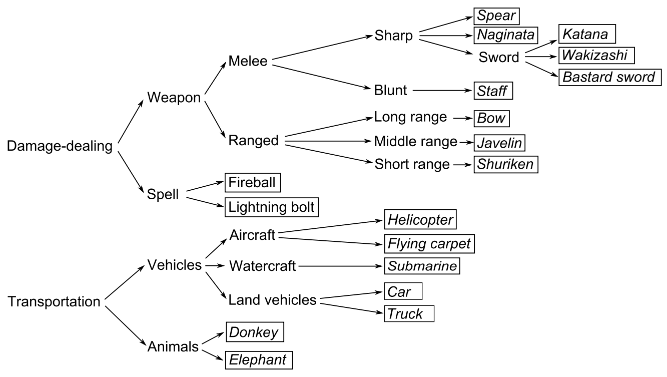


Fig. 2. Example type hierarchies of technologies and products.

Tech Categories:

GENERAL (G)
DAMAGE DEALING (D)
DAMAGE COUNTERING/EVADING (C)
DAMAGE ABSORBING (A)
POWER SOURCE (P)
ENGINES (E)
SENSORS (S)
COMPUTING (M)
HEALING (H)

Non-tech Modifier Sets:

Ammo: {Basic Ammo}
Fuel: {Energy Crystals}
Food: {Worms}
Hull_material: {Basic Alloy}

AC Abbreviations:

C: Cost
S: Size
W: Weight
P: Power Consumption
T: Thrust
N: Range
M: Communication
V: Effect Velocity
U: Accuracy
(K|I)A:
(Kinetic|Incendiary) Attack
(K|I)D:
(Kinetic|Incendiary) Defence

BV Abbreviations:
As corresponding ACs, but prefixed with "B"

Fig. 3. Categories, modifier abbreviations, and NTMs of the example game.

analysis of their different semantic subclasses. Categories built on is-a relationships are often sufficient, but other relations can also be used. In any case, each product and a large portion of technologies should be classified into – possibly several – categories.[6] The categories may form hierarchies. An example hierarchy is shown in Fig. 2: the leaves are technologies and products, and the intermediate nodes are categories. A leaf belongs to all the categories along the path from the root to it. (Separating techs from products here is not a matter of importance, and cannot even be done without extra knowledge of the related game.) Each leaf also forms a category consisting only of the leaf.

When using CATFAT, the current technological state of the gameplay is represented for each player by a set of global properties and properties of technologies and products. It is often convenient to divide properties into two categories: *base values* (BVs) defining initial values and *adjustment coefficients* (ACs) for typical value modifications.

For example, the value for a global AC called "building cost" could be 1.0 at the beginning of the game, and by developing, say, *logistics* the coefficient could be reduced during the game. This reduction would then be reflected – via a simple multiplication with the corresponding base value – to the cost of a specific building operation. For a transport unit, on the other hand, e.g., speed, weight, and cargo capacity could be crucial features to be modeled this way, and fuel type could serve as an example of a non-numeric property that still could be represented with a BV property.

On the other hand, fuel is also a good example of an external non-tech resource affecting features and usability of some technologies. More generally, a technology may be affected in a game by arbitrary non-tech modifiers (NTMs) – variables able to take values in their respective value sets that can also change during a game. For instance, the NTM *fuel* could have a value from the set {gasoline, hydrogen, gunpowder}. Commonly applicable relations between techs and NTMs are "uses" and "has" relations, but other types are also possible.

[6]Strictly speaking at least all the base techs (introduced in Section IV-B, as are also the other technology types mentioned here) should have categories that could be inherited by generated techs. If a modifier tech has categories, they can be inherited too, and any technology that should be affected by property changes for a category should be categorized into it. For simplicity, one can categorize all of the technologies; a generic category can be used, if needed.

B. Technology Types

All the technologies present in a predefined tree (*core tree*) are *core technologies*. That tree will be augmented procedurally with *generated technologies*. Core techs – and their relationships – can be seen as a compact (compressed) presentation of all potential larger tech trees that can be generated – and in the optimal situation, wanted.

Base technologies are selected representants of different fundamental categories among the core technologies. These chosen ones have an important role in the tech generation process: they offer initial tech names and features to be modified in the process.

Modifier technologies are able to modify properties; developing them may affect technologies and products or the gameplay (player-wise or globally), typically by changing AC values. Also BVs can be manipulated. The modifications may also introduce possible technologies to be generated into the tree – *generable techs*. They can manifest themselves as actual generated techs. Generable techs are formed using base technologies or generated techs as the basis and modifying their names and properties according to the rules defined by the modifier tech. Categorizing technologies and their products simplifies forming these rules.

The technology type groups presented here may overlap. Core trees consist of base techs, modifier techs, and possibly other techs not used by CATFAT (all thus being core techs). It is also possible for a single technology to be both, a base tech and a modifier tech simultaneously.

C. Technology Generation

To demonstrate and explain the actual technology generation process, we take an imaginary space strategy game as an example and let building spacecrafts be an essential part of it. Consider a tech tree used to design spaceship parts. The technology categories used are listed in the box of the upper left corner of Fig. 3. In this simple example, there is no need for hierarchies. The box on the right-hand side presents the AC and BV property abbreviations used. All the technologies and products do not necessarily need every one of these properties. The starting NTM sets are listed in the box of the lower left corner of Fig. 3.

The core tree consists of the technologies depicted in Fig. 4 using rectangles and ellipses with solid edges and

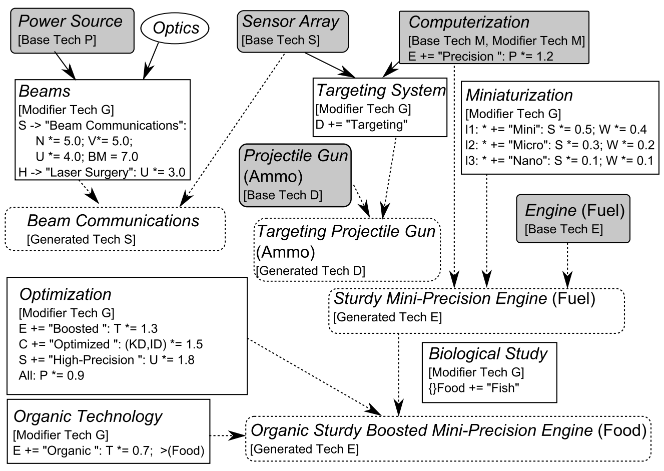


Fig. 4. Example core tree (techs with solid edges and solid arrows) and some generable techs (technologies with dotted edges).

the solid arrows defining the prerequisite relationships. The grey shapes represent base technologies, and the corresponding categories are indicated in brackets. In order to develop a tech with incoming arrows, all the corresponding techs at the start points of the arrows (tails) must first be obtained. Modifier technologies are drawn using rectangles with non-rounded corners. They typically have rules to modify existing tech names and properties category-wise. For instance, *Beams* has a rule for dealing with a technology of the category SENSORS. In the beginning there is only one such a tech available, namely the base tech *Sensor Array*. Based on the rule, the generable technology *Beam Communications* can be produced.

More generally, a base technology or a generable technology (*modified tech*) and a modifier technology with rules compatible with the categories of the modified tech can produce a new generable tech. This is done by creating a copy of the modified tech and then adjusting the copy according to the rules of the modifier technology.[7] In other words, the properties of the produced generable tech are initially set similar to those of the modified tech, and then modified by the rules. These modifications can be, e.g., multiplications, additions, or setting specific property values.

The name of a tech is also determined by the rules generating it. In the case of modifier tech *Beams* and category SENSORS, the name of the base tech is just replaced by a new one. In contrast, all the other modifier techs, except *Biological Study*, concatenate modifying strings with the names of the modified techs in order to form the names for the corresponding generable techs.

There may be several concatenations along the way, and typically the order of modifying strings is important. This must be taken into account when designing the rules. A modifying string of a rule may have, for instance, a numerical value attached indicating its need to occur immediate before – or in the case of postfixes, after – the basic name. The modifier strings can then be ordered based on these values. Other arranging rules can also be applied, if necessary. In Fig. 4, only prefix concatenation is demonstrated without any order data.

By default, the generable techs inherit categories of their parents and can be modified again by different modifier techs. The categories can, however, be controlled by the rules of generation. Either all the parents or only the ones currently not used as modifier techs (as is the case in Fig. 4) may contribute to the categories of their respective children.

Biological Study is here an example of a modifier tech that adds a NTM into a NTM set. Additional sets can also be defined, and existing NTMs and NTM sets can be removed.

Miniaturization demonstrates that a tech can consist of several levels to be developed. This is a convention adopted to make tech tree presentations more compact; instead of defining three different modifier techs in the tree, only one is defined with three levels of development with their corresponding effects. Initially, the actual game should only generate the technology corresponding to the first level. The tech corresponding to the second level ought to be presented only after the first-level tech has been developed, and so forth.

Typically developing technologies in a game costs resources like time, energy, or gold. Defining the prices to be set for generated technologies is basically a challenging task, especially, if there are several resource types in use. If there is only one type to consider, however, the difficulty can be overcome by combining the use of an *evaluator function* and a *balance corrector function*. Evaluator functions have to be implemented case-by-case. The idea is to make such functions evaluate the values of technologies under scrutiny based on their properties (effects in the game) and use these estimates in order to determine the prices for developing the techs. Also, the value estimates of, e.g., the children of the techs can be used in the process.

The purpose of the balance corrector function is to check through the current tree and make necessary modifications to the costs – and possibly also to the effects – in order to prevent generating too powerful paths and keeping the tree in balance in the sense that basically advanced techs should be more expensive than less advanced ones, and different techs of, e.g., the same historical era should have total development costs (including all the prerequisites) of the same magnitude. In the case of several resources to consider, the same principles apply, but additional metadata is needed to keep the assigned costs semantically meaningful.

The basic approach can be improved by using modifier entities that act similarly to modifier technologies, but are not included in the tree as technologies and cannot be developed as conventional techs. Predefined game events or random events could lead to applying modifying rules of these entities. For instance, a player could complete a quest on behalf of an industrial company in a game. As a reward, limited products of that company – specialized items with company-specific characteristics – might be made available to the player as technologies. Negotiating deals with, e.g., nations or species could lead to somewhat similar results. There could also be reverse effects limiting the availability or removing certain techs, launched by other events. Let us call game events with modifier tech–like capabilities *modifier events*.

[7]For simplicity, here we restrict ourselves to the cases, in which each generable tech is based on one modified tech, but more complex schemes can also be allowed.

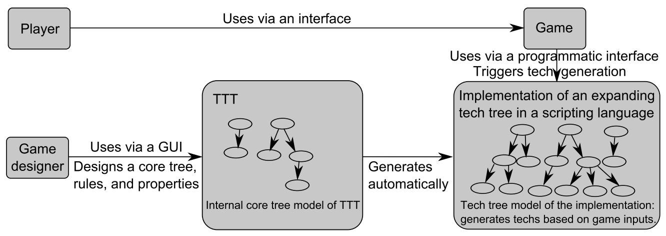
The general procedure of applying CATFAT in a game with dynamic tech generation based on the actual technological situation is as follows:

```
initially:
  $C \leftarrow \emptyset$

whenever tech $t$ is developed:
  if $t$ is a modifier tech:
    select the rules of the lowest level of the tech as \
    the active ones and the rest as inactive ones.
    if $t$ has more than one level:
      generate a next level tech, containing the rest of \
      the levels, as a child of $t$
    for each active rule $r$ of $t$:
      if $r$ is a general rule not bound to categories:
        apply $r$
      else:
        for each nonmodifier tech $m \in r.category$:
          generate a tech candidate, $c$, based on $m$ \
          using rule $r$
          $C \leftarrow C \cup c$
  else:
    for each developed modifier tech $m$:
      for each active rule $r$ of $m$:
        if $t \in r.category$:
          generate a tech candidate, $c$, based on $t$ using \
          rule $r$
          $C \leftarrow C \cup c$
    for each modifier event $e$ with active generation rules:
      for each active generation rule $r$ of $e$:
        if $t \in r.category$:
          generate a tech candidate, $c$, based on $t$ using \
          rule $r$
          $C \leftarrow C \cup c$
  Select technologies from $C$ and add each of them as \
  a child of its corresponding modified tech. (All the \
  candidates can be added, or only some of them, \
  based on, e.g., the currently selected focus area of \
  the research work in the game.)
  Remove the selected techs from $C$.
  run evaluator function
  run balance corrector function

whenever modifier event $e$ takes place:
  for each active rule $r$ of $e$:
    if $r$ is a general rule not bound to categories:
      apply $r$
    else:
      for each nonmodifier tech $m \in r.category$:
        generate a tech candidate, $c$, based on $m$ \
        using rule $r$
        $C \leftarrow C \cup c$
  Select technologies from $C$ and add each of them as \
  a child of its corresponding modified tech.
  run evaluator function
  run balance corrector function
```

This basic procedure can be modified as seen fit, but the basic idea is to add technologies to the pool of possible techs as the player progresses in a tree and then augment the tree suitably from that pool. If there are effects reducing possibilities to develop technologies, this must be taken into account, and the pool must be updated correspondingly.



Player
Uses via an interface
Game
Uses via a programmatic interface
Triggers tech generation
TTT
Game designer
Uses via a GUI
Designs a core tree, rules, and properties
Internal core tree model of TTT
Generates automatically
Implementation of an expanding tech tree in a scripting language
Tech tree model of the implementation: generates techs based on game inputs.

Fig. 5. Relations of different actors and components.

As a part of this method, it is futile to try to define exactly, how the selection of technologies to be added among the candidates should be done, because technology trees and games differ a lot from each other. Therefore, the selection method should be designed or chosen case-by-case.

The selection can be implemented either as an internal tech tree functionality or by the main program using the tree. Our suggestion is to let the trees take care of the whole generation process including the selection, because this way trees can be swapped with each other easily. Unnecessary dependencies between tech trees and their users should be avoided.

V. Tool Support

As a merely abstract approach CATFAT is useless. With suitable software tool support, however, it can be turned to a practical method. As prior work, we have implemented TTT to facilitate the design process, evaluate tech trees, manipulate their properties, and generate code automatically. In order to experiment with CATFAT, TTT was extended to support rule-based technology generation as described in this paper.

So far, TTT only supports disjoint categories, but it is trivial to add support for hierarchies by, e.g., defining and storing hierarchy graphs similarly to tech trees and then defining, which hierarchies a technology tree should use. Categories of technologies can be easily changed via a graphical user interface and default properties for technologies can be set. Naturally, properties of a particular tech can also be easily modified. Generation rules must, for now, be expressed using a syntax resembling the one used in Fig. 4. In the future, graphical tools and mouse gesture shortcuts can be added for even easier rule definition.

TTT can generate a functional technology tree implementation in Lua language based on the design defined via its GUI. Currently, such automatically generated tech tree implementations augment themselves by all the possible additional technologies every time a tech is developed. So far, there is no support for modifier events, but that is a straightforward feature to add, if desired.

Fig. 5 clarifies the role of TTT in the process; a game designer uses it to define the core tree and to create the necessary rules for modifier techs. TTT then generates a functional tech tree implementation in Lua. It is a rather autonomous entity having its internal data models, bookkeeping, and functions for performing necessary operations. It also offers a simple interface, via which the actual game can use the tree and ask the tree to perform different operations concerning itself. Based on input received via this interface and the internal state of the tree, more techs are generated as necessary and the data model is updated.

Moreover, TTT serves as a test environment for tech tree designers; the designer can select technologies to be virtually developed in the design view, and the tool generates techs and adds them to the visual tech tree presentation. The properties of the generated techs can be observed and modifications to the rules can be made accordingly. This speeds up the initial testing, because the tech tree can be "run" outside of the context and constraints of the actual game. Within the tool, there are also different tech tree analysis and measurement functionalities available for evaluating the tree properties.

VI. Experiments

To test the usability of our approach, we created a small core tree – corresponding to the one illustrated in Fig. 4 – for an imaginary game focusing on improving spacecrafts by upgrading, modifying and inventing different components for them. After modeling the core tree using TTT, we simulated proceeding in the tree and let the system generate additional content accordingly.

In Fig. 6, a partial screenshot of the tech manipulating area of TTT user interface is shown. A user can select technologies to be developed using a pointing device (a mouse), and the tool generates more technologies to the tree accordingly. The properties of selected technologies can be seen and manipulated in a separate property window.

We also let TTT to create a functional tech tree implementation in Lua based on the designed core tree, and run several sequences of technology selections via the tech tree–using interface (from a command line) corresponding to the sequences simulated inside TTT. In order to compare these results with the ones obtained by simulations within TTT, the processes were kept deterministic. We could verify that the Lua script generated by TTT was able to create and alter technologies and properties similarly to TTT in its internal simulations.[8] A main game could use the technology properties and general tech tree functionality via the Lua interface, and the tech tree could be expanded at the runtime by the Lua scripts generated by TTT without any effort by the main program.

These are, of course, only initial experiments, and more testing is needed. Because of this fact and the limited space in the paper, explaining the experimental setup in more detail is omitted as negligible; the proper evaluation of the approach deserves its own paper anyway. However, based on the initial tests, it strongly seems that the method is viable and the tool support facilitates creating expandable tech trees considerably. Gathering experiences of using CATFAT and TTT also in the context of real game development is important, but at this point we have to leave this as future work as well.

[8]At the moment, the internal technology development simulation functionality of TTT is implemented in C++, so it made sense to compare the results with the ones given by the Lua implementation generated by TTT. In future, it might be reasonable to make TTT automatically generate a Lua implementation of the core tree under scrutiny for performing any in-tool simulations instead of using its own internal implementation of the algorithm.

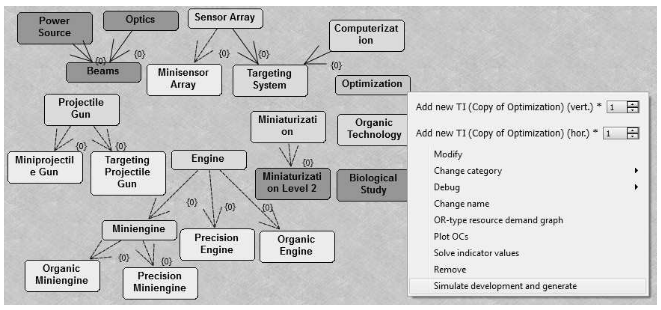


Fig. 6. A partial screen capture of the TTT GUI. Development of *Targeting System* was just simulated, and TTT generated *Targeting Projectile Gun*.

VII. Discussion

The set of technologies that can be generated with the approach presented in this paper is still limited, since additional modifier techs are not generated in the process, and using a single modifier tech several times "in a row" is generally a bad idea. It is also hard to generally guarantee the high quality of each technology that is generated. The need to use natural language restricts possibilities to generate technologies automatically, but with languages like English the approach is still feasible. The quality of the generated content depends on the modeling choices made and the definitions given. Thus, rigorous planning improves the quality.

Because the method is only capable to create content derivable from the given technologies and rules, it may look like only a way to represent data in a compressed form, and in a sense, this point of view is totally valid. However, we argue that we are dealing with automated content generation: all the generation methods need some initialization and use some rules to construct "new" content. A person defining the core tree and the rules may not be able to notice all the implications, and a computer can, based on the provided data, form unforeseen technologies.

Basically, the approach is all about changing the focus from first developing lots of detailed content and then pruning, modifying and unifying it, to designing a compact rule set capable to define the possible achievements. Despite its limitations, our approach has its benefits. In contrast to the more common method to create tech tree variations by removing techs from a large pre-existing technology tree structure, growing a tech tree based on few properties, techs, and rules makes it possible to easily adjust general mechanics and idea manifestations globally. The process is also meant to produce a unified logic throughout the tree, which can be a cumbersome task to carry out with traditional means. Although the core tree has to be built with consideration, applying CATFAT may still also result in savings of time and work.

There are, of course, tech tree types, for which it is hard to generate additional technologies by simply concatenating additional strings to tech names and letting techs inherit most of their properties. In such cases, lots of effort must be put into defining non-generalizable rules for generating single techs. This means returning to the traditional way to define tech trees.

Nevertheless, if the technologies are generated as the game progresses, one cannot – at least in theory and without prior experience or with randomization used in generation – see the possibilities offered in the distant future. Traditionally, tech trees allow one to "build toward a goal" [17]. However, even if the whole tree happens to be predetermined, we consider revealing it all at once in the beginning of the game generally a rather poor idea. Although one can optimize better with such prior knowledge, often this kind of foreseeing makes a game feel unrealistic and highlights the rigidity of the tree.

One could critisize our approach, because it does not necessarily lead to any visual or functional novelties concerning user experience; this work does not try to change technology trees as structures into anything unprecedented. A casual gamer could try a game without even noticing if it was using sophisticated mechanisms to create technologies or not. This argument, however, applies to PCG quite generally and is not specific to our approach; PCG is typically used to enhance conventional games, not to create totally new types of them. Even the casual gamer – not especially interested in tech trees – might, however, notice the difference, if the game was played again, and the tree was formed differently because of some randomization or learning scheme used. If the user actions were taken into account in the dynamic tech tree creation, the overall user experience might be improved from the baseline case of a rigid tech tree, even if the user could not appreciate the mechanisms in work.

In this paper we have presented examples of simple multiplicative property modification only, and so far that is the only modification type supported by our implementation besides setting values explicitly. Such modifications seem to be viable and useful, but naturally, other ways to control properties may be added in order to further improve the approach.

In addition to generating techs during runtime, CATFAT can be used to facilitate traditional tech tree design process: only a small amount of work is required to generate a tech tree suggestion within the desired thematic and modeling constraints. The tech tree for the final product can be prepared based on such generated suggestions. For this kind of use it might be beneficial to produce relatively large and modifiable "generic" core trees for different game genres and settings. With standardized tech tree implementation formats, offering such products might even be convertible into a business model.

VIII. Conclusions

We have presented CATFAT, a general approach for augmenting technology trees by creating additional technologies on runtime. The method reverses the conventional process of creating temporal variance by first defining a large tech tree and then leaving technologies out. CATFAT rather starts with a minimal skeleton tree and a set of rules, and generates techs as they are needed. Designing rules instead of single technologies facilitates keeping the tech tree consistent. The approach can also be used in a more conventional tech tree building process to generate suggestions to start the design work with.

Initial tests indicate that the method can be used in practice, and the resulting tech trees are of adequate quality. However, mostly the method is still on the idea level, and further evaluation and testing is still needed to ensure its general applicability and usefulness. The key challenges are different natures and characteristics of technology trees used within the wide array of digital games, the difficulties of natural language processing, and the visualization of tech-related entities.

As future work, we intend to continue testing and improving the CATFAT method and the TTT software. Generating complex content related to techs – such as unit types with their graphical representations – is a challenge that remains still to be tackled. Interesting results might be obtained, e.g., by using genetic methods. Adding a user behavior–observing manager to control the generation in order to customize the gameplay automatically for individual players might be worth trying. Such functionality could also be incorporated, e.g., into the evaluator and balance corrector functions used.

References

[1] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. MIT Press Ltd, 2003.

[2] A. Solo, "Innovative tech trees in space strategy games," http://www.spacesector.com/blog/2009/07/dynamic-and-specialized-technology-research-in-space-strategy-games/, 2009, accessed March 30, 2015.

[3] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 9, no. 1, pp. 1:1–1:22, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2422956.2422957

[4] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[5] E. Watrall, "Chopping down the tech tree: Perspectives of technological linearity in god games, part one," http://www.gamasutra.com/view/feature/131570/chopping_down_the_tech_tree_.php, 2000, accessed July 19, 2012.

[6] T. Ghys, "Technology trees: Freedom and determinism in historical strategy games," *Game Studies*, vol. 12, no. 1, Sep. 2012.

[7] T. J. Heinimäki and T. Elomaa, "Facilitating technology forestry: Software tool support for creating functional technology trees," in *Proceedings of the Third International Conference on Innovative Computing Technology, INTECH 2013*, Aug. 2013, pp. 510–519.

[8] T. J. Heinimäki and T. Elomaa, "Quality measures for improving technology trees," *International Journal of Computer Games Technology*, vol. 2015, pp. Article ID 975 371, 10 pages, 2015.

[9] D. Thue and V. Bulitko, "Procedural game adaptation: Framing experience management as changing an MDP," in *Intelligent Narrative Technologies: Papers from the 2012 AIIDE Workshop (AAAI Technical Report WS-12-14)*. AAAI Press, Oct. 2012, pp. 44–50.

[10] G. N. Yannakakis and J. Hallam, "Real-time game adaptation for optimizing player satisfaction," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 2, pp. 121–133, Jun. 2009.

[11] E. Adams, *Fundamentals of Game Design*, 2nd ed. New Riders Publishing, 2009.

[12] A. Zook and M. O. Riedl, "Automatic game design via mechanic generation," in *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014.

[13] J. Alvarez and D. Djaouti, "An introduction to serious game – Definitions and concepts," in *Proceedings of the Serious Games & Simulation Workshop*, 2011, pp. 10–15.

[14] P. Rego, P. M. Moreira, and L. P. Reis, "Serious games for rehabilitation: A survey and a classification towards a taxonomy," in *the Proceedings of the 2010 5th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, June 2010, pp. 47:1–47:6.

[15] K. Squire and H. Jenkins, "Harnessing the power of games in education," *InSight*, vol. 3, no. 1, pp. 5–33, 2003.

[16] N. Hallford and J. Hallford, *Swords & Circuitry: A Designer's Guide to Computer Role-Playing Games*, ser. PRIMA TECH's Game Development Series. Prima Publishing, 2001.

[17] P. Tozour, "Introduction to Bayesian networks and reasoning under uncertainty," *AI Game Programming Wisdom*, vol. 1, pp. 345–357, 2002.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3633-5
ISSN 1459-2045