DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

nyan - Hierarchical Key-Value Database with Inheritance and Runtime Patching

Jonas Jelten

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

nyan - Hierarchical Key-Value Database with Inheritance and Runtime Patching

nyan - Hierarchische Schlüssel-Werte Datenbank mit Vererbung und Laufzeitveränderungen

Author:	Jonas Jelten
Supervisor:	Prof. Dr. Gudrun Klinker
Advisor:	M.Sc. Sandro Weber
Date:	2017-10-15

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 2017-10-15 Jonas Jelten

Abstract

Complex applications such as real-time strategy games often source their content and configuration from a dedicated description interface. This separated interface requires a powerful data description framework to express the desired structures. This thesis introduces `nyan`, a data description framework which is designed as an external content and configuration storage. `nyan` incorporates several features which are unavailable in current data description systems. These include type-safe representation of hierarchically related data and transitive modifications at runtime. The framework provides two main interfaces. A user-friendly language in which configuration data is declared and an API over which the data can be accessed and modified. The data description language uses the concept of inheritance to enable specification of hierarchical structures. The API provides means of dynamic data modification and subsequent rollbacks. This is realized by tracking all occuring runtime changes and their time of commit. Dynamic changes are specified as patches, also written in the `nyan` language. In contrast to existing configuration systems, `nyan` provides all of these features directly through its semantics without sacrificing its generalizability in contrast to an application-specific language. The `nyan` database library is implemented in C++14 and is published as a free software project.

Zusammenfassung

Komplexe Anwendungen wie Echtzeit-Strategiespiele beziehen ihren Inhalt und ihre Konfiguration oft über eine dedizierte Beschreibungsschnittstelle. Diese getrennte Schnittstelle erfordert ein mächtiges Datenbeschreibungssystem, um die gewünschten Strukturen auszudrücken. Diese Arbeit stellt `nyan` vor, ein Datenbeschreibungssystem, das als externe Daten- und Konfigurationsspeicherung konzipiert ist. `nyan` enthält einige Funktionen, die in aktuellen Datenbeschreibungs-Systemen nicht verfügbar sind. Dazu gehören die typsichere Darstellung hierarchisch verknüpfter Daten und die transitive Anpassung zur Laufzeit. Das System bietet zwei wesentliche Schnittstellen. Eine benutzerfreundliche Sprache, in der Konfigurationsdaten deklariert werden und eine API, über die auf die Daten zugegriffen und diese geändert werden können. Die Datenbeschreibungssprache nutzt das Prinzip der Vererbung, um die Angabe hierarchischer Strukturen zu ermöglichen. Die API bietet die Möglichkeit der dynamischen Datenmodifikation und des anschließenden Zurücksetzens. Dies wird durch die Aufzeichnung aller auftretenden Laufzeitänderungen und deren Übernahmezeit realisiert. Dynamische Änderungen werden als Patches angegeben, die ebenfalls in der `nyan`-Sprache geschrieben werden. Im Gegensatz zu bestehenden Konfigurationssystemen stellt `nyan` all diese Funktionen direkt durch seine Semantik zur Verfügung, ohne seine Verallgemeinerungsfähigkeit im Gegensatz zu einer anwendungsspezifischen Sprache zu verlieren. Die `nyan` Datenbank-Bibliothek ist in C++14 implementiert und wird als freies Software-Projekt veröffentlicht.

Contents

1 Introduction

1.1 Motivation

Reusing parts of a software for different applications saves time and thought. Frameworks are created to provide reusable designs for entire applications [23].

For game development, the game engine is the framework to build the game with: When designed well, there is a clear distinction between engine and game content [5][7]. The building blocks of the game engine can then be combined into the desired game behavior and look. The interface of the game engine, which allows the configuration and recombination of the engine's features (like game logic, user interface) is its application programming interface (API) [21].

For example in real-time strategy games, such as Age of Empires or StarCraft, this interface must be powerful enough to reflect all aspects of the game mechanics. In these games, the goal is to collect resources to build up an army and win against the (human or computer controlled) opponents. Each player chooses a civilization or nation to play as. During the game, each player can control many units (200 or more), and each type of unit has special traits: It has a distinct look, abilities (behavior) and can be improved through research (upgrades). The units are structured in hierarchies, for example in Age of Empires, a "crossbowman" is an "archer", which is a "ranged unit", which is a "unit". When an upgrade for all ranged units is researched, it will affect all crossbowmen, and archers. An upgrade for a crossbowman does not affect archers or ranged units. There is also the need for team bonuses (a player's civilization provides improvements for all players that are a member of one team) [36].

A game engine has to provide configuration data structures and computational support to perform the game simulation that follows the configuration information. Modern games support the creation of mods, which are plug-ins to enhance and overhaul the game easily. They can, for example, add new civilizations, perform tuning to unit properties so the game becomes more balanced, add new stories and scenarios etc. Support for game mods can bring great improvements and new content for the project [49].

`nyan` is a framework to be used by an application like a game engine. `nyan` provides features to create advanced configuration systems, for example to extend the game logic and modify assets. The need for such a system was discovered during development of the `openage` game engine [36]. `openage` is a open-source realtime strategy game engine, which is designed to run Age of Empires 2 and similar real-time strategy games [36]. One goal of the project is to not only run the game alone, but also to allow modifications and extensions through mods. Independently developed mod should be able to be combined without conflicts. `nyan` was designed for exactly that purpose: To create a general-purpose run-time configuration system for the configuration and extension of the `openage` game engine. The system is designed to be used in other games and applications as well, therefore usage is not restricted to `openage`. In layman's terms: `nyan` is not made for building a machine, but rather to provide a nice way to control its levers and valves.

1.2 Goals

The overall goal of this thesis was to develop a data specification framework which can handle hierarchical and connected data structures and their transitive modifications. This framework is designed to be embedded in an application. The framework consists of a data description language, a storage database and an API to be used by the application. The API is used by the application for queries and data modifications.

The data description language is the `nyan` language, which is used to store configuration records in "`.nyan`" files. The language must be expressive enough to allow to provide the features mentioned in section 1.1. The language to be created has to be declarative [27] and type safe. A proper parser for this language has to be created. The integrity of loaded data is checked when new files are loaded to detect errors as early as possible [28].

For sanity and type checks, `nyan` must support the configuration of the allowed data structure, which is the data *schema*. An application can then rely on it for queries.

In order to extend the application, it must be possible to update the data schema while the application is running. To allow independent modifications to the schema, present data must not become invalid, but data for the new schema must be allowed. The `nyan` database must therefore provide an API enabling for queries to records and schema configurations.

When this is implemented properly, a game engine like `openage` can be changed through mods and new content can be added. Those mods perform their changes through the `nyan` language, where they describe their entry points, new artwork, new unit behavior and properties, and so on. `nyan` is not a general-purpose programming language, it should only make it possible to describe data structures for values and to register custom functions in the application so they are called when appropriate.

The `openage` engine is attempting an event-based simulation approach, where game state predictions are made, and when preconditions change, those may be dropped [60][36]. Predictions are used to schedule updates in the `nyan` database to be activated at a future point in time, for example the prediction of a change to a speed property that all horse units will run faster in 10 seconds. Those updates must be rolled back if the prediction turns out to be wrong [60].

nyan must be able to support this kind of history tracking and rollbacks.

In the end, nyan shall be a framework to support all those requirements and be completely rounded, following the usability principle ”Fancy is easy. Simple is hard.” [56] and guided by the Zen of Python [40].

1.3 Outline

The current introduction chapter summarizes the thesis’ subject and purpose. Background information regarding involved concepts for understanding following sections is provided in Chapter 2. Related approaches for the problem nyan is designed to solve are presented in Chapter 3. In Chapter 4, the design of the nyan language, database and API are elaborated, including all necessary components. Chapter 5 presents a possible implementation that works in practice, followed by Chapter 6 which examines the implementation’s effectiveness and discusses properties and potential problems of the design ideas. Finally, Chapter 7 proposes possible extensions for the future and concludes this thesis.

2 Foundation

This chapter will provide information for understanding later sections of this thesis. It introduces fundamental concepts, on which the work described in this thesis is based on.

2.1 Game Engine

A *game engine* is a framework for creating interactive realtime graphics applications. It implements algorithms for calculating the virtual world simulation, which is the game that is run [5].

The game engine is the foundation of a game, if it does not support a certain feature or provide some means to add it, the game cannot use it. The interface which the game makes use of to access and configure those features is called API. Hence, the crucial component of a game engine is its API [21].

If one has access to the source code of the game engine, any feature can be added by the developers. Once the game was distributed to a user base, changes can only be delivered if users perform software updates, which are authored by the engine developers. This includes changes to the engine API, but software using this API can be distributed independently. That means if the API enables it, other developers can change and enhance the game without ever touching the game engine itself. Because of that, a game engine should provide a good API so content creators don't have to adapt the game engine [7].

Today's notable high-class commercial engines are Creation, GameBryo, Unity, Unreal, Source, CryEngine [26].

2.2 Modding APIs

The word "modding" is a derived gerund from the verb "modify" and means the customization of something. Mainly used in the technology world, it means the attempt to get more out of a given product. The community which collaborates and organizes itself to create so-called "mods" is growing bigger and bigger [49].

Such modifications often are not intended by the original manufacturer, but can bring some significant improvements over the "vanilla" (i.e. unchanged) product. For example, Apple had removed the standard headphone jack from the iPhone 7, but recently a "modder" added it back in [31].

Though this example is from the hardware world, in many cases it is easier to change and extend a product in the software world, especially in case of games [49]. The game industry has discovered this and actively supports the creation of new content by the community, for example in the Steam Community Workshop for games like Skyrim [32].

As most games are proprietary, mods can only be created by utilizing the provided API, because changes to the game engine are not easily possible [17]. If there is no API or the API functionality is limited, modders have no chance, at least at first sight.

In some games though, the desire to make changes to games is so immense that extended means are used: The community then creates a modding API themselves by reverse engineering and extending the game engine. This was done for example for Minecraft [6] (*Minecraft Forge* [29] and *Sponge* [30]) or Dwarf Fortress (*DFHack* [16]).

With this approach, even though the game publishers did not intend it, any game can be given a modding API, even if this means considerably more effort than having a proper interface in the first place [21][32].

2.3 Database

A database is software for storing organized collections of information. Today, they are crucial for information processing to perform fast storage and retrieval of data. There are several approaches to the concept of a database, mainly differing in the possible data structures and organizations (e.g. key-value, graph, objects, tables) [33][24].

Database systems strive to comply with the so-called ACID properties, which describe that a database shall follow: Do only *atomic* transactions (as described below), stay *consistent* so that an invalid database state is never reached, *isolate* multiple clients so their transactions don't interfere and guarantee *durability* so that the database state is not affected by crashes or power loss [19].

To achieve atomic changes to stored values, databases perform transactions. A transaction stores one or many operations to perform on the records, and the database system guarantees that either all those changes or, in case of any problem, none are committed, leaving the records unchanged. To achieve this, two approaches are popular: "write ahead logging" creates a list of changes to be done, and when no more changes are to be done, the database is locked and the list is traversed. In case of conflicts, the list is walked backwards and the changes are undone, until the original state is reached again. The other approach is "shadow paging", where the records to be changed are copied, and changes are then performed on this copy. When problems are detected, this copy is deleted and the previous state is restored. Otherwise, the copy is installed as the new state [19].

2.4 Inheritance

Inheritance is a well-known concept performed deliberately (e.g. by inheriting a family item) or in biology/genetics (e.g. by inheriting eye color) [39].

The same concept, that a parent passes down something to its children, was initially invented for the SIMULA programming language [14]. In object oriented programming, *classes* are used to encapsulate several values to a logical group. A new class can then list parents to gain their features as well. The *subclass* possesses all properties of its "base class", but can override or customize any behavior [54]. This allows an incremental approach of building blocks, where each refinement can be done as another subclass [8]. It is possible to have an *abstract base class*, which declares that some property or method will be there, but provides no actual value. That way, it forces that some subclass implements the missing values before it can be used. With abstract classes, a software can assume that a subclass is structured in a predictable way, and can do computations with this assumption, without knowing the effective implementation first [23].

When it is allowed to inherit from multiple parents, the "diamond problem" arises [11]. The problem stems from the ambiguity of which parent's method to choose in a situation like illustrated in figure 2.1. When `B` and `C` both override the same property of `A`, which one shall be chosen in `D`: The variant of `B` or the one of `C`? Is `A` contained once or twice in `D` [8]?

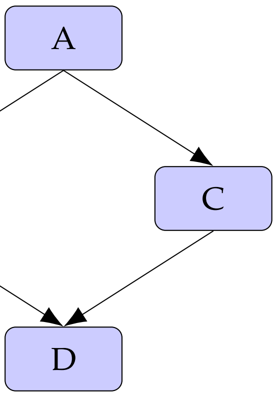

Figure 2.1: The inheritance diamond

A possible solution to this ambiguity is the `C3` linearization algorithm, which turns the inheritance graph of some class into a predictable list. The linearization for a class is calculated recursively by traversing the inheritance graph [2]. The `C3` linearization is used in Python for its method resolution order [43].

The `C3` linearization calculation works as follows [2]:

```
# calculation of linearization of class cls(a, b, ...)
c3(cls) = [cls] + merge(c3(a), c3(b), ..., [a, b, ...])
```

The result of the linearization is combined by the class name itself and the `merge` function. The merge function takes lists as its arguments, and the lists are created from recursive linearization calls to C3. The recursion is terminated when a class with no parents is linearized.

The merge function takes the first head element of all its arguments (which are lists) which is not present in any tail of all those lists. The head element is the first entry of a list, the tail are all remaining elements. If a head element missing from all tails is found, it is put into the results list, and it is removed as head element from all other lists. This is repeated until all lists of the merge function are empty. If none of the heads of those lists is a candidate (as each appears somewhere in a tail), no linearization is possible and the algorithm aborts.

In example 2.1, we had 4 classes, defined as A, B(A), C(A) and D(B,C). The linearization of D with C3 is calculated as shown in Listing 1.

```
# recurse into parents
c3(D)     = [D] + merge(c3(B), c3(C), [B, C])
  c3(B)   = [B] + merge(c3(A), [A])
    c3(A) = [A] + []
  c3(B)   = [B] + merge([A], [A])
  c3(B)   = [B, A]
c3(D)     = [D] + merge([B, A], c3(C), [B, C])
  c3(C)   = [C] + merge(c3(A), [A])
    c3(A) = [A] + []
  c3(C)   = [C] + merge([A], [A])
  c3(C)   = [C, A]
c3(D)     = [D] + merge([B, A], [C, A], [B, C])
# now, the real calculation
c3(D)     = [D] + merge([B, A], [C, A], [B, C])
c3(D)     = [D, B] + merge([A], [C, A], [C])
c3(D)     = [D, B, C] + merge([A], [A], [])
c3(D)     = [D, B, C, A]
```

Listing 1: Example of C3 linearization

Each indentation level means one level of recursion to calculate the nested invocations of c3(B) and c3(C).

The result list [D, C, B, A] means that when looking for a property, these classes will be queried for this in that list's order. It is also possible to use this list in reverse for aggregations: Starting at A, data flows down to B, then to C and is collected at D [2].

2.5 Language Oriented Programming

"Language oriented programming" means designing a language around the problem one is trying to solve, rather than adapting existing general purpose tools for the problem. A domain specific language is created to fit its purpose.

The advantage is that such a language may be very elegant for the specific goal it is designed to solve, but it is unsuitable for applications out of its scope [61].

The nyan language is domain specific, following this approach. The problem is to provide an extensible, type-safe mod API, which can handle inherited data structures and can track and revert changes over time.

3 Related Work

There have been many attempts to the problem of run-time configuration. If the configuration of an application was not changeable at runtime, all parameters had to be hardcoded, thus requiring recompilation of the code. Even programming languages like Python can be seen as a form of runtime configuration, as their virtual machine (VM) (the interpreter) stays the same and the Python programming language is basically a behavior configuration for the VM [10].

A common approach for run-time configuration storage is the usage of a general purpose markup language, such as `YAML`. If important features are missing, more domain specific languages such as `QML` for graphical user interface (GUI) creation were created.

3.1 YAML and JSON

`JSON` aims to be a fat-free alternative to XML and is a subset of the ECMAScript programming language [13][9]. It provides basic syntax to store nested key-value pairs, lists with numbers, strings or boolean values. It does not allow any value typing, although types can be expressed as string literals and type-checked by an external framework like `JSON` schema [18]. `JSON` does not provide a built-in document structure and does not allow comments [9]. An example of `JSON` is presented in Listing 2.

`YAML` is another popular data storage format, which is a superset of `JSON`, focuses on feature-completeness to express native data structures and still remain human readable. It builds on the lessons learned from XML and tries to be simpler with more built-in types [4]. `YAML` allows typing for data fields, and

```
{"Image": {
    "Width":  800, "Height": 600,
    "Title":  "Bengal Cat",
    "Thumbnail": {
        "Url":    "http://magic.cats/image/cute.jpg",
        "Height": 125,
        "Width": 100
    },
    "Animated" : false,
    "IDs": [116, 943, 234, 38793]
}}
```

Listing 2: Example of `JSON` code

data fields can reference another, therefore YAML documents are data graphs. YAML does not provide a built-in data schema validator, external tools have to be used. Example code is provided in Listing 3.

```
# sequencer protocol for material work
---
- step:  &step0  # defines anchor label &step0
    device:         CNC 9000
    tool:           drill 5.4
    path:           [12, 234, 23, 10]
    speed:          530
- step:  &step1
    instrument:     CNC 9000
    tool:           grinder 42
    path:           [0, 400]
    speed:          90
- step:          # reuse &step0, override the path
    <<: *step0
    path:           [50, 321, 0, -31]
- step:  *step1  # completely reuses &step1
```

Listing 3: Example of YAML code

YAML can also represent hierarchical data structures, but aggregation logic and patching is missing in its design. It can share and override keys with its "merge keys" feature, but there is no way to extend or modify the values by operators. Hence, to add operators in YAML, one would have to declare a convention to embed operators as strings into keys: `property_increment : value` could be written instead of `property += value`. The merge-keys feature can't be used for this though as the key-name is different then. This means that a single member must be represented by multiple key-values, where one entry is for the name, one for the operator and one for the value. The aggregation logic to evaluate the operators then has to be implemented manually because YAML is not designed for such a use case.

YAML and JSON both are pure data description languages where all the data handling is left to the application reading this data. Schema validations are only possible with external tools. Their ability to express a data interface which can then be used and overlayed by independent data packs is not built-in and has to be interpreted by the application. Modification of values is not built-in as there is no math operations or history tracking.

3.2 QML

QML is a declarative user interface (UI) markup language for QtQuick [46], which is a software development framework from Qt [47].

It is designed to allow the easy creation of interactive applications and describe its components interactions and relations. The default UI building blocks are provided by the QtQuick modules, an example is in Listing 4.

```
import QtQuick 2.9

Rectangle {
    id: canvas
    width: 250
    height: 200
    color: "blue"
    Image {
        id: logo
        source: "pics/logo.png"
        anchors.centerIn: parent
        x: canvas.height / 5
    }
}
```

Listing 4: Example of QML code

The QML language has a syntax similar to that of CSS and the structure of JSON, and it embeds JavaScript for calculations and additional customizations. Nested declarations are influenced by their parent implicitly (in the example, the image is placed inside the rectancle), which can be further customized (e.g. the x positioning relative to the canvas height).

This means that properties can be bound to the value of another object via its special id property, thus the data model is a graph. QML also provides built-in support for transitions between property changes (e.g. smooth animations and the like), the "keyframes" of the animation are set up with explicit "states".

As QML is a part of the Qt-framework, it cannot be used without it. Although new QML-types and their available members can be added through its C++-API, new types can't be added at runtime, so mods in a scripting language would not be able to add new types for new API functionality.

QML is primarily designed for UI building, not as a general purpose data interface. Combination and linking of all the elements in a QML document can be done easily, but the format is not designed for the extension and modification of a document at runtime.

A QML document has no schema verification although all the object properties have types. The types can be declared and activated from the Qt-C++-API, or new combined types are created within QML documents itself [15].

3.3 Mod APIs

Many successful games provide a mod API or the community has created an inofficial one. In this section, some of the typical approaches to such interfaces are presented.

There are multiple approaches to such an API: A code-only interface, a data-only configuration or a hybrid approach. Code interfaces operate by registering functions as hooks into specific triggers, provided by the application. Data-only configuration is very common for all kinds of applications, in most cases this is done through key-value based configuration files (e.g. `.ini` or `.yaml`). A hybrid approach is the combination of both, where configuration files are used to make changes to the application behavior and also to change or register function hooks. The overall feature set depends on the interface: If there is no suitable hook for a function or no configuration variable, then a mod cannot perform the desired change.

The main reasons for not allowing to add new code to an application is the increased complexity of the extension API and code execution rights. If only data variables can be changed, the codebase is self-contained and no third party code can be executed on purpose. If third-party code is desirable, either reviews of this code or a proper sandbox for untrusted code is required.

3.3.1 Minecraft Forge

Minecraft is a cube-based open-world sandbox game [41]. The current state-of-the-art Minecraft API is Forge [29], which is used and configured through Java, thus a pure code API. New code is registered with the API to Minecraft by Forge, through its `RegistryEvents`. New items, blocks and biomes can be created and then registered easily.

Forge can inject values into prepared members of Java classes, so mods can change fields of other mods, if those are designed for it.

Forge is not created by the Minecraft developers themselves, but instead is a community effort. They have spent much time of reverse-engineering Minecraft to inject their hooks at the appropriate places, so that mod developers then can rely on Forge, which adds their code at the right place in Minecraft itself [29].

3.3.2 OpenTTD NewGRF

OpenTTD is a open-source reimplementation of Transport Tycoon, which is a transport business simulation game [1]. The OpenTTD engine can be extended with data from their “New Graphics Resource File (`NewGRF`)” files [37]. A `NewGRF` file can be created from the `NewGRF Meta Language` (`NML`). It is a custom language with predefined blocks, which are provided by the OpenTTD engine. The statements are a mixture between declarative statements and code, which is then bundled into a `.grf` file [38]. OpenTTD has strong limitations of `grf` compatibility, conflicts can easily occur [37]. Replacements with `NML` can be done with the explicit `replace` statement, which is a predefined block. Components of another `grf` can be disabled requiring target item identifiers [37]. The `grf` file format and available commands in `NML` are explicitly designed for Transport Tycoon, so it is not suitable for other game engines [34].

3.3.3 Skyrim Creation Kit

Skyrim is an open-world role playing game in the Elder Scrolls universe [55]. Skyrim has built-in modding capabilities with its "Creation Kit". It simplifies the creation of mod packs because the whole work flow and all possibilities are supported by its interactive GUI [50]. Creation Kit is a hybrid mod API, which allows data changes and to attach scripts in the Papyrus scripting language [52].

There are interactive dialogues that provide input elements for every possible action for the current object, script or asset. Bethesda provides tutorials on the Creation Kit wiki [51]. All the information about additions and changes is stored in `.esm` and `.esp` files, the former is a "master file", the latter a "patch file". These files contain records in binary format, designed to be edited with the Creation Kit. There have been some attempts to create replacement tools, for example "SkyEdit" [58].

The "patch files" may introduce new records or reference to existing ones to change properties. Records are simply overwritten, depending on the load order of the `.esp` files. The plugin loaded last has highest priority, when the same record is modified by multiple mods [59]. Although the modding utilities provided by the Creation Kit are very powerful, it is non-free software and is designed for the Creation Engine only [17]. Therefore it is not possible to reuse the tools for other games, except by reusing the ideas and implementing them again.

4 Design

The overview of components needed for the operation of nyan can be seen in the overview Figure 4.1. Files written in the nyan language are used as input. Their content is processed through a lexer and parser and is then stored in the nyan database. The database can be accessed over the nyan-API.

There are two public interfaces to the nyan framework: The nyan-API to interact with the nyan database and the input and configuration through records written in the nyan language.

This chapter will describe the concepts designed for the nyan framework. First, the overall idea is presented in Section 4.2, then the components and their roles are elaborated.

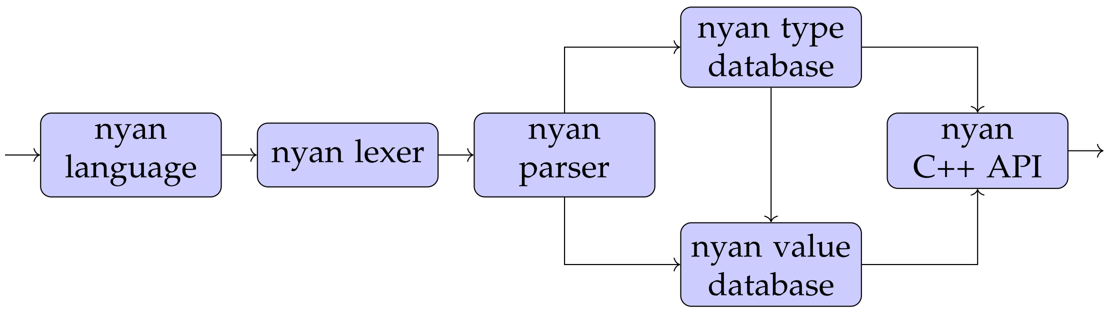

Figure 4.1: Overview of components

4.1 Engine and Content Separation

A game engine (see Section 2.1) is an application that provides the framework for building a game. The game content (its unique artwork, characters, sounds, levels, etc.) is independent from the engine and can be provided in different formats.

The format of that content data is defined by the engine, as well as the features it is able to simulate and display. The implemented algorithms define the feature set of the game engine, which types of games it is suitable for, the hardware requirements and the overall behavior.

The nyan language is designed to be used as one possible content format, which is provided to a game engine through the nyan database. The game engine loads its data schema into the nyan database. The schema describes the allowed data structures and value types. The game engine defines the schema so it can later access and update records by the structure, names and types it has declared through it.

As the nyan language is used for describing the engine configuration declaratively [27] so it does not provide the possibility to write code, there is no need to sandbox it and prevent code execution.

The game engine is responsible for setting up the data schema, loading data, changing data and handling query results obtained from the nyan-database. In particular this means the engine is responsible for the load order of possibly conflicting records: If changes are planned to be done on the same data record, the engine has to decide about the order. In practice, this is mainly determined by the game logic implemented in the engine. The progressing in-game-time and world state can provide the correct order for changes in the nyan database e.g. power upgrades.

It is important to keep in mind that the nyan framework merely provides a configuration tool and alterable state storage, all the algorithmic logic has to be implemented in the application that embeds the nyan framework.

4.2 Concept

The overall concept of nyan revolves around the possibility to access and modify the stored data at any time with little possibility for conflicts or ambiguity so that multiple change requests can be active at once.

The nyan database stores a value for a key. A key can be accessed by a human-readable name, to which alterable data can be assigned as a value. The idea is now that a result to a query of a key can be a combination of values, depending on how the keys are arranged. The database also stores the history of changes for to values. By the usage of a custom timestamp, any point in this history can be queried.

The nyan language is designed to be human read- and writable as it is the primary entity content creators will have contact with. This means it must be read and written intuitively.

In order for the application to access data in the database, data has to be organized according to a schema provided by the application. The schema defines the permitted structure and value types of data records. Hence, the schema must be loaded into the nyan database by the application before content can be added through the nyan-API. The published data schema equals the “content-API” of the application: Configuration can be performed on the structures and types defined by the schema.

Run-time code plugins for the engine may require schema extensions, for example for adding data fields to existing game objects so that they can store properties accessed by a scripting language.

The concept used for the nyan framework is to describe both the schema and content in the same language as the data records, which is the nyan language.

The approach for this unification is the storage of nyan-*objects*. Such an object stores an arbitrary number of *members*, which are key-value pairs. A member always has a *data type*, which restricts the kind of values to be allowed. Because a `nyan`-object simultaneously declares data types and their values, they are a unification of the concept of a "class" and an "object".

`nyan`-objects are organized in *namespaces*, which are created by the directory hierarchy of files where they are defined in. `nyan`-objects can be identified uniquely through their fully qualified object name (fqon), which is the combination of the directory structure and their object name. A fqon allows to reference objects unambiguously from independent mod projects. *Namespaces* and fqons are further presented in Section 4.6;

`nyan`-objects can inherit data from other objects. This inheritance allows creation of a data inheritance graph, where objects can inherit data from other objects and apply modifications to it. The modification is done by an operator and a value. This is a form of object inheritance [12].

When a member value of an object is queried, the result is determined by walking down the inheritance tree of the object hierarchy, starting at the root object(s). On the way, modifications are accumulated. This mechanism allows changes made to parents to be propagated to all of its children.

Modifications to the data are stored in the database in the form of patches. All of the possible changes are therefore accessible by name, and patches can change patches. The patch applications are triggered by the game engine, possibly according to its implementation of initial data loading, game initialization, game logic, etc.

All the modifications are tracked on a timeline, so that rollbacks to any previous state are possible. This is built into the design so predictions of the game state can perform changes in the `nyan`-database, and if those are no longer valid, one can go back to a previous point in time.

These features were chosen to allow easy modding and integration for games. Data packs can contain game content, stored in a directory hierarchy of `.nyan` files. Those data packs are essentially a mod pack for the engine, it provides or changes the configuration for the engine. The type system requirements of data eliminates errors during runtime of the game, as most of the problems can be detected at load-time of the mod. When this mod is activated (load order, point in time) has to be determined by the engine.

4.3 Input Language

The input language, called the `nyan` language, has to be portable and human readable. The storage format is plain text, so that they can be edited with arbitrary text editors. They could be loaded and presented in a fancy GUI, but such a program is not required.

The language is designed to be very compact, but readability is given priority over memory consumption. A major goal was to create the language in such a way, that data definitions and changes can be expressed without redundancy.

The main goal was to create it in a way that it is editable intuitively. For that, a syntax similar to Python [43] was chosen. The style was adapted for the concept of `nyan`, but remained "pythonic" [45]. Hence, it should be easy to read and write even for people with little or no programming experience.

Python's whitespace indentation is relevant for stating code blocks [42], this is also true for `nyan`.

In the following sections, relevant language components are introduced. This includes the definition of `nyan` objects, the allowed data types, the inheritance declarations, operators and value assignments and the definition of patches.

4.3.1 Object Definition

A `nyan`-object is a named group of key-value pairs. The name of the group is the `nyan`-object name. Key-value pairs assigned in this object are called *members*. Each initial definition of a member must be annotated with a type.

The member type is defined after the member name with `keyname : type = value`. The basic structure for an object definition is illustrated in Listing 5.

```
ObjectName():
    member_name : TypeName = value
    another_member : int = 123
    member_without_value : SomeType
    ...
```

Listing 5: Basic definition of a `nyan`-object and its members

The order of members does not matter and there must not be two members with the same name in an object. A member may have no value. An object containing a member with no value is called `abstract`. This is a way of forcing derived objects to specify a member value. This mechanism will be elaborated in Section 4.3.4.

Objects can also be defined within another object. The inner one is a *nested* object, and can contain nested objects itself. The purpose of nested objects is to allow further grouping, which is also supported by the naming schema for them: The name of a nested object is a combination of the name of its parent and the nested object name. This namespace creation and its reference procedure are presented in Section 4.6

4.3.2 Types

Types in the `nyan` language are introduced enable static error analysis. The allowed data types for a member are either *primitive*, *collection* or *object*. Primitive and collection types are built into the language, whereas object types are defined by the user.

Built-in primitive types are present in many programming languages, for example in Python [44]. The available primitive types in the nyan language are presented in Table 4.1.

Type	Example	Description
`text`	`"random text"`	Immutable string constant
`int`	`1337`	Numeric type for integers
`float`	`42.235,inf`	Numeric type for floating point numbers
`bool`	`True,False`	Truth type for boolean operations
`file`	`"./name"`	Filename, relative to the file this value is defined in. If the path is absolute, returned as-is.

Table 4.1: Available primitive types in the `nyan` language

The collection types allow to unite multiple values of the same type in one container. Valid collection types are `set` and `orderedset`. Collections have a element type, which is declared after the collection type. Example definitions of members with primitive and collection types are provided in Listing 6.

```
SomeObject():
    some_member : text = "some boring string value"
    another_member : int = 42
    third_member : float = 123.456
    fourth_member : bool = True
    fifth_member : file = "./directory/graphic.png"

    set_member : set(int) = {13, 37, 42, 13, 42}
    orderedset_member : orderedset(text) = o{
        "this", "is", "this", "a", "test"
    }
```

Listing 6: Primitive and collection member types

The difference between `set` and `orderedset` is only observable in query results. An ordered set strictly keeps its element order, elements that already were in the set are moved to the end when a new element is inserted. The resulting value of `orderedset_member` from Listing 6 therefore is `["is", "this", "a", "test"]`, the value of `set_member` is `{42, 13, 37}` but the numbers could also be in any other order. The right hand side value also has a different creation, a `set` value is created by `{...}`, an `orderedset` value is created by `o{...}`. A set can also be assigned an ordered set value, but not the other way round, while dropping the order of elements is no problem, creating it when none was intended would be ambiguous.

A `list` type was omitted deliberately from the design to avoid the complexity needed for list combinations. If a sensible way of merging lists is found, lists can easily be added to the `nyan` framework later.

The third possible type for a value is a `nyan`-object. This means an object can be used as a value for a member or in a collection of values in any other object. The type name of a `nyan`-object is its object name. If an object `B` inherits from an object `A` (as presented in Section 4.3.3), then `B` is also of type `A`, and can therefore be used as a value for a member of type `A`.

There are two built-in object type names: `Object` and `Patch`. The former denotes any `nyan`-object to be stored, the latter specifies any patch object that can be stored. Patch objects will further be explained in Section 4.3.5. An example of the application of objects as types can be seen in Listing 7.

```
ValueObject():
    funny_member : text = "not funny"

TestObject():
    # can hold any object as value:
    generic_member : Object = ValueObject
    # can only hold ValueObjects:
    boring_member : ValueObject = ValueObject
```

Listing 7: Primitive and collection member types

4.3.3 Inheritance

`nyan`-objects can inherit from one or multiple parent objects. Inheritance is transitive, so a child object is of type of all its parent objects, and the parents of those, and so on. An example for the syntax to declare inheritance relations is presented in Listing 8.

```
ParentObject():
    ...

ChildObject(ParentObject):
    ...

RandomObject():
    ...

GrandchildObject(ChildObject, RandomObject):
    ...
```

Listing 8: Object inheritance syntax

Inheritance is used to provide a child with all the members of its parent. This mechanism enables reuse of parental base values as well as parent-relative modification of members. How the value can be updated depends on the type of the member. Each type has a number of allowed operators, presented in Section 4.3.4. A child object therefore gains access to all members of all parents and can build upon them by operations specified by the member type.

There can be more than one parent for an object, which allows multiple inheritance, as presented in Section 2.4. The problems introduced with this mechanism are discussed in Section 4.5.

Inheritance allows to assume that a child object has a member of a parent: The member was inherited and therefore is also present in the child. An illustration can be seen in Listing 9, where each `ValueDemo.member` set entry can be queried for the value of `a`, it is prescribed by `BaseObject`.

```
BaseObject():
    a : int = 0

ChildObject(BaseObject):
    a = 1
    b : int = 10

OtherChild(BaseObject):
    a = 2
    c : int = 20

ValueDemo():
    member : set(BaseObject) = {ChildObject,
                                 OtherChild}
```

Listing 9: Subtyping used for object values

4.3.4 Values and Operators

Each member in an object has a type, as described in Section 4.3.2. When inheriting (see Section 4.3.3), the value of the current object can be defined relatively to the parent object. This means that when the current object's member value is requested, it is constructed by evaluation of the parent's member value followed by the application of a type-specific operator and value. All member types support the assignment operator =. Other allowed operators are listed in Table 4.2.

Objects can only be assigned as values if they are not `abstract` (that is, they define or inherit a member that has no value) and their type matches the member type. The type constraint guarantees access to members defined by that object or any of its children. Members do not vanish, so they can be accessed from any child object. As an object used as value must not be abstract guarantees, a request to the member value can be done, and in turn forces the

Type	Operator	Description
any	`=`	Value override
`text`	`+=`	String appending
`int` and `float`	`+=, *=, -=, /=`	Arithmetic operations
`bool`	`&=, |=`	`and` and `or` operation
`file`	`= "./moist/cake"`	Assignment only

`set(type)`	`= {value, val, ..}`
	`+= {..}, |= {..}`
	`-= {..}`
	`&= {..}`
		Assignment
		Set union
		Element removal
		Set intersection

`orderedset(type)`	`= o{value, val, ..}`
	`+= o{..}`
	`-= o{..}, -= {..}`
	`&= o{..}, &= {..}`
		Assignment
		Add values to end
		Element removal
		Keep only those values

Object	`=`	Assignment

Table 4.2: Operators for each type

object to be initialized. That way a content API (i.e. abstract objects) can force values to be set.

This also allows that the API is further refined, for example by a mod that introduces an extension to an existing API, and still forces any usage of it to fill in values that the behavior/engine code can then use for the game simulation or any other component to be configured.

4.3.5 Patch Definitions

Values would remain the same forever if there was no way to change them. In order to modify them, `nyan` uses *patches*. A patch is a `nyan`-object which specifies changes to members of another `nyan`-object, which is its patch *target*. As a patch is a `nyan`-object, it can be target of another patch. Hence, the system allows to create mods for mods.

The *activation* or *application* of a patch is triggered via the `nyan`-database-API (see Section 4.9), thus the decision is not done in the `nyan` framework, but in the software that uses it. One patch can be applied multiple times, then it will perform all the changes once again.

A patch targets exactly one `nyan`-object. This target can not be changed. If the target could be changed, type checks would need to test for soundness of value changes in anticipation of a different target as well.

It is possible to apply the patch to a child of the targeted object, though. It is guaranteed to possess all the members and types of the original target.

The patch target is written down as `<target>` in the object definition. An example is shown in Listing 10. An object which defines a target or inherits one from a parent is a patch.

```
RandomObject():
    random_value : int = 0

SimplePatch<RandomObject>():
    random_value += 1
```

Listing 10: Patch definition

A patch `P` that targets object `A` is similar to an object `B` that inherits from `A`: If `P` and `B` have the same modifications for `A`, then `B` will evaluate to the same values as `A` will after the patch was applied. After the `P`-application, `B`'s values will of course be updated as well.

An illustration of this is shown in Listing 11, where `B.random_value == 15`. The evaluation of `A.random_value` will be `15` after `P` was applied. As `B` is inheriting from `A`, which was just changed, `B.random_value == 20`. Thus a patch behaves like inheritance, except the target is updated to the new value.

```
A():
    random_value : int = 10

P<A>():
    random_value += 5

B(A):
    random_value += 5
```

Listing 11: Update logic similarity between inheritance and patching

A patch can inherit from any number `nyan`-objects (including other patches) to specialize it further. A patch target can only be specified if no (transitive) parent did specify a target object. All children of a patch will implicitly affect the same patch target. That way, patches can be further specialized but the target can't be changed.

When a patch is applied, all members present in the target object are updated according to the changes of the patch. The patch is linearized first (see Section 2.4), and the members are updated for each parent in the linearization result. That means if the patch has parents, the changes requested by them are applied first, the changes of the patch afterwards. This approach implements the intuition about further specializing patches by inheritance: The most specialized patch is applied last.

A special feature of patches is their ability to change the inheritance parents of a `nyan`-object. This is introduced to allow the injection of a parent in between an existing parent-child relationship. This is denoted by the `[NewParent+, +OtherNewParent, ...]` syntax. The position of the `+` specifies if the object is added at the front (`Name+`) or the end of the current parent list (`+Name`).

If the target has parents [A, B] and we apply [+C, +D, E+], the result is [E, A, B, C, D]. The linearization must be valid after the parent addition by the constraints presented in Section 2.4. In Listing 12 an example for the mechanism of injecting a new parent is provided.

```
RootObject():
    magic_value : int = 9001

ChildObject(RootObject):
    magic_value -= 1

InjectedObject(RootObject):
    magic_value -= 7661

InjectIt<ChildObject>[InjectedObject+]():
    magic_value += 2
```

Listing 12: Inheritance modification by patch

The initial query result of `ChildObject.magic_value == 9000`. When the `InjectIt`-patch is applied, `InjectedObject` will be the new direct parent of `ChildObject`. The value 1 of `ChildObject.magic_value` is also modified. It is increased by `+= 2` to 3. The resulting member operation therefore is `-= 3`. When `ChildObject.magic_value` is evaluated again, the result is $9001 - 7661 - 3 == 1337$.

Besides values, patches may also override operators in the target object. The example showed the update of value 1 to 3 by +=, its operator remained the same though (−=). To allow the override of the operator and member value, `@` is used. It instructs to replace the operator and value of the target object to be replaced by the operator and value in the patch, that is, the part after the `@` is copied, as illustrated in Listing 13. When `FixOperator` is applied, `ChildObject.magic_value` will evaluate to $9001 + 999 = 10000$. When `FixOperatorFix` is applied and `FixOperator` is applied after that, `FixOperator` will have stored `@*= 2`, and therefore updated `ChildObject` to store `*= 2`. Hence, `ChildObject.magic_value` evaluates to $9001 * 2 = 18002$.

The number of `@` modifiers is limited by the depth of the patches: One `@` can only be added if the target object is a patch. In the example, the maximum number is 2, if there were more `@` modifiers, the non-patch `ChildObject` would get a leftover `@`.

Patches are applied in `transactions`, which are further described in section 4.8. If the addition of parents induces linearization problems (see 2.4), the transaction will fail. The only other way to avoid those failures would be prohibiting inheritance changes at runtime: We cannot foresee the order or effective activation of patches. They may be valid if another patch was never

```
BaseObject():
    magic_value : int = 9001

ChildObject(BaseObject):
    magic_value -= 1

FixOperator<ChildObject>():
    magic_value @+= 999

FixOperatorFix<FixOperator>():
    magic_value @@*= 2
```

Listing 13: A patch can replace operators

activated, but a failure occurs if it is applied. An example for such a construction is in Listing 14, where either `A` or `B` can be applied, but after that, application of the other is impossible.

```
ObjA():
    ...

ObjB():
    ...

A<ObjA>[+ObjB]():
    ...

B<ObjB>[+ObjA]():
    ...
```

Listing 14: Mutual exclusion of patch applications

The avoidance of such conflicts is the responsibility of the creators: Authors of mods have to synchronize themselves to prevent conflicts.

All patches are checked for sanity after all objects are loaded. The type compatibility for their changes are verified with the target object.

Patches may add new members to their target, but this cannot be anticipated. Therefore, no other object can assume it was added. If an unknown member is modified, the patch can't be loaded. A newly-added member is unknown to other patches.

4.3.6 Language Grammar

The `nyan` language can be parsed with the grammar depicted in Listing 15.

```
⟨alpha⟩        ::= 'a-zA-Z_'
⟨alphanum⟩     ::= 'a-zA-Z0-9_'
⟨identifier⟩   ::= ⟨alpha⟩ ⟨alphanum⟩*
⟨namespace⟩    ::= ⟨identifier⟩ ('.' ⟨identifier⟩)*
⟨number⟩       ::= '-'? '0-9'+ ('.' '0-9'+)?
⟨operator⟩     ::= '@'* ('-+*/|%&' '='? | '=')
⟨file⟩         ::= ( (⟨import⟩ | ⟨object⟩) \n )*
⟨import⟩       ::= import ⟨namespace⟩ (as ⟨identifier⟩)?
⟨object⟩       ::= ⟨identifier⟩
                   ('<' ⟨namespace⟩ '>')?
                   ('[' (⟨parent-add⟩(',' ⟨parent-add⟩)*)? ']' )?
                   '(' (⟨namespace⟩(',' ⟨namespace⟩)* )? ') :\n'
                   INDENT (⟨obj-content⟩\n)+ DEDENT
⟨parent-add⟩   ::= '+' ⟨namespace⟩ | ⟨namespace⟩ '+'
⟨obj-content⟩  ::= ⟨identifier⟩ ':' ⟨type⟩
                |  ⟨identifier⟩ ⟨operator⟩ ⟨value⟩
                |  ⟨identifier⟩ ':' ⟨type⟩ ⟨operator⟩ ⟨value⟩
                |  ⟨object⟩ | '...'
⟨type⟩         ::= ⟨namespace⟩ | 'set(' ⟨namespace⟩ ')' | 'orderedset('
                   ⟨namespace⟩ ')'
⟨value⟩        ::= ⟨identifier⟩ | ⟨number⟩ | 'o'? '{' ⟨value⟩ (',' ⟨value⟩)* '}'
                   ''' ANY ''' | '"' ANY '"'
```

Listing 15: Grammar for the `nyan` language, the entry is ⟨*file*⟩

Special handling is needed to figure out when `INDENT` and `DEDENT` are recognized. Those tokens have to be emitted if the indentation of the line changes, the level is designed to be exactly 4 spaces per level. This means that the token stream must already track the indentation level whitespace.

The value handling also deserves some special attention, as only two line wrap rules are allowed. If a line gets too long, it is usually continued in the next one. The indentation of wrapped lines and the indentation of the closing bracket is enforced by the lexical analysis in order to guarantee a uniform visual appearance. Examples for this are provided in Listing 16.

```
IndentDemo():
    # good: no newline after {, content aligned at {
    member : set(int) = {1, 3, 3, 7,
                         4, 2}

    # good: newline after {, content indented +4
    orderedmember : orderedset(int) = o{
        1, 2, 3, 1,
        2, 3, 2
    }

    # wrong: indent of 9001 not aligned at {
    nope : set(int) = {2, 3, 5,
        9001, 17,
    }

    # wrong: indent of } not at level of 'nope'
    nope : set(int) = {
        1, 2, 3, 4}
```

Listing 16: Indentation enforcement

4.4 Type System

Although values can be changed with patches (see Section 4.3.5), value types can never change. Because of that, the type system is independent of the database state. It is only provided with new information when `nyan`-objects are loaded. Type compatibility is checked during a request to load data into the database storage.

All of the type information handled by `nyan` is stored at runtime. The application that exposes its API feeds this schema information to `nyan` at every startup. This allows the schema to be extended by mods which provide new scriptable features.

A `nyan`-object's type name is its object name. Types of object members must be declared when a member is initially created. This type is reused for all patches and child objects that also operate on that member. When a child object is created, all the types of used but not initially declared members are inferred from the linearized inheritance (explained in Section 2.4). To obtain the type of a member of a patch, its originating member must be found by following the patch target, which in turn might be defined in a parent of the current object (as patches can also inherit). The procedure is to linearize the parents and in each, try to find the initial definition of the member. If there are multiple results, the conflict needs to be resolved in the manner Section 4.5 explains.

Once the member type is found, the operator and value are checked for validity. The allowed operators are listed in Table 4.2.

Because the schema is expressed through (optionally "abstract") objects with type information, the application can assume that queries to those object and their members will work and are type-correct. In practice, an application can provide an API like in Listing 17.

```
# API from application
ConfigAPI():
    Picture():
        alt_text : text = ""
        path : file

    View():
        background_image : Picture
        name : text

    volume : int = 50
    current_view : View

# API usage from an application plugin
HomeScreen(ConfigAPI.View):
    CatPicture(ConfigAPI.Picture):
        alt_text = "cute kittenz"
        path = "./assets/cute.png"

    background_image = CatPicture
    name = "Main screen"

# Activation of the home screen
ActivateHome<ConfigAPI>():
    current_view = HomeScreen
    volume = 80
```

Listing 17: API guarantees type safety

This API forces values to be set by the API user (`path`, `name`, `current_view`). Otherwise, an object cannot be used on the right hand side, as explained in Section 4.3.4.

The application can now query `ConfigAPI.volume`, the result is guaranteed to be an `int`, and it will have value `50`. `ConfigAPI.current_view` will have no value at first, the value will be assigned once the application decides that the `ActivateHome`-patch is activated.

Afterwards, the value of `ConfigAPI.volume` is changed to `80` and a query to `ConfigAPI.current_view` results in a `nyan`-object of type `ConfigAPI.View`. On that object, it is possible to query for `object.name` (which is of type `text`) and `object.background_image` (which is a `ConfigAPI.Picture`-`nyan`-object), which can in turn be queried for it's alternative text (`alt_text`) and the image path (`path`).

Therefore `nyan` makes is possible to create a type-safe object-oriented configuration API.

4.5 Multiple Inheritance

The nyan language allows an object to inherit from more than one parent object, this is called *multiple inheritance* [11][8].

Multiple inheritance is achieved by specifying multiple parents in a nyan-object definition. It is then composed of multiple sources, and it is possible to add a parent at runtime through the patches (see Section 4.3.5) applied in transactions from Section 4.8. An application of such a composition for openage would be that of a deer, which is a Unit, Huntable and ResourceSpot. To clarify which parent is meant for an overridden member, its name can be prefixed with the object name, like in Listing 18.

```
Deer(Unit, Huntable, ResourceSpot):
    # explicit targeting
    Unit.sprite = "./assets/deer.png"
    Huntable.flee_type = DeerFlee

    # implicitly targets ResourceSpot.resource_type
    resource_type = Food
```

Listing 18: Member name qualifications

Naming ambiguities can occur when multiple parents define a member with the same name. The data structure is no longer tree-based but rather a graph. There can be two reasons for a name conflict: Two independent objects declare this member, or it originates from a common parent. An example for both situations is shown in Listing 19.

```
Base():
    setting : int = 0

Intermediate0(Base):
    setting += 1
    property : int = 10

Intermediate1(Base):
    setting *= 2
    property : int = 20

Independent():
    setting : int = 100
    property : int = 1234

Mixed(Intermediate0, Intermediate1, Independent):
    setting += 3    # which origin is meant?
    property += 5
```

Listing 19: Ambiguous member name due to multiple inheritance

To resolve the conflict, the origin of the member is determined first. This is done by walking over the parent linearization list. If there is only one member definition (i.e. member name with its type), the member is from a common parent (in the example, that would be `Base.setting`). If there is more than one definition, the conflict can only be resolved by additional member name qualification that prefixes an object where the member name is still unique. In the example, both `Mixed.setting` and `Mixed.property` must be qualified so the member origin is clear. In the example, `Mixed.setting` must be specified as `Base.setting` in case this member was intended to be modified. `Mixed.property` must be prefixed with one of `Independent`, `Intermediate0` or `Intermediate1`, depending on the desired target.

The conflict detection is performed for each object when they are loaded. The linearization (see Section 2.4) of `Mixed` is `[Mixed, Intermediate0, Intermediate1, Base, Independent]`. For each member, this list is walked to gather objects which define a member with this name. For `setting`, the result is `[Base, Independent]`, for `property` the result is:

`[Intermediate0, Intermediate1, Independent]`

Since each list has more than one element, the conflict must be resolved by selecting one element of each list as the name qualification prefix. This qualification does not change the member value of the prefix object, instead this just indicates which origin member is meant.

The inheritance mechanism in combination with patching can be used to inject a new custom object in between existing inheritance hierarchies, which is useful to add new members to existing objects, as illustrated in Figure 4.2.

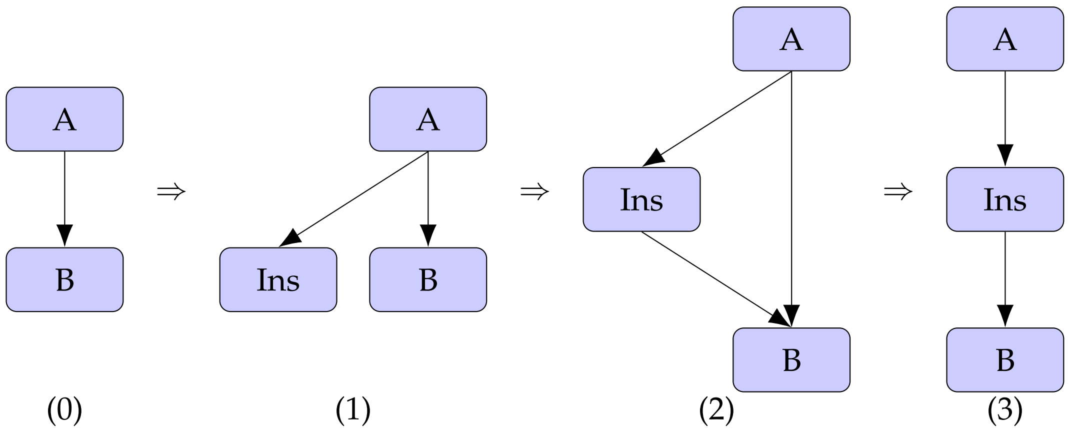

Figure 4.2: Parent object injection through multiple inheritance

In order to reproduce this example in `nyan`, the parent is injected by a patch (see Listing 20). The `Activate`-patch is applied whenever the application decides, for example when the plugins are loaded and initial patches are enabled.

Without the mod, state (0) is active. When the mod-`nyan`-data it is loaded, the new object is just added and available in the database but is not yet used (1). When a transaction with `Activate` is committed(see Section 4.8), (2) will be

```
# original state
A():
    ...

B(A):
    ...

# loaded by a mod
Ins(A):
    ...

Activate<B>[Ins+]():
    ...
```

Listing 20: Injection of new parent through a patch

the active state. The linearization of this will result in (3), because `C3(B(Ins, A))` results in `[B, Ins, A]`, which is the desired injection of a new "middle" parent.

4.6 Namespaces and Importing

Namespaces in the `nyan` language are used for grouping object names and directory hierarchies with files. They allow to organize data hierarchically on your filesystem. A namespace is a list of name components separated by a dot. A `nyan` file name implies its namespace. That means the filename must not contain a "`.`" (except in the `.nyan` suffix) to prevent naming conflicts. This namespace scheme is similar to the one used in Python [43]. The application decides what the root of this namespace is. It may strip or add any elements to a namespace if necessary, for example to transparently provide access to `.nyan` files stored in a compressed archive. The application could prefix the identifier of the mod pack to all the `nyan`-objects in it, so their references are unique.

For each mod pack, the application determines a filesystem root, which `nyan` uses for its import base. The path, relative to this custom root, is used as the first part of the unique namespace name. Object definitions in this `.nyan`-file are prefixed with this namespace.

```
# filename:  unicornmod/units/fluffy.nyan
# namespace: unicornmod.units.fluffy
Rainbow():      # top-level object
    Goldpot():  # nested object
        amount : int = 9001
```

Listing 21: fqon creation from filename and object

An example for such a fqon-creation is in Listing 21. The full name of each object is its "fully qualified object name (fqon)", which can be referenced from any other mod pack or from the application distinctly. In the example, this would be `unicornmod.units.fluffy.Rainbow.Goldpot`.

This name is rather impractical for writing `nyan`-files, mainly because of its length. Because of this, the referenced name is searched from the innermost scope towards the namespace root until a matching object is found. Ambiguous names are therefore resolved by picking the object name that is "closest" in the current namespace, because namespaces towards the root are only checked if no name matches for the current one. A referenced `nyan` object name can therefore be shadowed by ambiguous replacements which have higher priority in the search from the current namespace scope towards the namespace root.

The namespaces are not always available for reference, because they need to be imported first. The reason is that this allows to follow all imports of files so unknown files can be opened and parsed. Known files are not imported again. Imports are done with the "`import ...`" statement, at the top level of a `.nyan`-file:

```
import unicornmod.units.fluffy
```

It is possible to create convenience aliases of imported namespace names with the "`import ... as ...`" statement.

This instructs the `nyan` framework to import the desired namespace and assign an alias to it (right side), which expands to the left side when used. In practice, this alias mechanism can avoid long fqons, as shown in Listing 22.

```
Candy(unicornmod.units.fluffy.Rainbow.Goldpot):
    amount = 10

# is the same as:
import unicornmod.units.fluffy.Rainbow.Goldpot as Pot

Candy(Pot):
    amount = 10

# which is also the same as:
import unicornmod.units.fluffy as fluffers

Candy(fluffers.Rainbow.Goldpot):
    amount = 10
```

Listing 22: Import with custom aliases

Object inheritance can never be cyclic. A member of an object A may refer to an object B which has a member pointing to object A. Cyclic value references are therefore allowed. The order of declared `nyan`-objects in a file does not matter. An object name can be used even if it will be declared later in a `.nyan` file. This works because object member values are always “pointers”.

The graph structure of available objects is loaded before their members are assigned their values. The compatibility for the value type can therefore be tested when the members are set up after the object type graph creation. This means there are implicit forward declarations.

4.7 Database Views

In many competitive games there is a different game state for every player, each team and the general game. For example, this can be required because an effect for a team affects all players in it (team bonuses) but research done by a player does not necessarily affect other players.

So it is required that teams and players have a distinct view on the database state, but those views still depend on each other. The dependencies are a tree, a possible example is illustrated in Figure 4.3.

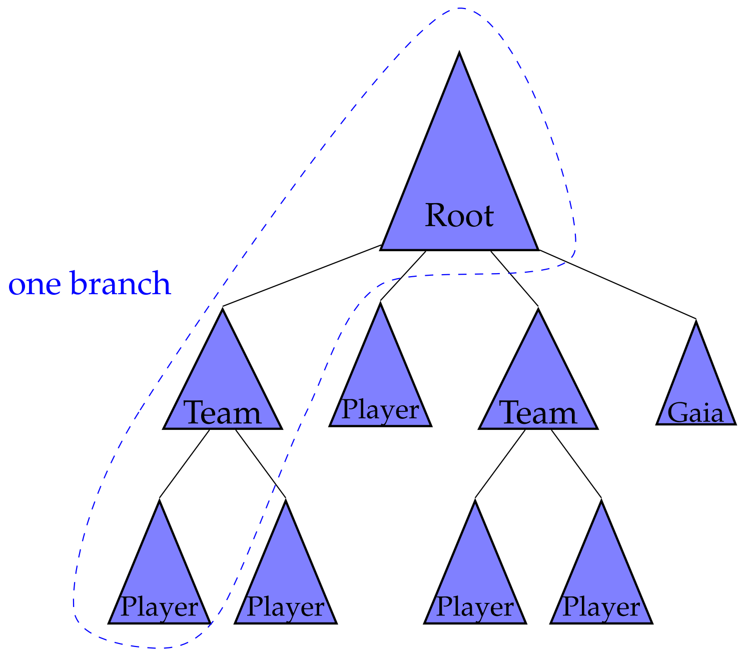

Figure 4.3: Different state views for different players

In the `nyan` database, views are used to separate multiple change histories that started from the same base state. A view always has either a parent view or the root database state as its base.

Each view can store a different database state, but a parent view state change will lead to a state change in its child views.

When a view is created, its state reflects the initial database state. The initial state is the collection of all loaded `nyan`-objects, without any patches applied.

New nyan-objects can only be loaded into the root database, all changes on them can only be performed in views. When a change is committed through a transaction (see Section 4.8), the data modifications are performed in the view. The changes are also propagated to all depending views. A team bonus can therefore be implemented as a set of patches that is applied in the view for the team. All players in the team are subject to the change, other teams remain unaffected.

Views are used for the storage of changes in the database. If the views don't depend on each other, they can operate completely separated.

4.8 Transactions

Transactions are used to perform changes in a database [19] (see Section 2.3). In nyan, one or more patches are packed in a transaction so they are all activated at once or, if errors occur, none of them is applied.

When a patch is added to a transaction, it is possible to customize the patch target. A patch stores its target object already, which is used by default. The custom target must be an object which is also of type of the original target. This is required so the new target has all the properties of the default target.

A transaction must be created for a view (see Section 4.7), as only views can hold state changes through patches. Additionally, transactions are performed at a custom point in time.

The time specified for a transaction has several purposes: A modification can be planned without having any effect before a specified point in time. Transactions that are conducted after this point in time are dropped. If they are still valid, the application must recommit the changes. If two transactions are scheduled to happen at the same time, their changes are consolidated in the order of their commits.

Each view has a separate transaction history, but a parent view will always propagate the transaction to a child view. A child view can then add its own transactions with a later timestamp, but once the parent commits to a timestamp before that, the child's transactions after this timestamp will be deleted.

The propagation mechanism can be refined, but it would still mean that an earlier transaction must be taken as the base for transactions after that. Currently, these have to be activated manually again, instead of being replayed on top of a new base internally by nyan.

The mechanism of committing transactions at a predicted timestamp is used if a game engine uses the nyan database to predict a future game state. The event-driven design of openage uses this to store ahead-of-time calculations of the game state [60]. A possible situation for predictions done in openage is illustrated in figure 4.4. State C and E equal "reality", H, F and G are predictions of the future. An example prediction could be that in 10 seconds the research to make villagers stronger will be finished. The patches for this will be active from the point in time their transaction was predicted for. The patches' effects are included in calculations for queries after that timestamp.

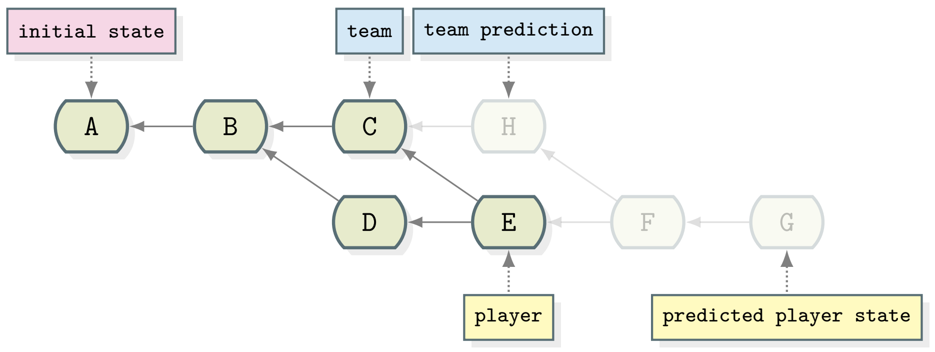

Figure 4.4: Game state change history and predictions over time

The interpretation of the point in time has to be done by the application, the nyan database only provides a sequential integer number for transaction ordering. Those could be used for game ticks, nanoseconds or any other suitable measurement unit. The decision when a prediction is created or invalidated and the interpretation which time stamp is the "current" state is up to the application [60].

Patches may add new parents to an object, for example to achieve injecting an object in an existing inheritance relation like presented in Section 4.5. It is not possible to remove parents from an object: If this was allowed, members could vanish. This would lead to many runtime errors, therefore this is not allowed.

Due to strict type checking for member values when objects and patches are loaded, the only problems that can not be detected earlier is C3 linearization errors. This means that either there is cyclic inheritance or that no linearization can be found, caused by a patch in a transaction. A simple example for this was presented in Listing 14.

Before a transaction is stored permanently in a view, the nyan database checks for linearization problems. If the state would not be consistent after the transaction was stored (that is, the C3 algorithm can't find valid linearizations for all objects affected by inheritance changes), the transaction fails and none of the patches is applied.

4.9 Application Interaction

The nyan database is designed to be embedded into an application as a shared library. The interface to this library is simple, making it easier to keep stable. All of the nyan data structures are created at runtime, which means the API to the application has to make this as convenient as possible.

The interface has to provide handles for views (from Section 4.7), transactions (from Section 4.8) and nyan-objects (from Section 4.3.1).

The identification of nyan-objects must be done through strings of their fqon. The same is true for object members, which have to be referred to by their name.

The basic idea is that the application loads the data schema into the `nyan` database so it can then perform requests on this structure and names.

This schema also defines the data types, so the application can use this to perform type conversions to its native type system: At compile-time of the application, the schema and the access of data in the application must be aligned. Therefore type conversions are possible, as long as `nyan` guarantees type safety.

Before a query result can be converted to the application's native language type, the query must be performed. Queries operate on a `nyan`-object handle and evaluate one or more of its members.

The first step in the calculation is the linearization of the parents of that object (see Section 2.4). Then, the resulting list is walked until an assignment operator is found (=). The iteration is stopped because all changes after that would be overwritten by that =. The assigned value is copied as the origin for the query result. Then, the list of objects is walked backwards, starting with at the object which had the assignment operator. For each object, the operation is performed on the query result, until the list head is reached again.

This result can be cached and must be marked invalid if the value changes in one of the objects of the inheritance graph through a patch. The result can then either be calculated again once it is queried, or the result can be calculated as soon as the patch is applied. The decision here is a trade-off between fast patch applications or fast value queries.

A query is performed for a specific point in time. This is required because the transaction history is recorded. Each of the visited object's state is then chosen based on the point in time for a query. By default, transactions and queries operate at time `t = 0` so the database behaves like it had no state history.

5 Implementation

The implementation is published under the GNU LGPLv3 license. The code repository can be accessed through GitHub [22].

The language of choice for the implementation is `C++` [57], as it is a fast system programming language used in many applications and games. The implementation was done using the `C++14` language standard.

The rest of this chapter will provide insights into the structure and implementation details of `nyan`. The implementation closely follows the design described in Section 4 and consists of the modules shown in Figure 4.1.

The `nyan` database code for embedding in an application is compiled in a shared library, `libnyan.so`. The only header an application needs to include is `nyan/nyan.h`.

In the following sections, key components of the implementation are presented. First, the implementation of the language parsing is explained in Sections 5.1 and 5.2. Section 5.3 presents the creation of the initial database state. How database states are stored is explained in Section 5.4. The creation of new states in views through transactions is elaborated in Section 5.5, followed by the implementation of data queries in Section 5.6.

5.1 Lexical Analysis

The lexical analysis is performed by a tokenizer, which splits up the contents of a `nyan` file into tokens. A token is made of a group of characters and its type. The list of tokens emitted is then organized in the abstract syntax tree (see Section 5.2).

The token stream, consisting of all tokens of a `nyan` file from top to bottom, is created by the flex lexer [25].

All defined tokens are listed in Table 5.1.

The lexer ignores comments, that is all parts of a line beginning with `#`. All whitespace, except in strings and for line indentation, is ignored, too. The tokens will be matched greedily. That means as many characters as possible are put into one token.

The indentation level is determined after an end of line was reached. This number of leading spaces is then measured, and differences to the previous line are emitted as `INDENT` and `DEDENT` tokens for every group of 4 spaces. This is done in the `nyan::Lexer::handle_indent` method.

In this method, the correct indentation is enforced. Regular block indentation is a multiple of 4 spaces, but it is more complicated for the correct bracket

Token	Definition	Description
AS	as	
AT	@	
COLON	:	
COMMA	,	
DEDENT		Indentation level decrease
DOT	.	
ENDFILE		End of file
ENDLINE	\n	End of line
ELLIPSIS	...	
FLOAT	-?[0-9]+\.[0-9]*	
FROM	from	
ID	[A-Za-z_][A-Za-z0-9_]*	Letters and numbers
IMPORT	import	
INDENT		Indentation level increase
INVALID		Unknown content
INT	(-|0[xX])?[0-9]+	base16 and base10 numbers
LANGLE	<	
LBRACE	{	
LBRACKET	[	
LPAREN	(	
OPERATOR	[-+*/|%&]|[-+*/|%&]=|=	
PASS	pass	
RANGLE	>	
RBRACE	}	
RBRACKET	]	
RPAREN	)	
STRING	"(\.|[^"])*"	

Table 5.1: All possible tokens in the nyan parser

continuation levels. As designed in Listing 16, the correct level depends on the wrapping style, which is analyzed in `nyan::Lexer::track_brackets` and the corresponding level is stored on a indentation level stack. For closing brackets, the level is verified and removed from the stack.

5.2 Abstract Syntax Tree

The abstract syntax tree (AST) is built from the token stream, created by the lexer in Section 5.1. The AST structure directly follows the grammar from Listing 15.

The root AST object is of class `nyan::AST` (from `ast.h`), which then contains all `ASTObjects` and `ASTImports`. The `ASTObject` stores its name, a potential patch target, specification of possible inheritance changes, its parents, its members and other `ASTObjects` declared within its scope.

The decision how the AST is built up is done by "looking ahead" upcoming tokens, which then decide the path in the grammar structure. This means the parser for `nyan`, implemented within the AST buildup, is a left look-ahead parser (`LL(k)`) [48]. For the `nyan`-grammar `k = 2`, because if an `o` identifier is encountered for a member value, it may either be an object identifier or a set definition, if it is followed by a {.

While the AST is built up, many sanity checks are performed, but mainly the allowed order of tokens is verified. Some values are converted from their textual representation in the `nyan` file to `C++`-native data types like `enum`s and `int`s. All remaining text is stored as `std::string`.

The resulting AST now matches the grammar and can be processed further in multiple passes for the creation of the initial state.

5.3 Initial State

The application has to create an instance of `nyan::Database`, which is the entry point for the `nyan C++` API. In a `nyan::Database`, the initial state is created. The initial state is the collection of all `nyan`-objects with all their values and relations, without the application of any patch. To load content into the database, the application has to call `Database::load`, which expects an initial filename and a function that can provide files.

The function allows the application to provide the `nyan` database any file at any location, which may or may not be a real file on the filesystem: It can be used for transparently accessing compressed file archives or to provide file content over network from a server. The abstraction can be done by implementing specializations of the `nyan::File` class, which then perform the correct calls to access file content. This way, applications can load files into the `nyan` database no matter how they are stored. The `nyan` database calls this function whenever a file is requested. The first request will be done for the filename that was passed into the `load` call, subsequent ones for files included from there.

The creation of the initial database state is conducted in two phases.

The first phase is the AST creation and file imports. The initial file name is put into the set of namespaces to import. This set will then be processed while it has elements: First, a namespace name is taken from the set. Next, the file that matches this namespace is opened and parsed (see Sections 5.1 and 5.2). Imports in this file are put into the set of namespaces to be loaded. Then, the next import is processed the same way. Files are not imported again if their load succeeds. Cyclic imports are allowed.

The second phase consists of multiple passes on each file's AST. In the first step, the type database is filled with all known object names. These are constructed from the file namespace and the object name, thus, their fqon. No other information is stored at first. In the second pass, the type database can be filled with more information about each object. All available object names are already known by now, so all references to objects can be checked and expanded to their fqon. Hence, the patch target, inheritance change and parent object names can be extracted and stored: The patch target and inheritance changes are written to the type database, the object parents are stored into the initial state of this object. The former can't be changed at runtime, while the latter can, so they are stored at different locations. From the object-has-parent relation follows object-has-child. This information is deduced and stored. Subsequently, type information about object members is gathered and stored into the type database. Object references (as member type) are expanded in this step as well. The object name expansion algorithm is shown in Listing 23.

```python
def find_namespace(current_ns, search_ns, type_db, aliases):
    """
    current_ns and search_ns are lists of identifiers
    current_ns: the namespace an object was referenced in
    search_ns:  queried object name (a possibly cut fqon)
    type_db:    knows all valid fqons
    aliases:    a dict from identifier to full namespace

    returns the fqon for the search_ns, else it errors
    """
    search_base = current_ns
    while True:
        result = search_base.extend(search_ns)
        if result in type_db:
            return result

        # when root is reached, do alias expansion
        if len(search_base) == 0:
            result = aliases.get(search) or []
            result.extend(search)
            if result not in type_db:
                raise Exception("unknown name: " + search)
            return result

        search_base.pop()
```

Listing 23: Object name expansion algorithm to get an `fqon`

As a second step, all objects are now linearized (see Section 2.4). The results are stored in the initial object state store.

The third step is linking type information through inheritance hierarchies. The patch target is propagated for objects that inherit from a patch. The member type information object from the initial member definition is linked into the member of each child object that performs operations on it. The initial definition does not need to be searched every time then. Sanity checks for type conflicts, type re-definitions and inference problems are performed coincidentally. After this, the basic skeleton for filling in member values is done.

The fourth step fills in those values. It can be done only now as type information was not propagated before step three. For each object's initial state, one entry for each member is created and the value is assigned. Special handling is needed for values that are objects: The type-check must utilize the linearization of the allowed object type to allow children as well. After this, another check is prepared to verify only non-abstract `nyan`-objects are used as values. Depending on the member type and the value type, the operator is then checked for validity.

The fifth step performs further sanity checks now that all types and values have been loaded. For each object, it is checked that only patches have inheritance addition annotations (an object may inherit from a patch, then it has no written patch target). For each member of each object, there must be an assignment operator (=) before relative operations can be done. The last check is the verification for only non-abstract objects used in values, which was prepared in step four.

This procedure is executed for each mod and application that wants to load data with the `nyan::Database::load` function. The order has to be determined by the application, otherwise the imports may point to unknown files, if the application does not support loading files of a mod that is requested but was not already loaded.

5.4 Storage

The database storage is divide into two parts. One part stores type, structure and location information of `nyan`-objects, the other part stores inheritance information and values of `nyan`-objects. The first part is only ever extended, and only when loading new objects with the `nyan::Database::load` function, handled by the `nyan::MetaInfo` class. The second part is a `nyan::State` which represents the initial database state with its object inheritance and values.

The `nyan::MetaInfo` class stores information about each object: Its location (for beautiful error messages), information whether it is a patch (i.e. its patch target), its inheritance patching definition, and its initial linearization and children names. Furthermore, information about each member of each object is stored: Where it was defined (again, for error messages), if it was an initial definition, and a reference to its type. The child and linearization information are a template, those will be copied to a `nyan::State` in a `nyan::View` when changes are performed.

A `nyan::State` is the storage of non-permanent information about `nyan`-objects, it maps fqons to `nyan::ObjectState`s, which stores the parents and members. A `nyan::State` can store a reference to a `nyan::State` it is based on, this mechanism is then used for the change history described in Section 5.5.

The lookups from fqon to `ObjectState` and from member id to `Member`, are implemented with a `std::unordered_map`. This is the built-in key-value hash-map store in `C++`. Should their performance not be sufficient, more advanced storage techniques (like dedicated key-value databases, for example the Berkeley Database [35]) could be used in the future. Further adjustments have to be done then, mainly because the implementation relies on the copy and move semantics of `std::unordered_map` when new `ObjectStates` are constructed, which may be hard to reflect.

5.5 Views and Transactions

A `nyan::View` is created by invoking `nyan::Database::new_view`. The view stores a link to its database and its change history, which is then filled with entries through `nyan::Transactions`. A dependent child view is created with `nyan::View::new_child`, the new child view is then registered at its parent so transactions can then be propagated.

In a view, an object can be queried by its `fqon` using the `nyan::View::get` method. The result is a `nyan::Object`, which is a handle designed to be used as permanent reference to a `nyan`-object in the application. It represents the status of this object within a view, and it is used for value queries (see Section 5.6).

A view stores the history of `nyan::States`, which are indexed by a timestamp. Each state stores changed `nyan`-objects with their new members and values. The state history can be queried for a certain point in time. The result is the last collection of object states at either exactly that timestamp or the latest state before the timestamp. The lookup is performed with a `std::map`, which requires logarithmic time due to its tree structure.

To add a new state to this history, a transaction has to be performed with a `nyan::Transaction`, created by `nyan::View::new_transaction`. A point in time can be passed to this function. If none is given, the transaction will be performed for `t = 0`. When created, the transaction creates a new `nyan::State` for each view in the view hierarchy. Updated objects will be stored in this new state. This new state is based on an existing state, which is stored directly before the point in time the transaction is for.

Patches (fetched as `nyan::Object` from some `nyan::View`) have to be added to the transaction with its `nyan::Transaction::add`. For each patch added, the patch target object will be copied from its latest-matching state to each new state. This is the first step for the “shadow paging”-transaction approach (see Section 2.3. The parent linearization of the patch is used to gather all prerequisite patches. Those patches are then applied to the copied target object. This is repeated for every patch that is added until the new state is constructed.

When the transaction is committed, the only thing left to do is to register the newly-built states at their requested time in the history timeline of each view the transaction was for.

The first step is to take existing states at the same point in time as the transaction and merge them with the new states. This must be done as the existing state will be overwritten and thereby all its stored objects will be lost. The merge operation takes the existing state and updates it with all objects from the new state.

The second step is to update cached linearizations and tracked object children. In this process, the new inheritance graph for all children of objects that have changed their parents (due to patches that added new parents) is linearized with the `C3`-algorithm. If an error occurs during `C3`-linearization (no linearization found or cyclic inheritance), the transaction is marked as failed. None of the new states were registered, so the transaction has no effect. If no error occurs, all needed updates for value changes, linearizations and child tracking were gathered, including all necessary sanity checks.

The third and last step for a transaction commit is the permanent storage of the state containing the newly created object states and update caches in each view. The insertion of the new state (which contains all newly patched objects) at the desired timestamp also leads to the deletion of states after it.

5.6 Queries

Queries for values are always performed on a `nyan::Object`, obtained from a view through `nyan::View::get`. The object is a handle that is time-independent. It can be stored in the application logic and it will never become invalid, because `nyan`-objects can never be deleted in the database.

A query has to be done for a point in time of the value history. If no point in time is specified, `t = 0` is used.

Queries are performed with the templated `nyan::Object::get` function, which internally roughly uses the same algorithm as it is simplified in Listing 24. In principle, the linearization of the object is evaluated and the last assigned value of the member in question is copied and then updated for each operation of a child object. This uses the same implementation for value updates as the patching does.

The result can then be casted to the type specified by the template and by that a `C++`-native result is created. This cast can assume the type that was used for the member declaration.

Collections such as the `set` and `orderedset` are provided to `C++` through virtual iterators, which relay the value iteration to `nyan::ValueHolders`. They exist so hashing and comparison have a common interface, and the implementation can be value-specific. For the iteration to work in a generic way for all possible collection types, a lot of template meta programming and virtual inheritance is needed for the iterators. The entry point for the iteration is in `nyan::Container::begin`, which is then implemented for each container type. For the sets, this was unified to a common `nyan::SetBase`, which can create iterators for both the ordered and unordered set, through templates.

Apart from member values, a `nyan::Object` supports providing other information. It can generate a list of its parents at a point in time, check if it has a member at a given time, return if it extends a given fqon for a timestamp and return non-changing information: Its file location or its patch target.

```
def get_value(object, member, time):
    """
    object: fqon for the source object
    member: queried member identification
    time:   point in time the member is queried for
    """
    query_parents, result = [], None
    start_idx = 0
    linearization = get_linearization(object, time)

    for idx, parent_name in enumerate(linearization):
        parent_obj = get_object(parent_name, time)
        query_parents.append(parent_obj)
        if parent_obj.has_member(member) and\
           parent_obj.get_member_operation(member) == ASSIGN:
            result = parent_obj.get_member_value(member).copy()
            start_idx = idx
            break

    for idx in range(start_idx, -1, -1):
        change = query_parents[idx].get_member(member)
        result.apply(change)

    return result
```

Listing 24: Query result calculation

The `nyan::Object`-interface is very minimal and it is suitable for storage in the application directly at the point where this object's data is useful for application operation.

5.7 General Remarks

Much effort was spent to produce helpful error messages. For those, the locations of all `nyan`-objects and their members have to be tracked. In case of problems, the problem origin can be traced and displayed in a message.

Although many operations are written for run-time handling of `nyan` data, heavy use of templates, move semantics and `constexprs` allows many compiler optimizations and avoids data copies at runtime.

All sanity checks are implemented at load-time for `nyan`-objects, the only check during transactions is linearization verification.

The code is documented with comments compatible for automatic documentation generation with Doxygen [20].

6 Evaluation and Discussion

In this chapter, available functionality in the implementation is evaluated by the case study of `openage` [36].

The implementation presented in Chapter 5 directly follows the design from Chapter 4. In this chapter it is evaluated regarding its practical usability for application runtime configuration.

Testing and development of the `nyan` framework was done on a Thinkpad X220t with `Gentoo GNU/Linux 4.13.4-JJ x86_64` and `gcc 7.2.0`. The code is published under the `GNU LGPLv3` or later versions [22].

The first part provides examples that illustrate the functionality of the proposed design and implementation. Several configuration scenarios and a possible way to express them with `nyan` are presented and their relation to the design decisions are explained.

Next, the design and implementation choices are evaluated critically. This is followed by current shortcomings. Some of those problems can be the foundation for further improvements, which are suggested in Section 7.1.

6.1 Functionality

The `nyan` framework allows an application to implement a hybrid approach for the creation of a mod API. It can register scripting functions as well as data records. As `nyan` only allows configuration, the application developers need to decide on an exposed interface and what to do with it. The examples illustrate how `nyan` assists in the creation of configuration interfaces required by several different scenarios in the `openage` use-case. The first example in Section 6.1.1 shows how queries with the `nyan`-API are done. All other examples assume the knowledge of this concept.

6.1.1 Application Integration

The first demonstration is the creation of a simple mod API. The application provides a data schema and performs access to `nyan` based on this.

The following example illustrates how a mod API can be implemented. The application provides a `Mod` `nyan`-object which has a member that stores an ordered set of patches to apply. When a mod pack is created to add new content to the engine or change existing records, it creates a child object from this `Mod`-object and add patches to the set. The `Mod`-object is registered to the application with a mod description file afterwards.

After the application loads its schema in the form of nyan-objects, application code can then access customizations of those nyan-objects and their members through the nyan C++ API.

In Listings 25 and 26, an example implementation of this mechanism is presented.

```
# Engine API definition: engine.nyan

Mod():
    patches : orderedset(Patch)

Tech():
    patches : orderedset(Patch)

Unit():
    hp : int
    can_create : set(Unit) = {}
    can_research : set(Tech) = {}

CFG():
    initial_buildings : set(Unit)
    name : text

StartConfigs():
    # available start game configurations
    available : set(CFG) = {}
```

Listing 25: A simple mod-API example (engine.nyan)

The engine provides nyan-objects (the schema) which must be filled with values when used in the mod. A Unit may be implemented as a controllable object in the game world (a character or a building), a Tech is a technology available for research to improve a unit.

The Unit.hp member is enforced to be set to a value, otherwise the improved Unit object cannot be used in assignments. With this interface definition from the game engine, game content is created (in Listing 26). A Villager is a Unit that can build a TownCenter. In a TownCenter, new villagers can be created and the Loom technology can be researched, which improves the Villager's health points. The initial registration of this is the DefaultConfig, which configures the start situation for a player: It shall start with one TownCenter.

VillagerMod is the mod entry point, which changes the default game start configuration to be the one with a default town center. If this town center was not available by default, the game engine would never know about either Villagers or TownCenters, although the objects would be present in the nyan database.

```
# Data pack: content.nyan

import engine

Villager(engine.Unit):
    hp = 25
    can_create = {TownCenter}

Loom(Tech):
    HPBoost<Villager>():
        hp += 15

    patches = {HPBoost}

TownCenter(engine.Unit):
    hp = 2400
    can_create = {Villager}
    can_research = {Loom}

DefaultConfig(engine.CFG):
    initial_buildings = {TownCenter}
    name = "you'll start with a town center"

VillagerMod(engine.Mod):
    Activate<engine.StartConfigs>():
        available += {DefaultConfig}

    patches = {Activate}
```

Listing 26: Content creation with the given engine API (`content.nyan`)

In order to tell the engine about this `VillagerMod`, independent registration has to be done, for example in a plain text file presented in Listing 27.

```
# modpack.nfo
load: content.nyan
mod: content.VillagerMod
description: Adds villagers and town centers to the game
# could be extended with dependency and version information
```

Listing 27: Auxiliary configuration file to register mods (`modpack.nfo`)

This file is parsed by the game engine so it can know which filename to use and which object in this file describes the mod.

In summary, the load procedure works like this:

1. Load `engine.nyan` into the `nyan` database
2. Read `modpack.nfo`
3. Load `content.nyan` into the `nyan` database
4. Apply the ”mod-activating” patches in `content.DefaultMod`
5. Let the user select one of `engine.StartConfigs.available`
6. Generate a world map and place all units in `CFG.initial_buildings`
7. Display GUI elements for units that can create others and bring them into being
8. Display GUI elements to activate available research (`Loom`, in this case)

If this mod is enabled by the user, this means when a newly created villager is selected, it can build town centers! And the towncenter can research a healthpoint-upgrade for villagers. This behavior does not stem from `nyan`, it has to be implemented in the game engine.

The mod API definitions in `engine.nyan` have to be designed exactly the way the `C++` engine code is then using it. It sets up the data schema so that the `nyan` `C++`-API can then be used to provide the correct information to the application.

The following `C++`-code demonstrates the usage of the `nyan` `C++` API to load and access the `nyan` objects in the `nyan` database.

```
// callback function for reading nyan files via the engine
// we need this so nyan can access into e.g. archives of the engine.
std::string base_path = "/some/game/root";
auto file_fetcher = [base_path] (const std::string &filename) {
    return std::make_shared<File>(base_path + '/' + filename);
};

// initialization of API
auto db = std::make_shared<nyan::Database>();
db->load("engine.nyan", file_fetcher);

// gather all enabled mods and their order (simplified..)
std::vector<Mod> enabled_mods{{"modpack.nfo"}};

// load all enabled mods in order
for (auto &mod : enabled_mods) {
    ModInfo nfo = read_mod_file(mod.nfo_name);
    db->load(nfo.load, file_fetcher);
}

// modification view: this is the changed database state
std::shared_ptr<nyan::View> root = db->new_view();
```

```
// verify that the plugin-activation object is type-correct
nyan::Object mod_obj = root->get(nfo.mod);
if (not mod_obj.extends("engine.Mod", 0)) { error(); }

nyan::OrderedSet mod_patches
    = mod_obj.get<nyan::OrderedSet>("patches", 0);

// activation of userdata (at t=0)
nyan::Transaction mod_activation = root->new_transaction(0);

for (auto &patch : mod_patches.items<nyan::Patch>()) {
    mod_activation.add(patch);
}

if (not mod_activation.commit()) { error("failed transaction"); }

// presentation of userdata (t=0)
for (auto &obj : (root->get("engine.StartConfigs")
                  .get<nyan::Set>("available", 0)
                  .items<nyan::Object>())) {
    present_in_selection(obj);
}

// query the game profile with the GUI
nyan::Object startconfig = get_selected_startconfig(...);

// use result of GUI-selection
printf("generating game map with config %s",
       startconfig.get<nyan::Text>("name", 0));
place_buildings(startconfig.get<nyan::Set>("initial_buildings", 0));

// set up teams and players
auto player0 = std::make_shared<nyan::View>(root);
auto player1 = std::make_shared<nyan::View>(root);



// ====== let's assume the game runs now

// to check if a unit is dead:
engine::Unit engine_unit = ...;
nyan::Object unit_type = engine_unit.get_type();
int max_hp = unit_type.get<nyan::Int>("hp", current_game_time);
float damage = engine_unit.current_damage();
if (damage > max_hp) {
    engine_unit.die();
}
else {
    engine_unit.update_hp_bar(max_hp - damage);
}

// to display what units a selected entity can build:
nyan::Object selected = get_selected_object_type();
```

```
if (selected.extends("engine.Unit", current_game_time)) {
    for (auto &unit : (selected.get<nyan::Set>("can_create",
                                                current_game_time)
                             .items<nyan::Object>())) {
        display_creatable(unit);
    }
}

// technology research:
nyan::Object tech = get_tech_to_research();
std::shared_ptr<nyan::View> &target = target_player();
nyan::Transaction research = target.new_transaction(research_finish_time);
for (auto &patch : tech.get<nyan::Orderedset>(
                       "patches",
                       current_game_time
                   ).items<nyan::Patch>()) {
    research.add(patch);
}

if (not research.commit()) {
    error("failed research transaction");
}
```

Listing 28: Auxiliary configuration file to register mods (`modpack.nfo`)

The application has to initiate transactions for the mod activation. The application is not aware what modifications are done then by mods, this happens solely through `nyan`. When the `nyan`-objects are queried, the `nyan` database returns the customized content.

6.1.2 Unit Hierarchy

In `openage`, several unit types belong to one unit class. Examples for such classes are all horse-mounted units, all arrow-shooting units, all ships, units with swords, etc. Improvements are available to a single unit type (for example for a civilization's unique unit) or a whole class of units.

`nyan` assists in the representation of the hierarchical structure for this unit structure. It can represent the hierarchy with relative changes. Changes in this hierarchy are propagated transitively through value inheritance.

The engine must provide an interface for `Unit` and `RangedUnit` because it has to implement unit movement, damage calculations and projectile ballistics. The game content specification then uses this for the declaration of unit types and their relations. An example for this is in Listing 29, where the `Archer`, `Crossbowman` and `Arbalest` are defined.

A single unit on screen has a unit type, which may point to `Crossbowman` for example. When this unit type is queried, the resulting value is aggregated over the C3-linearization of the inheritance hierarchy (`hp = 30 + 5`).

```
# Engine API definitions:
Unit():
    hp : int
RangedUnit(Unit):
    range : float
    damage : float
    rate : float

# Content declaration:
Archer(RangedUnit):
    hp = 30
    range = 4.0
    damage = 4.0
    rate = 2.03
Crossbowman(Archer):
    hp += 5
    range += 1.0
    damage += 1.0
Arbalest(Crossbowman):
    hp += 5
    damage += 1.0
```

Listing 29: Possible unit type hierarchy definition

When mod wishes to introduce a new technology to research which improves the health points of all archers, this can be achieved with a patch just for `Archer`, because `Crossbowman` will still have 5 more health points than the `Archer`. The patch for such an improvement is in Listing 30. The mod introduces a new button in the town center which, when clicked, improves all archers, crossbowmen and arbalests by 10 health points.

```
TougherArchers(Tech):
    ImproveArcher<Archer>():
        hp += 10
    patches = {ImproveArcher}

VillagerMod(Mod):
    AddResearch<TownCenter>():
        can_research |= {TougherArchers}
    patches = {AddResearch}
```

Listing 30: Adding a new technology by a mod

This is possible because the inheritance hierarchy in `nyan` remains effective, independently of the patch application.

6.1.3 Mod Combination

In the previous two examples the `Loom` and `TougherArchers` technologies were defined. To balance the game, it might be better if loom would give villagers more health points, but the archer improvement has to be weakened. To achieve this, one can adjust already existent mods by introducing another mod. The mod declared in Listing 31 performs those changes so the `Loom` technology will then improve villagers by 20 health points, and the `TougherArchers` will only boost archers by 5 points.

```
GameBalance(Mod):
    BalanceLoom<Loom.HPBoost>():
        hp += 5
    BalanceArchers<TougherArchers.ImproveArcher>:
        hp -= 5
    patches = {BalanceLoom, BalanceArchers}
```

Listing 31: Modding a mod to balance the game

The mechanism used here is that a patch (`BalanceLoom`) modifies another patch (`HPBoost`). The `GameBalance` mod can be changed by another mod, if required.

6.1.4 Creating a Scripting API

`nyan` does provide any possibility to express and execute code, but regular member values can be used to define an entry-point for a fully dynamic scripting API. The names of hook functions to be called are set up through `nyan`. It is impossible to automatically check the validity of code. This can lead to unexpected runtime crashes. Handling for those errors must to be done in the application.

In order for code hooks to be active, the application must load the script file and call the desired function. A simple example is presented in Listing 32. The semantics and signature of the hook functions are prescribed by the application.

These code hooks can then even be modified by other mods: The filename and function name can be redirected to a different script, when desired.

In order to reduce redundancy instead of repeating the source file name, a specialized `Code` object that sets `Code.source` to a fixed value can be created. This object is then further specialized for changing the function name.

```
# Engine API:
Code():
    source : file
    func : text

Ability():
    check_available : Code
    on_activate : Code


# Usage for content creation:
FlyAbility(Ability):
    AvailFunc(Code):
        source = "./fly_mod/entry.py"
        func = "check_available"
    check_available = AvailFunc
    ActivateFunc(Code):
        source = "./fly_mod/action.py"
        func = "fly_handler"
    on_activate = ActivateFunc
```

Listing 32: Simple scripting API hook registration

6.1.5 Schema Extension

This example demonstrates the combination of a more complete content API for a game engine like `openage`.

First, the example assumes the engine now supports in-game resources. Also, several abilities for units like their movement are now available.

The content pack adds wood as a resource and creates villagers and town centers. Villagers can chop wood and bring it to the town center.

Listing 33 defines the application API. Listing 34 uses this API to define example game content. In Listing 35, a mod is declared, which extends the schema by adding a new resource. The Python script from Listing 36 further customizes the newly added `FoodGather` ability.

```
## engine.nyan: Engine feature API with resources
Mod():
    name : text
    patches : orderedset(Patch)

Code():
    source : file
    func : text

Ability():
    Default(Code):
        source = ""
        func = ""
    check_available : Code = Default
    on_activate : Code = Default

Resource():
    name : text
    icon : file

Unit():
    abilities : set(Ability)
    hp : int
    can_create : set(Unit) = {}

DropSite():
    accepted_resources : set(Resource)

ResourceSpot():
    Amount():
        type : Resource
        amount : int
    resources : set(Amount)

Movement(Ability):
    speed : float

LifeAbility(Ability):
    die_animation : file
    death_unit : Unit

DecayAbility(Ability):
    time : float

CollectResource(Ability):
    target : Resource
```

Listing 33: Example engine API with resources (`engine.nyan`)

```
## content.nyan: default content pack
import engine
Wood(engine.Resource):
    name = "Wood"
    icon = "wood.svg"

TownCenter(engine.Unit, engine.DropSite):
    hp = 2400
    accepted_resources = {Wood}
    can_create = {Villager}

Villager(engine.Unit):
    hp = 25
    Move(engine.Movement):
        speed = 15.0
    Life(engine.LifeAbility):
        die_animation = "./assets/aargh.ani"
        death_unit = DeadVillager
    ChopWood(engine.CollectResource):
        target = Wood

    abilities = {Move, ChopWood, Life}

DeadVillager(Villager):
    Decay(engine.DecayAbility):
        time = 40.0
    abilities = {Decay}
```

Listing 34: Example content mod pack with `Wood` resource (`content.nyan`)

```
## food_mod.nyan: adds new resource
Food(engine.Resource):
    name = "Food"
    icon = "assets/food_icon.png"

BerryBush(engine.ResourceSpot):
    BerryFood(engine.ResourceAmount):
        type = Food
        amount = 125
    resources = {BerryFood}

FoodGather(engine.CollectResource):
    target = Food
    Check(engine.Code):
        source = "behavior.py"
        func = "check_food_target"
    check_available = Check

VillagerMeat(engine.ResourceSpot.Amount):
    type = Food
    amount = 60
```

```
FoodMod(engine.Mod):
    TCAdd<content.TownCenter>():
        allowed_resources += {Food}
    Gathering<content.Villager>():
        abilities += {FoodGather}
    Cannibalism<content.DeadVillager>[+engine.ResourceSpot]():
        resources = {VillagerMeat}

    name = "Add the food resource to villagers"
    patches = o{TCAdd, Gathering, Cannibalism}
```

Listing 35: Example mod that adds `Food` (`food_mod.nyan`)

```
## behavior.py: plugin script for the food_mod
def check_food_target(unit, target):
    # only gather from foreign villagers
    # when there is food deprivation
    if target.type == "content.DeadVillager":
        if unit.owner.resources["Food"] < 200:
            if unit.owner != target.owner:
                return True
    elif target.type = "food_mod.BerryBush":
        return True
    return False
```

Listing 36: Script in the `food_mod` mod pack (`behavior.py`)

After the `FoodMod` from Listing 35 is enabled, villagers gain the ability to gather food from villagers that are not from the player. The script function arguments are selected by the game engine, `nyan` just provides the function name.

The engine API exposes its configuration through the objects of `engine.nyan`. Behavior for them is implemented in native code. The `MovementAbility` for example grants units pathfinding and walking capabilities, which is implemented in the engine. The interesting part is `FoodMod.Cannibalism`: It turns dead villagers into resource spots, which provide food. The food gathering ability is further refined using a Python script. The engine scripting API is exposed through `Ability.check_available` and is used in `behavior.py`. The registered function makes use of the schema extension: It uses the newly added `Food` resource for the availability check. Other resources can be added in a similar way.

6.2 Error Checking

All nyan files are error-checked at load time. If an error occurs, the load function fails and an exception is thrown.

An error occurs for example if a type check fails, the inheritance graph cannot be linearized or if unknown names are accessed.

Some notable errors and their messages are presented in Listing 37.

```
NewType():
    test : int
Base():
    setting : NewType
Override(Base):
    setting = NewType

test/test.nyan:6:14: error: this object has members without values: test
    setting = NewType
              ^~~~~~~

ProtoType():
    value : float
RealType(ProtoType):
    value = "text"

test/test.nyan:4:12: error: invalid value for number, expected float
    value = "text"
            ^~~~~~

Right():
    ...
Wrong():
    ...
Demo():
    value : Right = Wrong

test/test.nyan:6:20: error: value (resolved as test.Wrong)
                            does not match type test.Right
    value : Right = Wrong
                    ^~~~~
```

Listing 37: Various type checking errors

When nyan-files were loaded successfully, the correct structure access and conversions to a C++ type is in the responsibility of the application.

In the nyan database implementation, several value types are used that must be casted dynamically in order to be used natively. If the structure is used wrongly (e.g. access to missing objects or members), errors will be raised. If values are queried, but casted wrongly, this may result in undefined behavior (depending on the cast type). These are problems and errors the application must deal with, the nyan-API does not have bearing on this.

6.3 Security

The nyan language is purely used for data representation and does not allow any code execution. This is ensured by validation when data is loaded (see Listing 37). No native C++-objects other than the representation and storage components for the database are created, therefore no unsanitized data records are provided at the nyan-API.

The records are only provided to the application when it queries for them, so it may be possible to trigger bug in the application that embeds the nyan database by clever object creation, combination and registration. A possible bugs is the creation and registration of a nyan-object which in turn is used to create a C++-object in the application. If this object is casted statically without a runtime type check to a wrong type, the object causes undefined behavior. If the created object is casted dynamically, this error would be detected.

More errors and bugs can occur if configuration for code hooks is provided in the application, for example through the API in Section 6.1.4.

If code registered to a hook is untrusted, a proper sandbox is required to prevent the function from performing malicious tasks. This is particularly important for a public mod registry, where users can upload code that would then be executed by the application. A possible solution, except a sandbox, is code signing by the application authors. Unsigned third-party code could still be allowed for mods. For unsigned mods, a warning can be displayed which informs about code execution that may have unforeseen consequences. This has to be taken care of by the application developers.

6.4 Design Considerations

The key features of the nyan framework make it suitable for integration into an application that requires complex configuration scenarios.

The nyan language supports the declarative description of hierarchical and connected data structures through inheritance and cross references. This form of value propagation is directly built into the design and allows modification of many objects through a common parent.

Another form of value modification is the usage of patches, which perform permanent changes to its target. This means the old state of the target nyan-object state will vanish. The modification capabilities of using inheritance and patches is equal, therefore inheritance should be used whenever variants of an object shall coexist. Patches should be used when this coexistence is not needed.

The constraint that nyan-objects can only be used as values if all their members were assigned a value can be used to create an abstract data configuration interface which enforces value settings.

If this schema is used wrongly, for example because members were forgotten or used with the wrong type, the nyan database will issue errors that assist in debugging the problem. If nyan-objects not part of the schema are loaded into the database and are never used as value, they have no effect. Objects loaded for the schema only become important because the application assumes their existence for queries. From this follows that content additions only matter if the application performs queries to members that link to these additions.

The nyan framework allows to create scripting APIs. They can be created if the application interprets member values as filenames and function names. The function signature and passed arguments are prescribed by the application. The application is responsible for proper sandboxing or verification of this code execution.

All changes in the nyan database are contained in distinct views, so the same base state can develop differently, depending on the view. This is useful in game engines to separate for example players and teams. The game start is the same for all, then each team may have improvements depending on the civilization the players selected. The view on the database for each player then may evolve differently during the game, depending on the technologies he researches. This state view separation is directly supported by the nyan database, making it fit for tracking of independent states.

The history of all transactions within a view is tracked on a timeline, this allows the application to store state predictions, which may turn out wrong. If predictions shall be deleted, the nyan database can rollback to any previous point in time of the transaction history.

Queries are performed on a member of a nyan-object in the database by aggregating the values with custom operations in the C3-linearized parent list of the target object. This flattens the data graph and takes all object injections and value changes into account.

The nyan database is embedded into the application through its C++-API. In theory, a pure C for this interface can be created easily to achieve compatibility with most of today's programming languages.

All these features make nyan a potent runtime configuration framework that is suitable for complex applications scenarios like realtime strategy games such as openage.

6.5 Feature Comparison

nyan unifies several strengths of existing approaches. A comparison which lists the notable design properties of nyan is illustrated in Table 6.1. Other common configuration systems, presented in Section 3, do not provide the features of nyan at once.

nyan was created to provide a unified solution that combines several strength of other approaches. Non-developers and unexperienced mod creators can declare modifications in the nyan language. Application developers can easily integrate the nyan database with its C++ API to make use of the features the nyan framework provides.

The nyan language tries to remain at the simplicity level of JSON, but extends

Feature	JSON	YAML	QML	XML	Forge	GRF	Creation	nyan
Key-Value	☑	☑	☑	☑	☐	☑	☑	☑
Object references	☑	☑	☑	☑	☑	☐	☑	☑
Type-safe	☐	☑	☑	☑	☑	☑	☑	☑
Custom types	☐	☑	☑	☑	☑	☐	☑	☑
Data inheritance	☐	☑	☑	☐	☐	☐	☐	☑
Calculations	☐	☐	☑	☐	☑	☐	☐	☑
Enforced schema	☐	☐	☐	☑	☑	☑	☑	☑
Schema extension	☐	☐	☐	☐	☑	☐	☐	☑
Data overlays	☐	☐	☐	☐	☐	☑	☑	☑
Patches	☐	☐	☐	☐	☐	☐	☐	☑
Data history	☐	☐	☐	☐	☐	☐	☐	☑

Table 6.1: Available features for several data description systems

it with types. `JSON` could be used as input language for `nyan` with some effort, but it lacks the language features that value types, object inheritance, modification operators and patches can be expressed directly. It is of course possible, when provided transformation operations and additional information, to convert `JSON` to the `nyan` language.

`YAML` could also be converted into the `nyan` language if enough information like member modification operators are supplied. Duplicate dictionary keys, as allowed in `YAML`, must be forbidden then. The optional element type information can be mapped directly. The `YAML` "merge-keys" feature is a simplified form of the data inheritance mechanism of `nyan`, but it just supports the override of value. Relative modifications are not possible.

Parent-relative calculations like in `QML` are possible in `nyan` with member operators. A full code API like `Forge`, due to its nature, can be extended to provide arbitrary complex new interface functions. The downside is that untrusted code can be executed without a proper sandbox. The scope of application specific modding systems like `GRF` or the Creation Kit are very limited as they are a great choice for their domain, but they are unsuitable for other possible applications.

`nyan` provides data schema verification by enforcing type soundness and known type and member names. The schema can be extended by adding new `nyan`-objects. New records can then be based on the new `nyan`-objects so that code plugins can access them. `nyan`-patches provide a new way of expressing planned changes in a database. They allow the representation of the possible state space without the storage of all resulting changes. None of the current configuration systems provide a configuration change history.

6.6 Limitations

6.6.1 Database Properties

The current design and implementation does not follow all properties for ACID-compliance.

Transactions are performed atomically and are sanity checked to prevent state corruption so database consistency is ensured. This means the first two properties are satisfied.

Isolation for multiple clients is currently not implemented but is possible. The implementation is focused for support for only one client application that cannot access the `nyan` database from multiple threads in parallel. When a transaction is committed, transactions with a timestamp after it will be deleted. For parallel access, this means that the latest transaction commit will win to drop states created by transactions with a later timestamp. With considerable effort, it is possible to add client isolation to the `nyan`-API.

Database durability is missing by design. The initial state is created by the application by loading all required `nyan`-files. Changes in views are also created by the application through transactions. If the application terminates because of any reason, the state is gone. It can be recreated if the application stores the order and timestamps of successful transactions permanently on disk, those can then be restored by recreation of the initial state and replaying the transactions.

Currently, there is no serialization support for permanent storage. This is offloaded to the application, but can be added to the implementation if needed.

6.6.2 Expressions

It is not possible to formulate expressions as member values. Multiple objects can be chained by inheritance to achieve calculation formulas.

If the design allows expression assignment to members, object chains are no longer needed, but updates to the member value will replace the whole expression, which limits the possibilities to combine multiple mods. This can be improved by extending the language for expression patches, which transform a given expression into a different one by declared rules. If those rules can be modified with rules of the same syntax, expression modification-modifications would be possible in patches.

Cross-member references could be achieved with expressions. With this mechanism, the value of one member can be composed by values of other members, like it is possible with `QML` and of course real programming languages.

6.6.3 Update Notifications

Results can be retrieved only by initiative of the application when it performs queries in a view. Patches are also applied through transactions on the application's request. The patch activation can cause many other values to change, because the changes are propagated in the inheritance graph.

There is no way for the application to know which values did change because of the transaction. The application has perform requests to retrieve the updated record values.

All changes could be delivered to the application as notifications through callbacks. Then, when a transaction is committed, a function of the application is invoked for each value that changes due to propagating state updates. With this information, the application can update its internal state accordingly.

It is much easier for `nyan` to determine which values change than letting the application keep track of this and then perform queries to retrieve the results. This would be particularly interesting for event-based applications, which can avoid active queries if the `nyan`-API can deliver changes directly.

In practice, this will not be a problem if the application queries for `nyan`-objects it prescribed through the schema. Only `nyan`-objects reachable through the schema should be relevant for application operation.

7 Conclusion

7.1 Future Work

nyan can be extended with many new features and ideas that provide improvements to its current limitations.

7.1.1 Compiling nyan

Performance improvements in the interface from nyan to the applications can be achieved by converting a given nyan schema, i.e. a set of object definitions, into native code. This allows the application to access data directly in its natively supported type and no string lookups are necessary to find entries.

The conversion process is similar to that of Cython, which is able to convert Python code to C++ code. Compatibility to Python is ensured for generated interface functions, sufficient type annotations can be added in order for Cython to completely drop all of Python's dynamic typing so that C++-only code is emitted [3].

The same approach can be done for compiling the nyan language. A .nyan file could be compiled into a pair of .h and .cpp files, which defines a specialization of nyan::Object which stores all the nyan-object-members as native struct members of native type. This class still has to support accesses over the runtime member map for two reasons. First, when runtime data is loaded (the configuration .nyan files from a mod), all its string-indexed requests must be redirected to the native members of the class. Second, if this "precompiled" nyan-object is used as inheritance parent, new members added at runtime still have to be stored.

If a child nyan-object uses multiple parent objects where two or more of those were precompiled, only one of them can be used in its native form. Otherwise, a combined native class variant is required, which can not be generated if the need for a combined base object is not known beforehand.

One possibility for choosing one of the native parents to be the selected C++-type is some annotation in the inheritance parent list. The annotation is then used as a preference indication for the C++-class to instantiate.

7.1.2 Value Formulas

Currently, only one operation for each member is allowed. This could be extended to include arbitrary expressions. The formulas can be restricted at first, to only permit the reference to the current member once:

```
<member> = <value>|(<member>(<operation> <value>)+)
```

This would have the same effect as chaining multiple objects by inheritance to achieve this formula.

The more advanced variant would even allow combining arbitrary object member values into the formula. For the internal implementation, this would mean many extensions in order to track value changes properly. Objects that not necessarily a parent can then change its value, which would require an update in the calculated value then.

7.1.3 Event Callbacks

Currently, the API is designed for queries from the application, whenever it needs an answer it calls to `nyan`.

A way to extend this is to deliver notifications to the application, when a value changed. The trigger to initially apply a patch is done by the application already, but `nyan` can allow to register hooks for objects and members the application is interested in.

7.1.4 List and Dict Type

The `nyan` language could be extended with a `dict`-type. When a member is created as `dict`, the type has to be specified for the key and the value. Possibly, this would be `dict(keytype, valuetype)`.

The possible operators for this member are in Table 7.1.

Operator	Description
`= {key: value, k: v, ...}`	Assignment
`+= {k: v, ...}`, `|= {..}`	Data insertion/replacement
`-= {k, k, ...}`, `-= {k: v, ...}`	Deletion of keys
`&= {k, k, ...}`, `&= {k: v, ...}`	Keep only keys

Table 7.1: Possible operators for `dict`

A `list` type was not included in the current design as there was no apparent solution for the combination of lists. If two independent mods both add elements to a list, each other can not know if an element is already in the list. In a `set` this is no problem, it can be in there only once, but in a `list` it can be added multiple times.

7.1.5 Nesting Containers

Currently, only non-container data types (e.g. `Object`, `bool`, `text`) are possible for the type a container can contain. This can be extended to arbitrary types. That way, a container can hold other containers, and so on.

The application has to handle those nested structures properly, it makes no difference for the patching and value propagation mechanism inside `nyan`.

The only problem is that a `set` can not be hashed properly if an element inside of it changes its hash. If a set contains a set, the inner set may either not change its hash, or trigger a rehash of the outer set. A possible solution for this is to freeze the set contents, so it becomes a `frozenset`, which can not be changed any more.

The real-world need for such a feature is unclear because `nyan`-objects can contain `set` members and members with references to objects. If `set` nesting is required, an object can be created for each nesting depth, and an object reference is stored in the `set`. This also allows named access to each set, which would not be possible in a nested set.

7.1.6 Documentation Generation

It is possible to automatically generate API documentation for `nyan`-objects that make up a schema. The structure and types can directly be derived from the object declarations. Additional information and documentation can be added as comments, but it is not yet possible to assign this information to specific objects or members.

It would be necessary to extend the parser to not ignore any comment but instead support some kind of mark-up language to allow interface documentation, which can then be exported as `HTML` or LaTeX.

This can also be integrated into Doxygen, which generates a complete cross-reference web page for common programming languages [20].

7.1.7 Set Specialization Operators

A current design limitation is the possible specialization of containers through inheritance. This problem occurs because the current operators on collections (such as `set`s) do allow adding and removing elements, but not both at the same time. This can be improved by allowing formulas, as described in Section 7.1.2. With the current design, it is inconvenient to perform further specialization if an object present in a container is extended with the desire to replace the old object in this container.

This problem arises, for example, when an intermediate specialization of a unit type is declared. Its move-ability is intended to be specialized by other more concrete unit types. They can add their new, improved ability to the set, but the removal of the old entry is not guaranteed, especially if a mod performs some modifications to this set already: The object could already have been replaced, so its presence by a known name can't be guaranteed.

A possible solution is to annotate the container insertions with some keyword to mark it for the desired replacement strategy. Then, when a child object of an object that is already present in the set is inserted in this set, the parent is removed automatically.

To allow any combination of replacement and retaining of collection entries, three operators have to be defined. One to mark a entry in the collection that any child of this object will delete it (for example with `@`). Second, one to annotate the object to be inserted so that any of its parents already in the set are deleted (maybe with +). The third operator (e.g. `!`) prevents the action of desired replacement, so that a previous annotation with `@` is ignored.

```
Entity():
    abilities : set(Ability) = {}

MoveAbility(Ability):
    speed : float = 0

# partial specialization of an entity
Unit(Entity):
    Move(Ability):
        speed = 1.0

    # proposed annotation for auto-replacement:
    abilities += {@Move}

Villager(Unit):
    # specialization of unit movement ability:
    VillagerMove(Unit.Move):
        speed = 10.0

    # this will remove Move and add VillagerMove
    abilities += {VillagerMove}
```

Listing 38: Container specialization ambiguity issues

An example for such a situation is in Listing 38. If the proposed new set insertion operators are missing, the `Villager` will have `Move` and `VillagerMove` in its ability `set`, which then creates possible ambiguities for the game engine interpretation of both abilities.

It is possible to avoid such a situation by changing the data model to inheritance, then those three operators are not required so solve the problem, as illustrated in Listing 39. The downside is that with inheritance it's not permitted to remove the `Movable` as parent, which is allowed in a `set`.

```
Entity():
    ...
Movable():
    movespeed : float = ...
Unit(Entity, Movable):
    ...
Villager(Unit):
    movespeed = 10
```

Listing 39: Avoiding container specialization problem through inheritance

7.1.8 Serialization

Serialization will be useful for recreation of the view state. When nyan can export a view, all transactions and the related timestamps can be stored permanently on disk. When the initial database state is recreated, those transactions can be restored and the view state will be the same as it was when the state was dumped.

This approach can be extended to be robust against missing nyan-objects. The assumption that the initial database state must be the same as it was for the view serialization is not necessary: If new objects were added (e.g. by additional mods), the view history can still be restored. If objects vanished, for example when a mod was uninstalled, it is possible to restore unaffected changes, because patches for objects no longer known can simply be skipped. If unknown objects are used when patching values, either the whole patch or just this member have to be skipped. That way, the view serialization can be restored in the case the base state is different now. The consequences depend on the missing objects of course, and a warning should be issued by the application that objects have vanished.

7.1.9 Python Interface

The nyan-C++-API could be accompanied by a Python API, which allows the same operations and requests. openage is scripted in Python and nyan needs some form of Python interactions if mods need to access their custom data structures that they added to the schema.

The tool of choice for this job would be Cython [3], it allows to create functions that can be called from Python or C++ and redirect them to Python or C++. For nyan, the C++ interface would be wrapped and provided as pure Python functions, which can then be called from a mod script file. The same database has to be reachable from C++ and Python because a mod will not directly use nyan, but will instead be given a handle from the application through the application API.

7.2 Summary

This thesis introduced *nyan*, a framework for run-time configuration of applications that use hierarchical and connected data structures, primarily aimed for game engines as their modding interface.

First, the design of the input and schema language was presented, which is written in a declarative Python-like syntax with enforced indentation. It supports several data types and allows to declare `nyan`-objects, their data values and relations to other `nyan`-objects. Patches were introduced as a special form of a `nyan`-object which can change the values of a target object when requested. This request is performed through transactions, which perform atomic application of one or multiple patches. Transactions do not change the default database state, instead, they commit their changes to a database view. A view is a change history holder, created in order to support multiple parallel state requests. This is useful to represent the changes necessary for one team in a game, as well as for their individual players. To support this, views can be organized in hierarchies so transaction changes with patches can propagate to dependent views. Transactions can be performed at a custom point in time, which may be in the future, or past. The interpretation of time solely depends on the application, but in `nyan` this can be used to perform state predictions that may turn out wrong. These are useful for game engines that work event-based, where the game state is predicted into the future, and configuration changes (for example the current speed of units) have to be committed to the `nyan` state. `nyan` tries to provide useful error messages for any problem, especially if any type problems occur. The type constraints are defined by the declaration of `nyan`-objects themselves, which conveniently allows schema declarations and extensions through the same syntax. Type errors are detected at the load-time of data already to prevent crashes when the application has been running for some time and then activates the problematic entry.

Second, implementation details were presented which bring the design ideas to life in `C++`-code, which is released as free open-source software. `nyan` is designed to be embedded in an application as dynamic library, which provides all necessary features to use it as configuration and customization interface. The implementation consists of a parser, which reads in provided data to create an initial database state. During this, many sanity checks are performed and caches are prepared, until all values and the schema are verified so that no type errors remain. Views of this initial state provide distinct handles for `nyan`-objects, intended to be stored at the locations where values will be retrieved by queries. Value results are calculated by graph traversal over the linearized parent list of the target object. The result can be casted to a `C++`-native type through template meta-programming.

The next chapter presents examples on how the design and implementation can be used in practice to create a configuration interface. Several use-cases for language features of `nyan` are illustrated. If any of the type constraints are not satisfied, the database will emit error messages for debugging. There is no possibility to execute code in the `nyan` language. Untrusted functions may only be called if the application permits their registration through a exposed scripting interfaces. The `nyan` framework provides a type-safe key-value database that can record transaction history and allows schema extensions that do not invalidate existing data records. The records can represent directed acyclic graphs which are linearized by the C3 algorithm to perform calculations and value propagations in queries. To date, other configuration systems do not provide all the designed features at once.

7.3 Conclusion

The `nyan` framework combines several features not present in current configuration systems. The `nyan` language is used to declare `nyan`-objects as data records. They allow calculations and specialization through inheritance from multiple parent `nyan`-objects. After sanity checks, `nyan`-objects are stored in the `nyan` database, which is embedded in an application. The data schema is specified by the application through loading `nyan`-objects, so it can then access records by known names and types. By adding new `nyan`-objects, which declare new known names and types, the schema can be extended at runtime. Queries over the `nyan`-API to the `nyan` database are used to retrieve combined results from the data graph.

Notably, `nyan` uses patches to store possible changes to data records as a data record and thus allows secure and user-friendly handling. Patches are `nyan`-objects, so they can be modified as well. Multiple patches can perform different modifications on the same `nyan`-object, hence combinations of patches are possible. Recording the history of all changes enables predictions and rollbacks.

Systems like `JSON` or `YAML` do not provide such features. Application specific configuration systems like `NewGRF` or `Creation Kit` are not suitable to be used outside of their domain. Programming APIs like `Minecraft Forge` are very extensible, but allow execution of untrusted code without a sandbox. `nyan` does not allow code execution, but it can be used to register function hooks. `nyan` is a general purpose configuration framework, suitable for configuring complex scenarios like the behavior of real-time strategy games.

Appendix

Glossary

API **application programming interface**.
Specification of the communication interface between two programs . 1–3, 5, 6, 9, 13–15, 17, 18, 24, 29, 30, 37, 41, 47–50, 54–56, 58–64, 66, 67, 69, 71, 76

AST **abstract syntax tree**.
representation of the syntactic structure of a `nyan` file . 40, 41

fqon **fully qualified object name**.
Unique identifier for a `nyan`-object . 19, 33, 34, 37, 41–43, 45, 76

GUI **graphical user interface**.
An interactive software designed for mouse and keyboard control. Contains interface elements like buttons, text areas, menus and graphics . 11, 15, 19, 50

mod A ”modification” for an existing piece of work. 1, 2, 9, 13–15, 19, 24, 27, 29, 32, 43, 47–50, 52–55, 58, 60–63, 66, 67, 69, 70, 73, 76

mod pack A bundle of components that make up a mod with all its required information and assets. 15, 19, 33, 34, 47, 57, 58, 76

UI **user interface**.
part of a software project that interacts with the user graphically . 13

VM **virtual machine**.
”an efficient, isolated duplicate of a real computer machine” [53] . 11

List of Figures

List of Tables

List of Listings

Bibliography

[1] OpenTTD. https://www.openttd.org/. Accessed: 2017-09-23.

[2] Kim Barrett, Bob Cassels, Paul Haahr, David A Moon, Keith Playford, and P Tucker Withington. A monotonic superclass linearization for dylan. In *ACM SIGPLAN Notices*, volume 31, pages 69–82. ACM, 1996.

[3] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[4] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML™) version 1.2. *http://yaml.org, Tech. Rep*, 2009.

[5] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18 (1):46–53, 1998.

[6] Alex Blewitt. Minecraft modding with Forge. https://www.infoq.com/articles/minecraft-forge. Accessed: 2017-09-16.

[7] Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507. ACM, 2006.

[8] Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, 1992.

[9] Timothy William Bray. The JavaScript object notation (JSON) data interchange format. *RFC7159*, 2014.

[10] Peter John Brown. Writing interactive compilers and interpreters. *Wiley Series in Computing, Chichester: Wiley*, 1979.

[11] Luca Cardelli. A semantics of multiple inheritance. *Information and computation*, 76(2-3):138–164, 1988.

[12] Jose Felix Costa, Amilcar Sernadas, and Cristina Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31(1):5–26, 1994.

[13] Douglas Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML*, volume 2006, 2006.

[14] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[15] Defining Object Types through QML Documents. https://doc.qt.io/qt-5/qtqml-documents-definetypes.html. Accessed: 2017-09-16.

[16] DFHack Dwarf Fortress memory access library. https://dfhack.readthedocs.io/. Accessed: 2017-09-16.

[17] Free Software Foundation. What is free software? https://www.gnu.org/philosophy/free-sw.html. Accessed: 2017-09-16.

[18] Francis Galiegue, Kris Zyp, et al. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, page 32, 2013.

[19] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[20] Dimitri Van Heesch. Doxygen: Source code documentation generator tool. http://www.doxygen.org, 2008.

[21] Michi Henning. API design matters. *Queue*, 5(4):24–36, 2007.

[22] Jonas Jelten. nyan database code repository. https://github.com/SFTtech/nyan/. Accessed: 2017-09-25.

[23] Ralph E. Johnson and Vincent Russo. *Reusing object-oriented designs*. Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[24] Won Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on knowledge and Data Engineering*, 2(3):327–341, 1990.

[25] John Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, Inc., 2009.

[26] List of game engines. https://en.wikipedia.org/wiki/List_of_game_engines. Accessed: 2017-09-13.

[27] John W. Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[28] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[29] Minecraft Forge API. https://mcforge.readthedocs.io/. Accessed: 2017-09-16.

[30] Minecraft Sponge API. https://www.spongepowered.org/. Accessed: 2017-09-16.

[31] Modder builds his own custom iPhone 7 that restores the headphone jack. https://www.theverge.com/circuitbreaker/2017/9/7/16267418. Accessed: 2017-09-15.

[32] Kyle Andrew Moody. *Modders: changing the game through user-generated content and online communities*. The University of Iowa, 2014.

[33] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[34] NewGRF specification. https://newgrf-specs.tt-wiki.net/wiki/Main_Page. Accessed: 2017-09-19.

[35] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[36] openage - free age of empires engine. https://github.com/SFTtech/openage. Accessed: 2017-09-02.

[37] OpenTTD New Graphics Resource File. https://wiki.openttd.org/NewGRF. Accessed: 2017-09-19.

[38] OpenTTD NewGRF Meta Language. https://newgrf-specs.tt-wiki.net/wiki/NML:Main. Accessed: 2017-09-19.

[39] Helen Pearson. Genetics: what is a gene? *Nature*, 441(7092):398–401, 2006.

[40] PEP 20 – The Zen of Python. https://www.python.org/dev/peps/pep-0020/. Accessed: 2017-09-20.

[41] Markus Persson and Jens Bergensten. Minecraft. *Computer software. Stockholm, Sweden: Mojang AB. Retrieved from http://minecraft.net*, 2011.

[42] Python lexical analysis. https://docs.python.org/3/reference/lexical_analysis.html. Accessed: 2017-09-20.

[43] Python programming language. https://python.org/. Accessed: 2017-09-20.

[44] Python standard types. https://docs.python.org/3/library/stdtypes.html. Accessed: 2017-09-20.

[45] Pythonic code style. http://docs.python-guide.org/en/latest/writing/style/. Accessed: 2017-09-20.

[46] QML declarative interface language. https://doc.qt.io/qt-5/qmlapplications.html. Accessed: 2017-09-16.

[47] Qt Development Framework. https://www.qt.io/. Accessed: 2017-09-16.

[48] Daniel J Rosenkrantz and Richard Edwin Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, 1970.

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

[50] Skyrim Creation Kit. `https://www.creationkit.com/index.php?title=Main_Page`. Accessed: 2017-09-19.

[51] Skyrim Creation Kit Tutorials. `https://www.creationkit.com/index.php?title=Category:Tutorials`. Accessed: 2017-09-19.

[52] Skyrim Papyrus scripting. `https://www.creationkit.com/index.php?title=Category:Papyrus`. Accessed: 2017-09-19.

[53] James E Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[54] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *ACM Sigplan Notices*, volume 21, pages 38–45. ACM, 1986.

[55] Bethesda Softworks. The Elder Scrolls V: Skyrim. *Computer game*, 2011.

[56] Jimmy Soni and Rob Goodman. *A Mind at Play: How Claude Shannon Invented the Information Age*. Simon & Schuster, 2017.

[57] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[58] Tes5Mod File Format. `http://en.uesp.net/wiki/Tes5Mod:Mod_File_Format`. Accessed: 2017-09-19.

[59] Understanding the Creation Engine Data Format. `https://www.creationkit.com/index.php?title=Category:Getting_Started`. Accessed: 2017-09-19.

[60] Johannes Walcher. Event-driven game engine in realtime stategy games. Bachelor thesis, Technische Universität München, Oktober 2017.

[61] Martin P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.