Bringing Database Management Systems and Video Game Engines Together

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Daniel O'Grady
aus Mosbach

Tübingen
2021

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	25.06.2021
Dekan:	Prof. Dr. Thilo Stehle
1. Berichterstatter:	Prof. Dr. Torsten Grust
2. Berichterstatter:	Prof. Dr.-Ing. Hendrik P. A. Lensch

Declaration of Authorship

I, Daniel O'GRADY, declare that this thesis titled, "Bringing Database Management Systems and Video Game Engines Together" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TÜBINGEN UNIVERSITY

Abstract

Faculty of Science

Department of Computer Science

Doctor of Science

Bringing Database Management Systems and Video Game Engines Together

by Daniel O'GRADY

As video games gained more popularity through the years and became increasingly more complex, the share of operations on large amounts of data within games hase risen as well. This trend shifts the focus from mere implementation of functionality to the challenge of maintaining, browsing, and processing large amounts of data. While video game engines are gradually rediscovering concepts that are well-known in the relational world of databases by implementing them in an imperative style, it stands to reason to meet midway and instead implement data-heavy operations on the side of the *database management system (DBMS)* for increased data locality. This thesis explores what databases can bring to the table beside their already powerful capabilities of querying data when involving databases beyond their role of simple data storages. For that purpose, typical components of video game engines, that one would usually expect to find in the imperative parts of a video game, are explored and evaluated in terms of their practicality and usability when implemented in *SQL*. Existing intersections between database management systems and video games engines are pointed out in the introduction. The following chapter covers AI in video games on the example of *Monte Carlo tree search (MCTS)* and *deterministic finite automatons (DFAs)*, after which the generation of playing fields using either rule sets or predefined building blocks is covered. As third and final component, path finding, specifically $A*$, within the database is explored, with an optional expansion into the temporal dimension of the search space. Each chapter features a section in which the suitability of the examined component in combination with a DBMS is laid out, describing if the component is either suited for online usage, as in the case of pathfinding and MCTS or DFAs respectively, or should mainly be used offline to make use of the storage capabilities of DBMSs, as is the case for terrain generation. The thesis is wrapped up with a general evaluation of the marriage between DBMSs and video game engines, finding that while not revolutionising the gaming industry overnight, DBMSs can enhance video game development in the long run.

Acknowledgements

First and foremost, I want to thank my supervisor Torsten Grust for enabling and encouraging me to pursue research in an unusual field I love and care about, and for offering guidance on day-to-day obstacles one encounters in academia. I would also like to extend my thanks to my colleagues at the chair for database management systems in Tübingen. They provided me with many insightful talks and technical discussions which in several cases were the initial spark for the next step in my research. Finally, I am deeply grateful to my family and friends, both local and remote, for continuing encouragement, understanding, and patience throughout my journey.

Contents

Chapter 1

Introduction

> "This is a fairly star-gazing view of how to make games: the game is just one giant database, running dynamic queries all the time."
>
> Adam Martin

Video games and *relational database management systems (RDBMSs)* are like distant cousins that only occasionally meet at family gatherings. They both roughly stem from the 1960s and 1970s and have since experienced continuous development and commercialisation. And even though developing a video game and using a RDBMS often means tackling many similar problems, they are rarely used together – or at least not to the extent they could be. This mindset stood to reason during the early days of the two domains. While even the earliest RDBMS were subject to scientific research [19], video games were little more than diversions back then and their complexity was extremely limited by the technology of their time, devoid of considerable amounts of data that had to be handled. Take, for example, *Pong*, which is depicted in Figure 1.1. The two opposing players move their paddle up and down the screen to deflect a ball, much like in tennis. Each time a player misses a ball, their opponent is awarded a point. The game neither features complex logic, nor does it comprise of many game elements. In fact, the visuals of a game of Pong are so simple, that Figure 1.1 is not even an actual screenshot from the game, but a reconstruction built in LaTeX.

The complexity of video games has increased considerably to the current day, as not only have they become more sophisticated in terms of rules and elements, but also the sheer amount of objects that are being shown, computed, and *put into relation* with each other during the game has grown significantly. Video games are no longer restricted to short rounds that only last a few minutes, but can instead go on indefinitely, taking place in vast, persistent worlds, played by hundreds of players simultaneously. This shift opens up a bag of topics that need to be addressed: what does it mean when two players want to manipulate the same part of the game world at the same time? Can we efficiently find a subset of game elements that are relevant for a singular calculation, such as the objects that are currently visible to one player? When we need to do these operations over many game objects at once, how do we avoid swapping parts of the huge game world in and out of the main memory? *Database management systems (DBMSs)* offer solutions to many of these problems natively: it is the bread and butter of a RDBMS to handle large amounts of data with modest amounts of main memory and filter datasets based on arbitrary criteria, sped up by a variety of readily available index structures. Every RDBMS that offers the *atomicity, consistency, isolation, durability (ACID)* properties will solve the issue of miraculous duplication when two players try to grab an item at the same time (a well-known problem in the gaming community, called "duping", while the database community refers to it as "lost update"). Despite the relative similarities between RDBMS and video games, it is not uncommon for business logic of any kind (including video games')

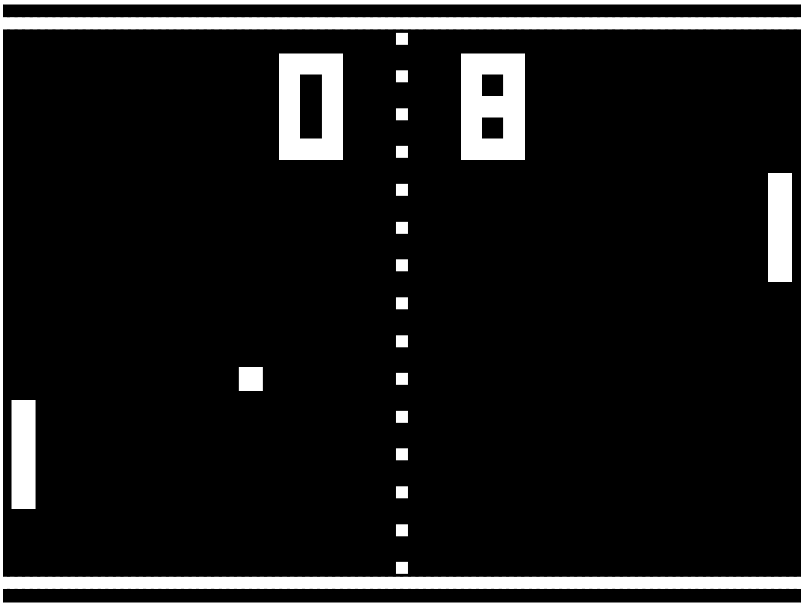
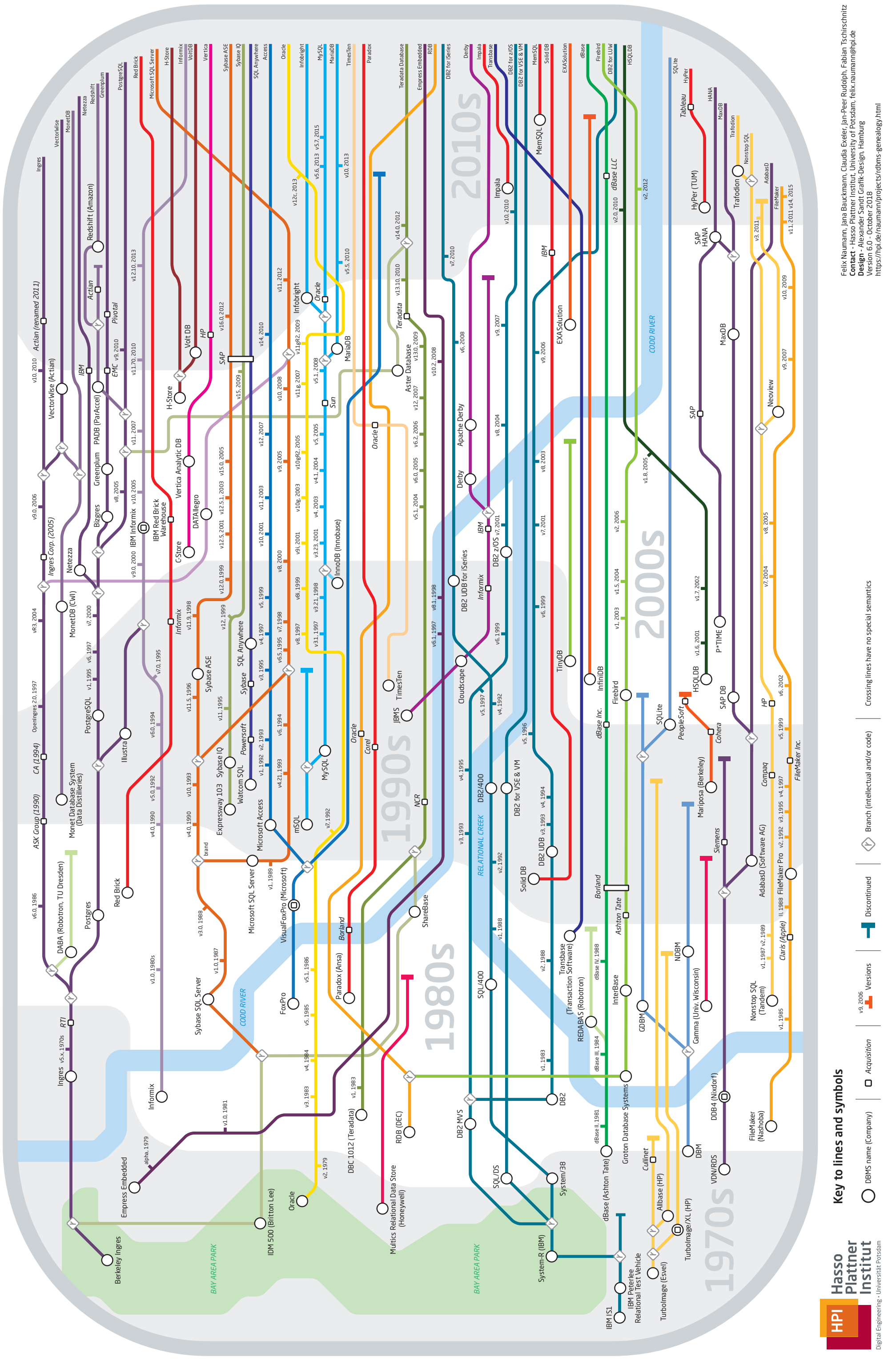


FIGURE 1.1: Pong, one of the earliest video games. The gameplay elements consist of the two paddles to the left and right, the ball (currently in the left half of the screen), score tracking at the top, and the boundaries and centre line of the playing field.

to read all data from the database, compute a subset of the data or transformation thereof in an imperative programming language, and then write it back periodically. Complex calculations of large amounts of data can be done much more efficiently if the *operations are pushed to the data* instead. Game developers may of course implement mechanisms that support the handling of their data – like indices – themselves. But DBMSs have decades of development and optimisation under their belt, as can be seen in Figure 1.2.

This thesis explores a stronger incorporation of RDBMSs in the process of video game development, namely establishing applications of RDBMSs beyond being dumbed down persistent storages, as they are currently being used. Said applicability is demonstrated straightforwardly by providing actual *SQL* code to solve tasks one would be presented with when implementing a modern video game. This thesis deliberately refrains from introducing a *Domain Specific Language (DSL)* to achieve the outlined tasks and uses only plain SQL without utilising any imperative extensions like *PostgreSQL*'s *PL/pgSQL*. We also refrain from utilising an *object-relational database management system (ORDBMS)* as a shortcut from the relational to the object oriented world in which games often live in, so the terms RDBMS and DBMS are used interchangeably throughout this document. While we try to present the code fragments in their entirety if possible, slight abstractions or simplifications are applied if it increases the readability without taking away from the completeness of the queries. This document is therefore code-heavy by design. To ease the reader into the examination of the code fragments, longer listings are broken into smaller parts with accompanying textual explanation. In that case, the line number of the listing is continued among all parts, and continuation is also implied by "… " in the last line of a continued listing.
Since video games are, and historically have been, largely a commercial domain where thorough scientific write-up often takes a secondary role, a lot of the information on internals of video games are propagated in sources that could be considered unquotable in scientific work, such as blogs of game developers, conference talks, personal correspondence with developers, or inspection of open source code video games. Usage of such sources will largely be restricted to cases when no more suitable sources are available.



Felix Naumann, Jana Bauckmann, Claudia Exeler, Jan-Peer Rudolph, Fabian Tschirschnitz
Contact - Hasso Plattner Institut, University of Potsdam, felix.naumann@hpi.de
Design - Alexander Sandt Grafik-Design, Hamburg
Version 6.0 - October 2018
https://hpi.de/naumann/projects/rdbms-genealogy.html

FIGURE 1.2: Genealogy of RDBMS over the years. This graphic was created by the *Hasso-Plattner-Institut für Digital Engineering gGmbH*.

1.1 Meet the Spouses

Since their humble beginnings in the 1950s [47, Chapter 2] video games have moved beyond being a pastime for a few interested enthusiasts. The *Entertainment Software Association* estimates the revenue generated by the video game industry in 2019 at around $ 43 billion [7] and found that over 65 thousand employees worked in the sector by 2018 [6]. *Serious gaming* as a way of ludic transfer of knowledge has found its way into education [46] and health care [100, 29], and, since the advent of modern mobile phones, video games can be accessed on the go even by casual gamers. Databases are not an entirely foreign concept to video game developers, as they offer a way to persistently store game states between sessions, but the two realms of data storage and computation are still largely separated to this day. The following section sheds some light on techniques in video game development that already suggest a certain proximity between video games and RDBMS, and gives an overview over vocabulary that is used throughout this document.

1.1.1 Architecture of Video Games

Early video games were often developed and played on dedicated hardware, like *arcade cabinets*, that had been tailored to one specific game [47, Chapter 3]. Developing a new video game could therefore mean for the developer to go back to the drawing board (literally) and design circuitry and software from ground up. These days, video games are run on general purpose processors, often implemented in widely spread high-level programming languages, enabling developers to recycle code snippets or whole modules from other games. Taking this one step further led to the development of *video game engines*, entire frameworks[1] to help developers create video games by encapsulating recurring tasks. Video games can be classified by their *genre*, which spans an extensive list of dozens of genres and sub-genres, ranging from sports, to puzzles, over real-time strategy, to rhythm games. This spawned a multitude of game engines that either specialise in one specific genre, like how the *OpenRA*-engine [53] is meant to support developers in creating *real time strategy* games, or they try to accommodate as many different genres as possible, like *Unity* [93] does. Depending on what the engine offers, components include, but are not limited to:

Physics engine to simulate physical effects, like gravity, velocity, collisions, particles, and rigid body dynamics.

Rendering engine which draws the visible part of the gaming world (*frustum*) to the screen. This may for example include ignoring objects that lie completely without the field of vision of the player (*culling*), faithfully rendering objects that partially overlap (*occlusion*), scaling objects based on their distance or rendering more distant objects in lesser detail (*level-of-detail*), and shader management.

Audio system which plays background music and ambient sounds, possibly fading them in and out based on the distance between the source and the player, or in a surround fashion to reinforce the impression of a three-dimensional world.

Input controls to receive input from the player through several devices, such as keyboard and mouse, controllers, joysticks, or steering wheels and pedals.

Gregory discusses various common components of engines, as well as their dependence on a certain genre in depth in [36, Chapter 1].

[1] The line between the definitions of “framework” and “engine” is blurry, but the terms are used interchangeably throughout this document.

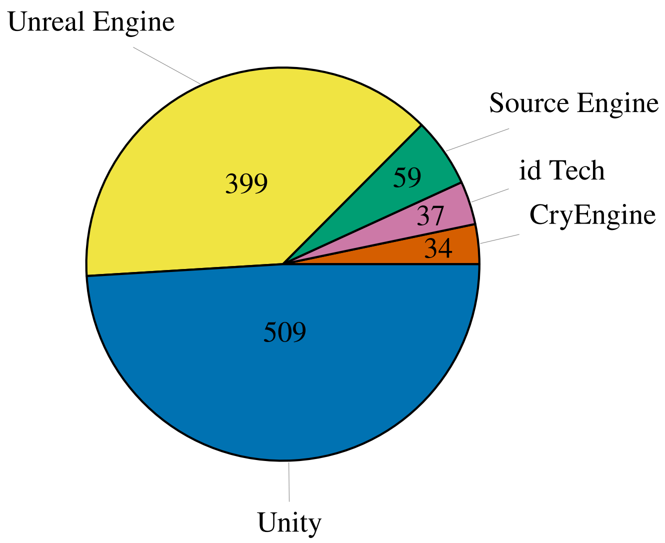


FIGURE 1.3: Number of games published on Steam per used video game engine (data retrieved on 5 February 2020). Note that only games for which an engine could conclusively be determined are included in this graph, limited to the five most commonly used engines. Also, different versions of the same engine are subsumed in the same share.

A common denominator across most engines is the concept of a *tick*, also referred to as *game loop*, which is a recurring event that gives components of the game an opportunity to update their state. The frequency at which a component is updated may vary from multiple times per second to only occasionally. For example, to maintain the illusion of fluid motion, games usually render between 30 and 60 *frames per second (FPS)*, but the physics may actually be updated multiple times between the rendering of two frames. In simple engines a developer is given little more than this update-render-loop, driven by the tick. They can then hook into this loop to implement their game logic in a DSL the framework provides, or the language the engine itself is written in.

1.1.2 Mind the Gap

Video games have for a long time been implemented in languages adhering to the *object oriented programming (OOP)* paradigm. Figure 1.3 gives an overview over the usage of the top five most heavily used game engines for games that have been published on the digital distribution service *Steam*. Most of these engines, with the exception of *idTech*, feature OOP-languages:

- *Cry Engine* [21]: C#, C++, LUA;
- idTech[2] [26, 71, 70, 72, 25]: Action Code Script, QuakeC;
- *Source Engine*: C++;
- *Unreal Engine* [94]: UnrealScript (proprietary language, similar to Java) in versions 1-3, C++ from version 4 onward [95];
- Unity [93]: C#, Boo, UnityScript (proprietary language, syntactically related to JavaScript), starting with v2018.2, C# is the only supported language [92].

[2] Also known as “Doom-Engine” (v1) and “Quake-Engine” (v2-3).

This factor seemingly moves video games and DBMSs even further apart, due to the *impedance mismatch* [20] (which was promptly addressed in various ways, as Carey and DeWitt outline in [14]) between these two worlds. The reasons for OOP being so prevalent are not far to seek, as it lends itself well to how game worlds are conceived. Entities usually appear in higher quantity than just one (think: during an adventure, the player encounters *many* enemies, guarding *many* doors, behind which *many* treasures are hidden), which can often be categorised by a list of attributes, decently fitting the concepts of structs and classes[3] of OOP. The following C#-snippet defines a `struct` called `Position2D` to be a composite of two integers `x` and `y`, representing a coordinate in a two-dimensional space.

```
public struct Position2D
{
    public int x;
    public int y;
};
...
```

OOP commonly takes this one step further and groups functionality with the data, demonstrated in the following fragment, where the functions[4] `Move`, `Damage`, and `IsAlive` are part of the class named `Enemy`.

```
public class Enemy
{
    public Position2D position;
    private int health;

    public void Move(int xDelta, int yDelta)
    {
        this.position.x += xDelta;
        this.position.y += yDelta;
    }

    public void Damage(int by)
    {
        this.health -= by;
    }

    public boolean IsAlive()
    {
        return this.health > 0;
    }

    public Enemy()
    {
        this.position = new Position2D();
    }
}
...
```

So all objects of type `Enemy` feature a common interface, enabling iteration over multiple instances:

```
public static void ApplyGravity(Enemy[] enemies)
{
    for (int i = 0; i < enemies.Length; i++)
    {
        enemies[i].position.y -= 1;
    }
}
```

Storing instances that way, where one element of a list makes up an entire entity with all its attributes (even if they may not be required) is also referred to as *array of structs (AOS)*. This style is being challenged in recent development, which moves video games closer to the realm of DBMS and is therefore discussed briefly in the following subsections.

[3] The semantics of the terms “struct” and “class” are not used consistently across programming languages, but here a `struct` denotes a conglomerate of named values, while a `class` also encompasses functionality.

[4] Functions in the context of a class are also commonly referred to as “methods”.

1.1.3 Data-Oriented Programming

During the 2010s, several game developers, including Mike Acton (*Insomniac Games*, *UnityTechnologies*) [1], Tony Albrecht (*Overbyte*, *Riot*) [2], and Jonathan Blow [10], have spoken out against AOS style in favour of a more *data-oriented* way of thinking about programs. In particular, they propose the usage of *structs of arrays (SOA)*. In SOAs, instead of grouping attributes into one object, all attributes are grouped separately. This makes sense as it is rare to access all attributes of an object at once. Instead, it is much more common for many similar objects to be transformed with regard to only a few of their attributes at a time, as is the case in the `for`-loop in `ApplyGravity` from the snippet above; each `Enemy` has their `position`, or more specifically the `y`-component of their `position`, updated. This circumstance was not lost on game developers, which in one instance led to a SOA-oriented design in the programming language *JAI*, which is being developed by Jonathan Blow at the time of writing [10]. Applied to the above example, the usage of SOAs over AOSs in a JAI-like fashion could look like the following snippet, where the instances of the innermost attribute `y` make up the arrays, whereas the instances of `Enemy` form an SOA, indicated by the keyword `SOA`.

```
public static void ApplyGravity(Enemy SOA enemies)
{
    for (int i = 0; i < enemies.positions.y.Length; i++)
    {
        enemies.positions.y[i] -= 1;
    }
}
```

While the reasoning behind this change is mainly motivated by improved caching behaviour and memory-access, or to facilitate data parallelism [9] it ties in with how *column-based storages* in some DBMSs like MonetDB [42] have handled data for a long time now, while also coexisting with the row-based approach the lends itself to the AOS style [55]. The memory layout of the two approaches is visualised in Figure 1.5.

1.1.4 Entity Component System

Game developers have come to realise that deeply nested inheritance can be both confusing and restrictive when trying to conceive a general framework for video games. They have therefore rediscovered a certain fondness for how information is laid out in RDBMS. The principle of *composition over inheritance* proposes to fragment complex objects into *components*, each responsible to attach a small amount of functionality to the composite object. The former complex object becomes a uniquely identifiable *entity*, that, in itself, does not contain sophisticated functionality [50]. This makes it easy to attach and detach additional functionality to and from entities even during runtime, giving the developer much more control over how their objects are structured. The running example from above would then be implemented as:

```
public class Enemy
{
    public readonly int id;

    public Enemy()
    {
        this.id = GenerateUniqueId();
    }
}

public class HealthComponent
{
    public readonly int entityId;
    private int health;

    public HealthComponent(int entityId)
    {
        this.entityId = entityId;
    }

    public void Damage(int by)
    {
        this.health -= by;
    }

    public boolean IsAlive()
    {
        return this.health > 0;
    }
}

public class PositionComponent
{
    public readonly int entityId;
    private Position2D position;

    public PositionComponent(int entityId)
    {
        this.entityId = entityId;
    }

    public void Move(int xDelta, int yDelta)
    {
        this.position.x += xDelta;
        this.position.y += yDelta;
    }

}
```

Where `GenerateUniqueId` is a function that generates identifiers that are unique among instances of the class, like a *primary key*. Note how this is a lot like how tables in DBMS are put into relation with each other by using foreign keys. Each class can now safely be thought of as a table in which instances of the corresponding class are stored as a single row. This also facilitates the use of SOAs, as selecting all rows of a table or a subset thereof can be done independently from other components. For example, selecting and manipulating the `PositionComponent` of all enemies can be done without loading their `HealthComponent`. Breaking up the nested structure of objects to access specific parts thereof more easily has been explored by the database community before in the guise of *flattening*, which selectively accesses nested attributes by exploiting relational layout in second normal form [91].

1.1.5 Data-Driven Games

Code traditionally not only defined the functionality and rules of games, but also the *content* thereof. Content can refer to several things in this context: *assets*, like graphics and audio, obviously qualify as such. But also the layout of the virtual world, tasks for the player to solve, and ultimately even parameters, like magnitude of gravitational pull on virtual objects, can serve as content. Most, if not all of these things, used to be baked into the code of the game itself, requiring recompilation whenever the content changed. Not only can this be tedious in general, but also unwieldy for uninitiated contributors. Disney's *Epic Mickey 2: The Power of Two*, for example, was worked on by over 700 people [44]. Of course, not the entire team consisted of programmers that would be able to shape their ideas into code, but probably also

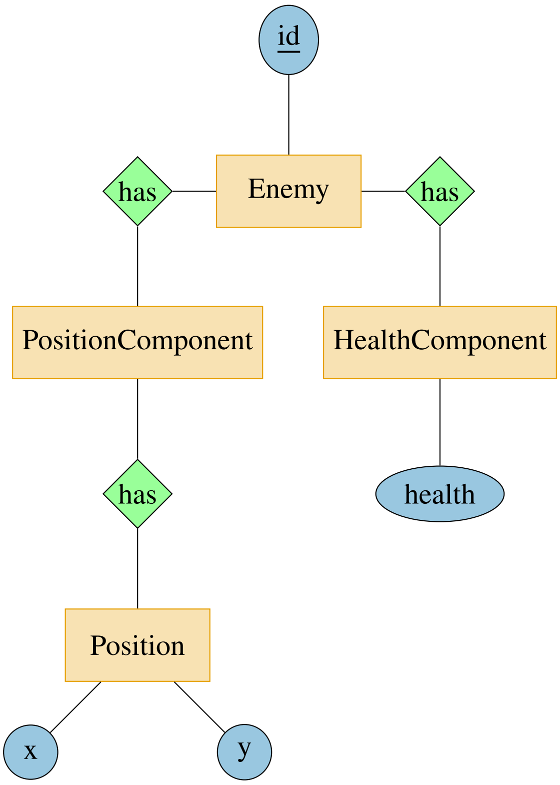
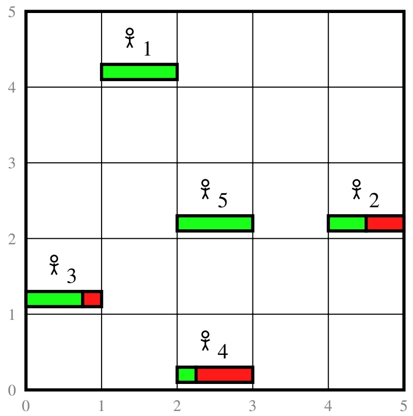


FIGURE 1.4: *Entity–relationship diagram (ERD)* for the running example which follows the *Entity Component System*, instead of classic inheritance and encapsulation from OOP.

included artists, writers, designers, and others. (Bates gives one possible breakdown for a development team in [8, Chapter 8].) Nowadays, code can serve as a scaffold, while content is provided in the form of *data*. This allows teams to be split up into different divisions, where core functionality and content can be developed independently. It also facilitates the use of user-provided content: the players themselves may enrich the games with their own ideas, increasing the *replay value* of a game. Game developers may provide their players with additional tools to produce assets, rule sets, and other types of content, and means to share them with fellow players.

But not only have video games shifted towards a paradigm that is agreeable with databases, SQL has also been enriched with features that make it much more akin to what programmers might expect to find in a programming language, as is elaborated in Section 1.2.



(a) Playing field with five enemies. As in the running example, each enemy has an ID (subscripted), a position, and health. Health may range from 0 to 100 in this example and is displayed through *health bars* in this figure, as is common in video games. The green (left) part of the bar signifies the remaining health points. For example, 🯅$_4$ has 25 health left.

enemy

id	x	y	health
1	1	4	100
2	4	2	50
3	0	1	75
4	2	0	25
5	2	2	100

(b) Enemies in tabular notation. Singular name for consistency reasons.

1	1	4	100	2	4	2	50	3	0

1	75	4	2	0	25	5	2	2	100

(c) Entries from Figure 1.5(b) stored using row-oriented memory layout, where all rows with their respective columns are stored in memory successively.

1	2	3	4	5	1	4	0	2	2

4	2	1	0	2	100	50	75	25	100

(d) Entries from Figure 1.5(b) stored using column-oriented memory layout, where all entries of the columns are stored in succession.

FIGURE 1.5: Several actors represented visually, relationally, and in-memory.

1.2 Recursive SQL Queries

SQL:1999 introduced the concept of *recursive queries* to the SQL standard [85]. Since then, many prominent DBMSs have gradually implemented this feature (*Oracle*: 2009 [54], PostgreSQL: 2009 [33], *SQLite*: 2014 [34], *MariaDB*: 2016 [31], *MySQL*: 2018 [32], *DuckDB*: 2019 [30]). This language feature allows results to be built up gradually, each addition to the resulting relation possibly being computed from intermediate results of earlier passes. As many of the ideas presented in this thesis hinge on this concept, the semantics of recursive queries are outlined below, following the syntax of PostgreSQL. As an example, the running sample of applying gravity to position components will be demonstrated using a recursive query, but instead of pulling each component down by just one unit, gravity will pull them down until they firmly stand on solid ground, i.e. until they collide with a platform. The relational input and a visual representation are given in Figure 1.6.

A recursive *common table expression (CTE)* consists of a non-recursive and a recursive subquery, as shown in Figure 1.7(a). The non-recursive subquery acts as a seed for further calculations and can be any valid expression, including the selection of literals or values from a table, as in the example. The recursive part may reference the CTE that is in the process of being built. Doing so actually references the *intermediate table* $\mathbb{R}_I$ of that CTE, which only contains the values of the execution of the last step of the recursion, or the seed respectively. This is outlined in Figure 1.7(b), where each computation step n was generated from $\mathsf{apply\_gravity}^{\circlearrowright}{}_{n-1}$. Additionally, a monotonously growing *working table* $\mathbb{R}_W$ is maintained, which contains the cumulative results of $\mathbb{R}_I$, shown in Figure 1.7(c). The recursion stops when $\mathbb{R}_I$ evaluates to an empty table, i.e. no new rows are added to $\mathbb{R}_W$. This could happen by a restricting predicate, as is the case in the example query, or by utilising `UNION` semantics (instead of `UNION ALL`), which are evaluated against $\mathbb{R}_W$ to identify duplicate values over all steps of the recursion. In this particular example, the query attempts to move all position components down by one unit unless that would place them inside any of the platforms, only computing successive results for components that have not yet collided. This becomes evident in $\mathsf{apply\_gravity}^{\circlearrowright}{}_3$, which contains only one row, as the position component with $\mathsf{pos\_id} = 1$ has already reached its final position. The collected rows of $\mathbb{R}_W$ then make up the contents of the CTE.

Note that multiple comma-separated CTEs $C_{1 \dots n}$ can be defined per query after the `WITH`-keyword, preceding the actual query. Also, each C_i may reference earlier $C_j | j < i$. In PostgreSQL, if at least one C_i is recursive, even if it is not the first in the list of CTEs, the whole list must be preceded by `WITH RECURSIVE` instead:

```
WITH RECURSIVE
a(x)   AS (...), -- non-recursive
b↻(x)  AS (...), -- recursive
c(x)   AS (...)  -- non-recursive
SELECT
    *
FROM
    a,b↻,c
```

As this blend of recursive and non-recursive expressions will be encountered frequently in this thesis, recursive CTEs will be pointed out by a superscripted "↻", as shown above. This notation, although consumable by the DBMS, is not required by PostgreSQL and solely serves as visual cue for the reader. CTEs may contain *side effects* which manipulate tables by inserting, deleting, or updating tows. This kind of CTEs will be prefixed with `se_` in this thesis to emphasise that the result of a CTE is probably of little interest – albeit the `RETURNING` clause may be used to execute side effects *and* generate intermediate data in one go.

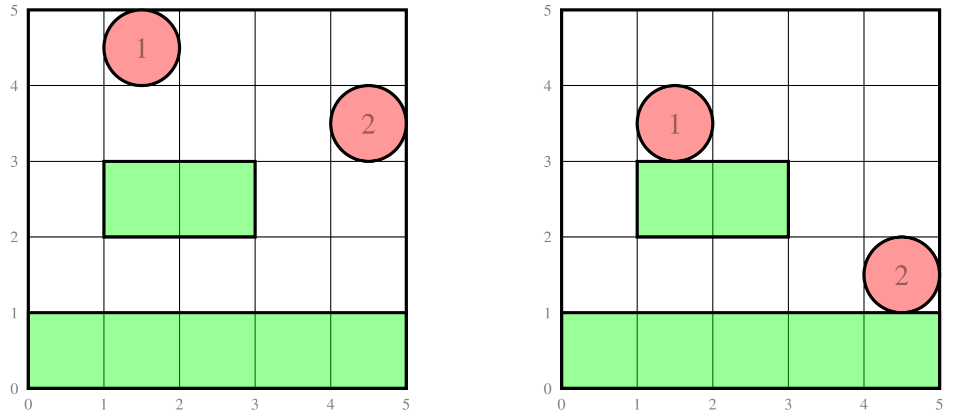
position_components

<u>id</u>	x	y
1	1	4
2	4	3

platforms

x	y	width
0	0	5
1	2	2

(a) Relational representation of the platforms and the initial position components used in the example. A platform is represented by its bottom left corner and its width. Each platform has a height of 1.



(b) Before gravity is applied.

(c) After gravity was applied.

FIGURE 1.6: Relational and visual representation of position components and platforms, used to demonstrate gravity application using a recursive query. Position components are marked with their id. Components do not refer to any entities in this example for brevity.

```
WITH RECURSIVE
apply_gravity↻(pos_id, x, y)  AS (

    SELECT
        pc.id AS pos_id,
        pc.x  AS x,
        pc.y  AS y                               } non-recursive part
    FROM
        position_components AS pc

    UNION ALL

    SELECT
        ag.pos_id AS pos_id,
        ag.x      AS x,
        ag.y - 1  AS y
    FROM
        apply_gravity↻ AS ag                                       } recursive part
        LEFT JOIN platforms AS p
          ON ag.y - 1 = p.y
             AND ag.x BETWEEN p.x AND p.x + p.width
    WHERE
        p.x IS NULL

)
SELECT
    pos_id,
    MIN(x),
    MIN(y)
FROM
   apply_gravity↻
GROUP BY
    pos_id
```

(a) Application of gravity to all position components until they land on a platform.

apply_gravity↻ $_1$

pos_id	x	y
1	1	4
2	4	3

apply_gravity↻ $_2$

pos_id	x	y
1	1	3
2	4	2

apply_gravity↻ $_3$

pos_id	x	y
2	4	1

apply_gravity↻ $_4$

pos_id	x	y

(b) Intermediate table of apply_gravity↻ in each recursive step. apply_gravity↻ $_4$ is an empty table.

apply_gravity↻

pos_id	x	y
1	1	4
2	4	3
1	1	3
2	4	2
2	4	1

(c) Final contents of apply_gravity↻.

output

pos_id	x	y
1	1	3
2	4	1

(d) Result of the query in Figure 1.7(a).

1.3 OpenRA

OpenRA, shown in Figure 1.8, is an open source game engine that focusses on real time strategy games. It started out as a reimplementation of the game *Red Alert* from the *Command and Conquer* series in 2007. The project has since been abstracted into a more general engine on which several aged real time strategy games have been reimplemented, but is also used to create heavily modified versions of classic games or implement new games from ground up. In this work, OpenRA is used as a vehicle to test some of the implemented components. This serves as a proof of concept that the presented techniques can indeed be seamlessly integrated into existing engines without requiring the entire runtime to be hurled over into the DBMS at once. It also acts as a hands-on experiment as to how deep one has to plunge into the innards of an engine to integrate a database into the established work flow. OpenRA was selected as a target platform over more prevalent competitors like, say, Unity, for the following reasons:

- Since OpenRA is completely open source and under GNU General Public License, all parts of it can be inspected and modified as seen fit.
- The engine is still under active development with a diligent community to contact for inquiries.
- Being written in C#, all core components are realised in an OOP language, which, as explained in the preceding sections, is still the de facto standard in the gaming industry.
- By concentrating on real time strategy games, OpenRA gives a clear-cut set of mechanisms that target the use case of managing several actors at once, which is especially suited for the use with DBMSs.

Chapters on features that were integrated into OpenRA have a brief elaboration on the process of incorporating the database into the respective component.

Connect to a forum account to identify yourself to other players, join private servers, and display badges.

FIGURE 1.8: Screenshot of the open source real time strategy game engine OpenRA. Displayed is the main menu of the reimplementation of *Red Alert*.

1.4 Methodology

1.4.1 Setup

All experiments in this write-up were conducted on a Lenovo Ideapad L340 81LW00B6GE with the following specifications:

- CPU: AMD Ryzen 7 3700U CPU
- RAM: 16 GiB of SODIMM DDR4
- Secondary memory: INTEL SSDPEKNW512G8L 512 GB SSD

running a 64 bit Manjaro 20.1 with Kernel version 5.4.64-1.

The presented SQL queries were all executed on a PostgreSQL 12.4 server with the following configuration adjustments:

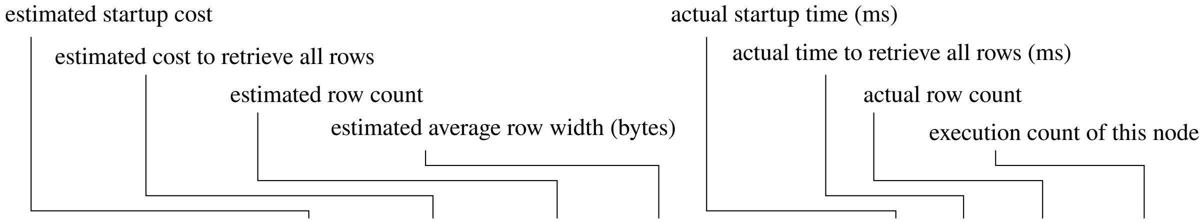
- shared_buffers = 2 GB
- effective_cache_size = 8 GB
- maintenance_work_mem = 512 MB
- wal_buffers = 16 MB
- random_page_cost = 1.1
- effective_io_concurrency = 200
- work_mem = 40 MB
- min_wal_size = 100 MB
- max_wal_size = 2 GB

These parameters were determined by consulting the configuration recommendations in the PostgreSQL wiki [64]. Queries were executed in isolation from the `psql` command line utility where appropriate and then timed utilising the "`\timing on`" switch PostgreSQL provides to measure the wall clock time to execute the query and fetch the result back to the client. To mitigate any caching or loading effects, queries were executed multiple times in succession to average the runtime. Whenever consecutive execution led to extensive variation between runs, e.g. when side effects influenced the workload, the minimum and maximum runtime is given in addition to the average with an accompanying explanation for the deviation. If an execution in isolation is not feasible, i.e. when the query has to be called from within another application, the measurement will be done accordingly. When runtimes call for more detailed analysis, the query plans produced by the DBMS will be consulted, which are outlined in Section 1.4.2.

1.4.2 Query Plans

PostgreSQL allows for a deep inspection of the plan employed by the DBMS. In practice, a DBMS can devise multiple plans for a query and estimate their respective execution cost. These options are then analysed to find the most effective course of action [65]. The selected sequence of operations and costs thereof can also be inspected by human readers which grants insight into bottlenecks and expensive operations. Preceding a query with `EXPLAIN` retrieves information about the selected plan and relevant estimations made by the planner, and using `EXPLAIN ANALYZE` also retrieves the time it actually took the DBMS to execute the plan. As this feature is utilised throughout this thesis and the presented output is specific for PostgreSQL, a quick rundown of the syntax of the plan in Figure 1.9 is given below. The plan was devised from the execution of the query shown in Figure 1.7(a).

Plans are generally trees where each node represents an operation. For example, a join usually consists of two child nodes, each producing a relation, and a condition on which the join takes place. The structure of this tree is visualised by indentation and arrows towards child nodes (`->`). While the nodes themselves can already give some discernment on inefficiently used operators or indices, the estimated and actual cost for executing a node grant additional insight. The estimated costs determined by the planner are displayed as a tuple. The unit for costs is arbitrary and depends on the planner's configuration. It could for example be the number of disk page fetches, but this piece of information is mainly intended to be able to compare costs of different plans [68, 66]. The specification of `cost` consists of *start-up cost* and *total cost*. E.g. `cost=18867.87..18869.87` denotes the estimated start-up cost of 18867.87 before the actual output phase can begin. It is followed by the estimated total cost of 18869.87 for retrieving *all* tuples, which is not necessarily done if a limiting clause is implemented. The planner expects 200 rows to be produced, which will have an average width of 12 bytes. The components of `actual time` are given in milliseconds and are therefore not directly comparable to the estimated costs, although being similarly split into a startup time and a time the query runs until completion. The final piece of information `loops` shows how many times the plan node was executed. This is especially relevant for nodes which are executed more than once by a parent node (e.g. in a nested loops joins or the recursive part of a `WITH RECURSIVE`-CTE). If a node is executed multiple times, `actual time` is an average over all iterations and the total runtime is the number of iterations times the actual time. When consulting the query plan, a single typical configuration for the input parameters is selected and executed using the `auto_explain` extension, to inspect plans of deeply nested calls within *user-defined functions (UDFs)*.


estimated startup cost
estimated cost to retrieve all rows
estimated row count
estimated average row width (bytes)
actual startup time (ms)
actual time to retrieve all rows (ms)
actual row count
execution count of this node

```
HashAggregate  (cost=18867.87..18869.87 rows=200 width=12) (actual time=0.316..0.378 rows=2 loops=1)
  Group Key: apply_gravity.pos_id

  CTE apply_gravity
    ->  Recursive Union  (cost=0.00..13513.34 rows=194710 width=12)
                         (actual time=0.022..0.274 rows=5 loops=1)
          ->  Seq Scan on position_components pc  (cost=0.00..30.40 rows=2040 width=12)
                                                  (actual time=0.011..0.022 rows=2 loops=1)
          ->  Hash Anti Join  (cost=55.90..958.87 rows=19267 width=12)
                              (actual time=0.040..0.060 rows=1 loops=3)
                Hash Cond: ((ag.y - 1) = p.y)
                Join Filter: ((ag.x >= p.x) AND (ag.x <= (p.x + p.width)))
                Rows Removed by Join Filter: 0
                ->  WorkTable Scan on apply_gravity ag  (cost=0.00..408.00 rows=20400 width=12)
                                                        (actual time=0.004..0.011 rows=2 loops=3)
                ->  Hash  (cost=30.40..30.40 rows=2040 width=12)
                          (actual time=0.045..0.055 rows=2 loops=1)
                      Buckets: 2048  Batches: 1  Memory Usage: 17kB
                      ->  Seq Scan on platforms p  (cost=0.00..30.40 rows=2040 width=12)
                                                   (actual time=0.008..0.021 rows=2 loops=1)
  ->  CTE Scan on apply_gravity  (cost=0.00..3894.20 rows=194710 width=12)
                                 (actual time=0.029..0.279 rows=5 loops=1)
Planning Time: 0.522 ms
Execution Time: 0.667 ms
```

FIGURE 1.9: Full execution plan for the query in Figure 1.7(a), generated by preceeding the query with `EXPLAIN ANALYZE`.

1.5 Structure of This Document

The remainder of this document is structured as follows: three components that could typically be found in video game engines are implemented in pure SQL to showcase how relational versions of these components can look like, how their relocation into a DBMS would impact the engine, and how it could be beneficial. The first visited component is *artificial intelligence (AI)* in the context of video games, which is considered on the strategic level of forming a long term plan using *Monte Carlo tree search (MCTS)*, and on the tactical level of performing smaller steps towards the greater goal with the use of *deterministic finite automatons (DFAs)*. Next is the generation of terrain on which games are played on, for which two custom algorithms are explored; one expanding a seed using a set of expansion rules, the other using predefined building blocks to put a map together iteratively. The third component is path finding, specifically through the use of the $A*$ algorithm, which is extended into the temporal dimensions using a booking system to avoid collisions of actors during the path finding phase. Each of these chapters features a separate evaluation section, in which gathered measurements are presented and assessed. The final chapter gives conclusions and critical review on how well the marriage between databases and video game engines worked and in what areas further potential can be perceived. As the subject area with its three independent components is rather broad, related work is given at the appropriate places instead of in a separate central chapter. All chapters contain listings to outline their accompanying implementations. Since the required relations differ largely in complexity, varying notations were selected for their definitions, based on what we deemed the most fitting and most easy to grasp in each given context. Appendix A contains *data definition language (DDL)* statements for all tables and views used throughout this work as unified reference in alphabetical order.

Chapter 2

Video Game AI

2.1 Artificial Intelligence in Video Games

The term AI in the context of video games, although well established [75], may be a bit misleading to academic readers. While games have long been a vehicle to further AI and put new techniques to the test [43, 81, 79], it does not necessarily refer to AI techniques such as *machine learning*, *neural networks*, or *general problem solving* techniques. Instead, AI in video games refers to the *reactive*, *proactive*, or *amending* behaviour the computer displays when partaking in a game, that may appear to be intelligent. Proactive and reactive behaviour mostly concerns computer-controlled agents, also called *non-player characters (NPCs)*, while amending behaviour may apply to agents controlled by the player, also called *playable characters (PCs)* or sometimes *avatars*. That can for example include

- phrases NPCs utter, either randomly (proactive), or when prompted by the player as part of a dialogue (reactive);
- behaviour NPCs display towards the player or other NPCs (reactive/ proactive);
- simple – in some cases scripted – behaviour, like jumping in place at varying heights (reactive/ proactive);
- complex behaviour with the goal to simulate human players with varying strategies and skill level (reactive/ proactive);
- routes NPCs follow either actively to simulate natural behaviour, such as a fisherman walking along a coast line (reactive/ proactive), or as part of the path search for a PC (amending);
- tactics and strategies the computer follows to achieve certain goals, such as opening a door, retreating from battle, or winning the game (reactive/ proactive).

Decision-making can take place on a *strategic* level, which encompasses the long term goals, or on a *tactical* level, which refers to smaller steps that are taken to work towards the long term goal. If a tactic is simple enough, e.g. having an NPC directly attack the PC on sight, it can easily be expressed by a DFA [75, Chapter 4.7.10][74, Chapter 5.18], which is discussed in Section 2.2. Expressing the overarching strategy in the same way can quickly lead to very large, unwieldy DFAs. This calls for abstract modelling, coupled with more sophisticated decision-making. MCTS, as one very recent technique to achieve this, is discussed in Section 2.3.

It is important to note that the goal of an industrial video game AI (as opposed to AI as used in academia) is *not* necessarily to find the optimal way to play the game, but to **present the player with an adequate challenge**, which neither exceeds, nor falls below the player's skills

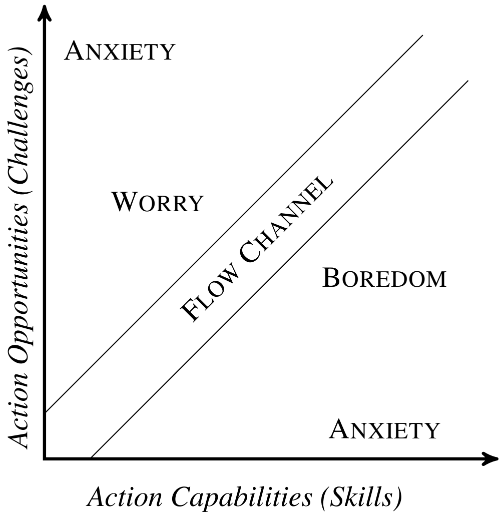

Anxiety
Worry
Flow Channel
Boredom
Anxiety
Action Opportunities (Challenges)
Action Capabilities (Skills)

FIGURE 2.1: Flow channel after Csikszentmihalyi. A task that is balanced between the challenge-level and skills of a person keeps them inside the Flow channel. Deviating from the Flow channel makes the person experience boredom, worry, or anxiety, respectively. Figure is created after [22, Figure 1].

drastically. This type of challenge that is aligned with a person's skill is known as *Flow experience*, which has been coined by Csikszentmihalyi [22]. Originally observed in extreme athletes, like rock climbers or race car drivers, Csikszentmihalyi describes a Flow experience as a subjective state in which the experiencing person is completely absorbed in an activity, illustrated in Figure 2.1, which requires "a clear set of goals", "balance between perceived challenges and perceived skills", and "presence of clear and immediate feedback" [23, p. 230 ff.]. Csikszentmihalyi postulates Flow as a strong motivator, therefore making it desirable to not always have the AI play as good as possible [98]. Some games, like *Left for Dead* [11], *God Hand* [16], and the *Half Life* game engine with its *Hamlet* system [41] even go as far as trying to read the skills of a player during a game and adjusting the difficulty dynamically. Presenting ways to make a video game AI optimal in terms of its decisions is therefore not the scope of this work.

2.2 Tactics Through DFA

Even after an overall strategy has been determined, the execution needs to be broken down into small, explicit pieces. For example, while the strategy in a real time strategy game may be to directly attack an enemy with all available forces, each actor still needs to locate the enemy, find a path to get there, follow that path until the enemy is in range, and then start attacking. Including these finely grained actions in the overall strategy would blow up the search space. Instead, the planning of the strategy can take place on a more abstract level and just include a direct attack on the enemy. White et al. propose an entire scripting language *Scalable Games Language (SGL)*, which allows game developers to define finely-grained behaviour for actors, which is then transformed into relational algebra [97, 84, 3].[1] But the steps each actor needs to follow to partake in that attack can also be encoded in a DFA, as shown in Figure 2.2.

Assuming a data-centric point of view exposes the suitability of database systems for this task: DFAs, states, and edges are each implemented as rows in separate tables, maintaining relationships through foreign keys. Each actor knows which DFA is currently controlling them

[1] SGL was later extended to model crowd simulations in the same vein [96].

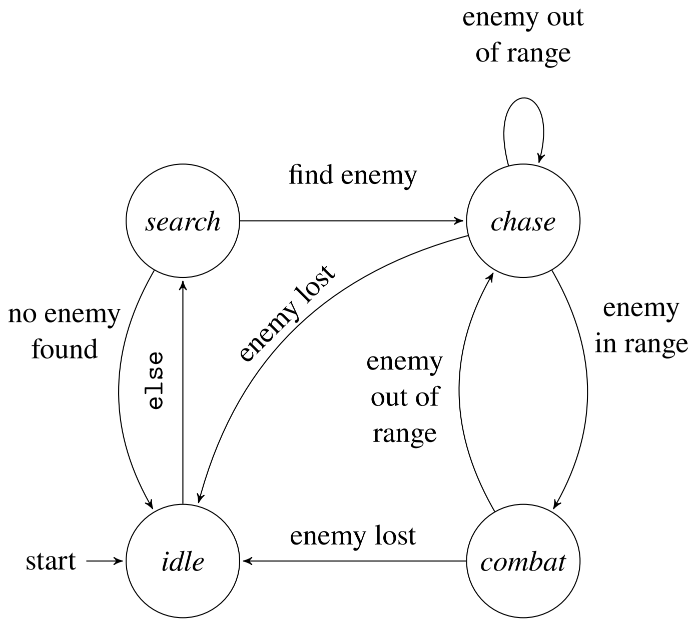


FIGURE 2.2: DFA representing a simple tactic for an NPC. The NPC is *idle* at first and automatically transits into the *search* state (`else` is an automatic transition if no other condition holds). In the *search* state it will look for the closest enemy. Once the NPC has found an enemy, it will start to *chase* (move towards) it and engage in *combat* as soon as the enemy is within the NPC's range. Losing the enemy – for example because that particular actor was defeated – restarts the DFA.

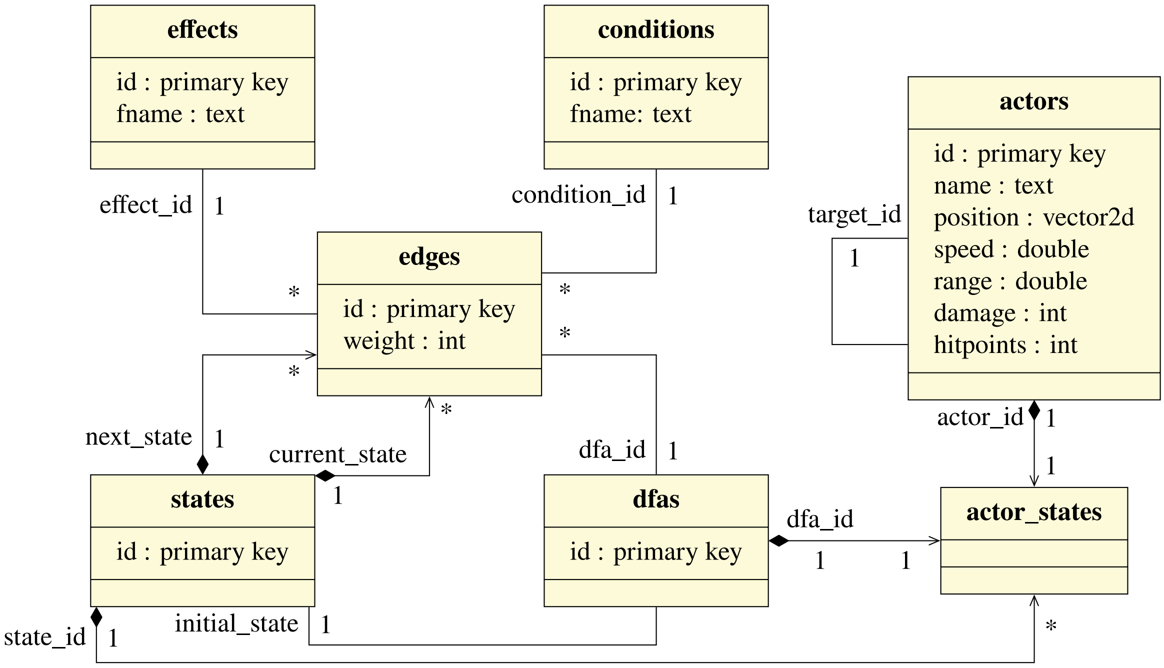


FIGURE 2.3: Schema required for running tactics through DFA. The table **actors** is a simplified supplementary table which contains actors as they are used in the running example in this chapter. The type vector2d is a custom data type which supports standard vector operations, which are assumed to be present in the guise of UDFs prefixed "`vector_`".

and what state they are on. Each edge has a *condition* that needs to be fulfilled in order for the residing actor to traverse the edge. Conditions are expressed as functions with the signature $f(actor\_id) \rightarrow boolean$ which are checked on every tick. If more than one edge passes the check, only the one with the highest `weight` is selected to avoid turning the DFA into an *non-deterministic finite automaton (NDFA)*, where an actor passes multiple edges simultaneously and then finds itself in more than one state. When an actor passes an edge, an *effect* can be triggered. Effects are functions with signature $f(actor\_id) \rightarrow void$, which may change the actor or other parts of the game state through side effects.
Since SQL does not support *higher order programming* natively, in which functions are treated as data as well, this approach employs a relaxed version of *defunctionalisation* for database systems as proposed by Grust et al. [37]. While their original approach allows for arbitrary function signatures and nested closures, the two described signatures for conditions and effects are sufficient to express any required functionality for our DFAs: both conditions and effects are allowed to fully query and manipulate the current game state. The only required information – around which actor the queries and updates need to be centred – is passed as a parameter. Conditions only need to return boolean values to determine whether an edge should be traversed, eliminating the need for arbitrary return types. This signature unification is required to properly express conditions and effects through a foreign key relation. Since each function signature requires its own table to store return and parameter types properly, we could not reference functions easily without also expressing in which table they reside. While only allowing functions with these two fixed signatures simplifies the process, generating a dispatcher for each signature is still required. Game developers can specify their DFAs with any desired transitions using conditions and effects in a higher order fashion and then systematically generate the dispatcher as described in [37]. The following section exemplifies the full realisation of the DFA from Figure 2.2 in pure SQL, for which the schema is given in Figure 2.3. As the shape of the actors and the effects and conditions that operate on them is highly dependent on the game at hand, we use a simplified schema for actors in the running example for DFAs.

First, we need to express all required conditions in the form of UDFs and implement the dispatcher `dispatch_condition`. Note that UDFs prefixed "`vector_`" are textbook vector calculations and their implementation is omitted for brevity. The condition functions prefixed "`cond_`" are related to the edges in Figure 2.2. The function `cond_true`, which is always satisfied, is used to express `else`-edges, which are edges that can always be traversed without having to fulfil a specific condition.

```sql
CREATE FUNCTION cond_true(_actor_id INT)
RETURNS BOOLEAN AS $$
    SELECT TRUE
$$ LANGUAGE sql;

CREATE FUNCTION cond_has_target(_actor_id INT)
RETURNS BOOLEAN AS $$
    SELECT
        COALESCE(that.hitpoints > 0, FALSE)
    FROM
        actors AS this,
        actors AS that
    WHERE
        this.id = _actor_id
        AND that.id = this.target_id
$$ LANGUAGE sql;

CREATE FUNCTION cond_target_in_range(_actor_id_this INT)
RETURNS BOOLEAN AS $$
    SELECT
        vector_distance(this.position, that.position) < this.range
    FROM
        actors AS this
        JOIN actors AS that
          ON this.target_id = that.id
    WHERE
        this.id = _actor_id_this
```

```
$$ LANGUAGE sql;

CREATE FUNCTION dispatch_condition(_fname TEXT, _actor_id INT)
RETURNS BOOLEAN AS $$
    SELECT CASE _fname
        WHEN 'cond_target_in_range' THEN cond_target_in_range(_actor_id)
        WHEN 'cond_has_target'      THEN cond_has_target(_actor_id)
        WHEN 'cond_true'            THEN cond_true(_actor_id)
    END
$$ LANGUAGE sql;
```

Effects are expressed in the same fashion as UDFs prefixed "`eff_`" with a designated dispatcher `dispatch_effect`. Here, each UDF expresses what happens when an actor resides on a certain state. Again, the functions are related to what can be seen in Figure 2.2.

```
CREATE FUNCTION eff_attack_target(_actor_id_this INT)
RETURNS VOID AS $$
    UPDATE actors AS that SET
        hitpoints = that.hitpoints - this.damage
    FROM
        actors AS this
    WHERE
        that.id = this.target_id AND this.id = _actor_id_this
$$ LANGUAGE sql;

CREATE FUNCTION eff_target_closest(_actor_id_this INT)
RETURNS VOID AS $$
    WITH
    target(actor_id) AS (
        SELECT
            that.id
        FROM
            actors AS this,
            actors AS that
        WHERE
            this.id = _actor_id_this
            that.id <> _actor_id_this
        ORDER BY
            vector_distance(this.position, that.position) DESC
        LIMIT 1
    )
    UPDATE actors AS this SET
        target_id = t.actor_id
    FROM
        target AS t
    WHERE
        this.id = _actor_id_this
$$ LANGUAGE sql;

CREATE FUNCTION eff_move_towards_target(_actor_id_this INT)
RETURNS VOID AS $$
    WITH movement(this_v, that_v, this_speed, diff_v) AS (
        SELECT
            this.position,
            that.position,
            this.speed,
            vector_substract(that.position, this.position)
        FROM
            actors AS this
            JOIN actors AS that
              ON this.target_id = that.id
        WHERE
            this.id = _actor_id_this
    )
    UPDATE actors AS this SET
        position = vector_add(m.this_v,
                              vector_scale(m.diff_v,
                                           LEAST(m.this_speed,
                                                 vector_length(m.diff_v))))
    FROM
        movement AS m
    WHERE
        this.id = _actor_id_this
$$ LANGUAGE sql;

CREATE FUNCTION dispatch_effect(_fname TEXT, _actor_id INT)
RETURNS VOID AS $$
    SELECT CASE _fname
        WHEN 'eff_move_towards_target' THEN eff_move_towards_target(_actor_id)
        WHEN 'eff_target_closest'      THEN eff_target_closest(_actor_id)
        WHEN 'eff_attack_target'       THEN eff_attack_target(_actor_id)
    END
$$ LANGUAGE sql;
```

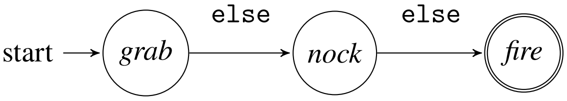


FIGURE 2.4: Example of `else`-edges coming in handy with a fixed sequence of states requiring no further conditions to be fulfilled.

Finally, the tick itself is also expressed through a UDF. Each actor evaluates the conditions for all outgoing edges of the state they are currently on. One of the edges which have their condition fulfilled determines the new state `e.next_state` for the executing actor. Utilising `JOIN LATERAL` in conjunction with `LIMIT 1` ensures that each actor traverses only up to one edge, avoiding situations where a single actor ends up in more than one state when multiple conditions evaluate to `TRUE`. Selecting `dispatch_effect(...)` (which always returns `VOID`) dispatches appropriate changes to the actor or the world around it as a side effect. All actors are then advanced to the target state of the edge they traversed in an `UPDATE`-statement.

```
CREATE OR REPLACE FUNCTION tick()
RETURNS VOID AS $$
    WITH updates(actor_id, next_state, effect_result) AS (
        SELECT
            advance.actor_id,
            advance.next_state,
            dispatch_effect(advance.fname, advance.actor_id)
        FROM
            actors AS a
            JOIN LATERAL (
                SELECT
                    states.actor_id,
                    e.next_state,
                    ef.fname
                FROM
                    actor_states AS states
                    JOIN edges AS e
                      ON (states.dfa_id, states.state_id) = (e.dfa_id, e.current_state)
                    JOIN conditions AS c
                      ON e.condition_id = c.id
                    LEFT JOIN effects AS ef -- effects can be NULL!
                      ON e.effect_id = ef.id
                WHERE
                    a.id = states.actor_id AND
                    dispatch_condition(c.fname, states.actor_id)
                ORDER BY
                    e.weight DESC
                LIMIT
                    1
            ) AS advance(actor_id, next_state, fname) ON TRUE
    )
    UPDATE actor_states AS current SET
        state_id = updates.next_state
    FROM
        updates
    WHERE
        current.actor_id = updates.actor_id
$$ LANGUAGE sql;
```

States can be amended with `else`-edges. For one, they facilitate effects that are to be applied when an actor should stay within the same state by creating an `else`-edges that loops back to the current state. Allowing them also enables time-sequential actions that need to be executed sequentially with each tick without having to fulfil any other conditions. For example an archer firing an arrow would require the actor to draw an arrow from the quiver, nock it, and then fire it at the target, as shown in Figure 2.4. These transitions do not require any additional conditions, as they just need time to pass. `else`-edges should always have the lowest `weight` to only be taken when no other edge is available, which makes them akin to the `else`-case in regular branching control flow. It should be noted that such edges differ from ϵ-edges that can be used in NDFAs, which allow the automaton to transit to another state and stay in the same state at the same time by requiring no input. In the case of `else`-edges we actually always have the next tick as input, making for well-defined transition points.

Having these bite-sized tactics available for actors can be used to conceive an overarching strategy by abstracting the tactics into *actions*. The exploration of a game tree is explained in the next section.

2.3 Monte Carlo Tree Search

After the IBM supercomputer *Deep Blue* had beaten Garry Kasparov in 1997 [56], the Chinese board game *Go*, shown in Figure 2.5, became the new frontier for game AI. Go has considerably less complex rules than chess: the two players alternately place immobile playing pieces, also called *stones*, on a 19×19 board (other dimensions, such as 9×9 or 13×13, are possible). Players try to enclose areas with their stones while avoiding having their own stones enclosed by enemy pieces, which marks them as *captured*. Points are given out by conquering parts of the board and capturing enemy stones. The game ends when no more stones can be placed. The player with the most points at the end of the game wins. Despite this manageable set of rules, Go produces unwieldy game trees. Allis estimates an average game of Go to last around 150 rounds, each having about 250 possible moves, reckoning the game-tree complexity at around 10^{360} [5, p. 174]. This explosion in search space called for the development of new exploration methods of which the most recent extensively used one is MCTS. In October of 2015, the computer program *AlphaGo*, which utilises MCTS, defeated the professional Go player Fan Hui on a 19×19 board without handicap. In March of 2016 AlphaGo defeated Lee Sedol, one of the world's best professional Go players [80].

MCTS at its core iteratively builds a partial game-tree until a computational budget is exhausted, modelling possible states of the game in terms of tree nodes. The tree can be successively expanded while the game is actually being played. The current state of the game can therefore be imagined as a pointer to one of the nodes in the game-tree. Making a move in the game advances that pointer to a child node – preferably one that has been thoroughly explored in an earlier iteration. (The human player may of course choose an action that was not yet considered by the *AI*, putting the game into an entirely unexplored state, effectively outsmarting the computer, which caters to the Flow effect.) Each node maintains a *node statistic* to denote how often it or its children have been visited and how promising it looks in terms of expected reward. The edges from parent to child nodes each hold an *action*, which transforms the parent state to the child state. Such an action can for example be the placement of a stone on the board in the case of Go. Or it can be an abstracted action, as described in Section 2.2. When building this tree, the algorithm tries to explore paths through the tree that are expected

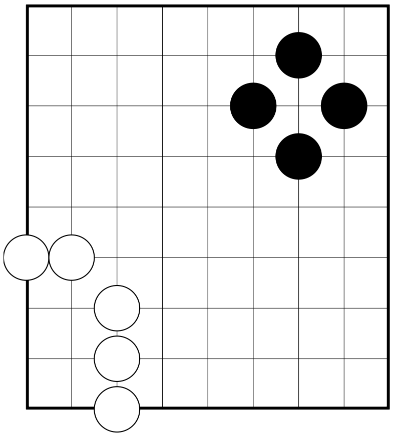
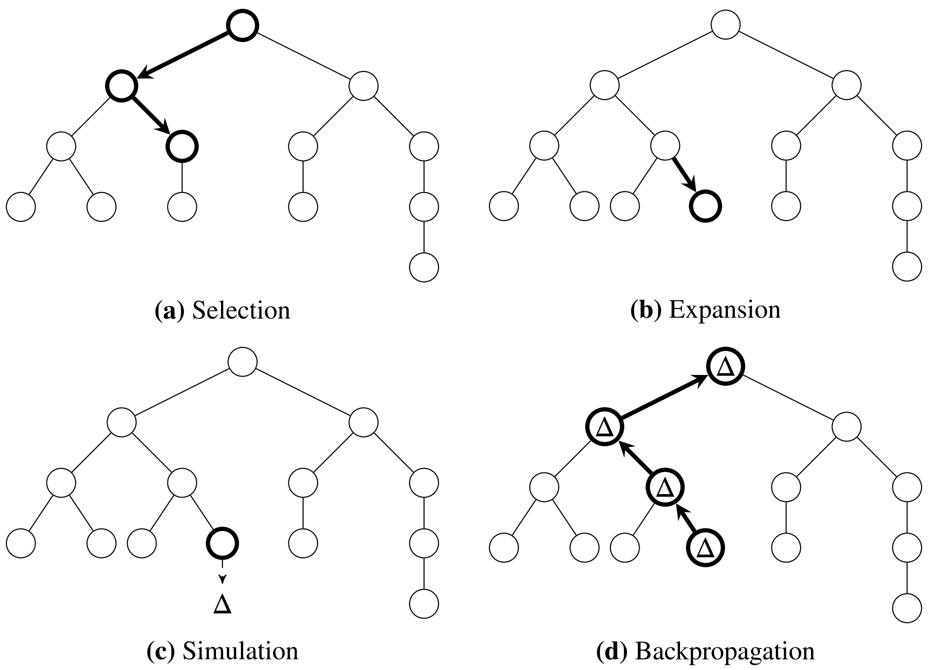


FIGURE 2.5: 9×9 go board. Both players have enclosed small areas of the board with their stones.



(a) Selection

(b) Expansion

(c) Simulation

(d) Backpropagation

FIGURE 2.6: Steps of the MCTS algorithm. Adapted from [13, Figure 2]

to yield high rewards, as well as areas that are not very well explored yet. The latter helps to identify paths through the tree that do not look promising immediately but turn out to have a high long term reward. This optimisation task is also known as the *multi-armed bandit* problem, where an agent is presented with a number of slot machines ("one-armed bandits"), each with an unknown but fixed winning probability. The agent is tasked to play the machines in a way that maximises their reward. In order to find machines with a high chance of winning the agent has to balance how they stick with machines with seemingly high chance of winning and unknown machines that may turn out to have an even higher chance. Browne et. al identify four basic steps for MCTS in [13], which are visualised in Figure 2.6:

1. **Selection:** Find an *expandable* node in the current tree. Expandable nodes are non-terminal states that can still be expanded by adding new children.
2. **Expansion:** Expand that node by applying an action to it that has not yet been applied, producing a new child node.
3. **Simulation:** Run a simulation starting from the newly attached node to determine an *outcome*.
4. **Backpropagation:** Propagate the outcome back up through the ancestor nodes to update their node statistics.

The remaining part of this section will explain the four steps of the algorithm in greater detail and lay out how they can be implemented in SQL.

Selection

The selection starts at the root node and determines a node that has not been fully expanded yet. That is a node to which an action can be applied that has not yet been applied to that node. The selection of the node (*tree policy*) should keep a balance between *exploitation* and *exploration*. Exploitation denotes further expansion of nodes that have already been visited and are known to promise a high reward. Exploration expresses how much unknown areas of the tree are being expanded, even if they do not seem to yield an immediate reward. This ensures finding paths through the tree that only pay off later in the game but may offer an overall better reward than paths that have an immediate reward. Nodes can be selected by various strategies, of which several are discussed in [13].

Expansion

The expansion step is straightforward. An action that has not yet been applied to the node that was selected in the last step is now applied, producing a new state. That state is set as child for the selected node.

Simulation

The simulation process starts at the newly created node and simulates the remaining game. That is, it chooses follow-up states following a *rollout policy* to reach a terminal state, in which the game score can be calculated. That policy can be a uniformly random function. Not every game is suited for random simulations. Imagine a game of chess, where certain moves can be reversed, for example a knight jumping back and forth between two positions. If both players happened to continuously execute such moves, the game can go on indefinitely or even infinitely. Schadd therefore proposes cancelling the random simulation after a few steps, evaluating the current game state heuristically, and instead increasing the number of samples [76]. SQL actually excels at this, as taking multiple samples and unrolling them in parallel for a few steps is more akin to relational thinking than sequentially following one single state for a prolonged time.

Backpropagation

Backpropagation updates the statistics for all ancestor nodes. That is, the newly found reward Δ and an increment of the visit-counter.

Finally, a move that is actually being played is selected from the tree. Various strategies for selecting the move are feasible, of which Chaslot [15] proposes four:

1. *Max child*: the node with the highest predicted reward.
2. *Robust child*: the most visited node.
3. *Robust-max child*: the node with both the highest visit-count and the highest reward (which does not necessarily exist at all times).
4. *Secure child*: the node that maximises a lower confidence bound.

Chaslot et al. have not found a significant difference between the four proposed strategies when the tree was thoroughly explored between moves. Instead, they found *max child* to perform slightly worse than the other strategies when the time constraints were tight. This again underlines the importance of exploring parts of the tree that do not seem to yield a high reward at first glance, but may turn out to be beneficial in the long run. The following section explains the implementation of the steps of MCTS in SQL. The final selection following any of the above strategies can be implemented using a trivial SQL-query and is therefore not shown in detail.

2.3.1 Relational Implementation

The following implementation of MCTS in SQL is agnostic towards the actual game for the most part, except for a few UDFs and views that are pointed out where appropriate. The game *noughts and crosses* is used as a running sample to illustrate the game-specific steps of the implementation. Noughts and crosses is a game for two players, one places × symbols, the other places ∘ symbols. Players take turn placing their symbol in an empty field in a 3×3 grid. A player wins if they manage to have three of their symbol in a horizontal, vertical, or diagonal line. It should be noted beforehand that the basic MCTS produces tree structures. An optional improvement which is elaborated in Section 2.3.2 may instead produce general graphs. But for now assume that the resulting structures are actual trees.

MCTS hinges on forming and traversing game trees. Tree-like structures are easily representable in relational form as a table of parent-child relations. Every row in a `tree_children` table consists of the parent state, the child state and the action that produced the child from the parent. All possible legal actions, depending on the game, are stored within the table `actions`. Any action is referenced by ID and the table itself is therefore independent of the specific game at hand. Arbitrary subtrees can be selected using the UDF `subtree` to only work with specific parts of the tree: since actually playing out a move moves the current game state down one level, all sibling states of the new state become unreachable and exploring them is therefore no longer required.

```sql
CREATE TABLE actions(
    id SERIAL PRIMARY KEY,
    ... -- game specific
);

CREATE TABLE states(
    id SERIAL PRIMARY KEY
);

CREATE TABLE tree_children(
    parent_id INT NOT NULL REFERENCES states(id),
    action_id INT NOT NULL REFERENCES actions(id),
    child_id  INT NOT NULL REFERENCES states(id),
    UNIQUE(parent_id, child_id), -- each tree can only be attached to another tree once
    UNIQUE(parent_id, action_id) -- per parent, each action may only be applied once
);

CREATE FUNCTION subtree(_root INT) RETURNS TABLE(state_id INT) AS $$
    WITH RECURSIVE
    tree↻(state_id) AS (
        SELECT s.id FROM states AS s WHERE s.id = _root
        UNION
        SELECT
            tc.child_id,
        FROM
            tree_children AS tc
            JOIN tree↻ AS t
              ON tc.parent_id = t.state_id
    )
    SELECT state_id FROM tree
$$ LANGUAGE sql;
```

MCTS requires statistics for each node, containing the expected reward when traversing an edge towards that node, and how many times it has been visited. Rewards for each node are stored as separate rows in the table `rewards`. The view `tree_statistics` shows how the total rewards can then easily be calculated by aggregating rewards for the same node. When a node is being inspected and receives a reward, all ancestor nodes of that node receive the same reward as well, which can be explicitly propagated by recursively ascending through the `tree_children` table, as is shown in the UDF `propagate_reward`.

```sql
CREATE TABLE rewards(
    state_id INT REFERENCES states(id),
    reward   NUMERIC
);

CREATE VIEW tree_statistics(state_id, total_reward, visit_count) AS (
    SELECT
        r.state_id      AS state_id,
        SUM(r.reward)   AS total_reward,
        COUNT(r.reward) AS visit_count
    FROM
```

```
        rewards AS r
    GROUP BY
        r.state_id
);

CREATE FUNCTION propagate_reward(_state_id INT, _reward NUMERIC)
RETURNS TABLE(state_id INT, reward NUMERIC) AS $$
    WITH RECURSIVE
    ancestors↻(state_id) AS (
        SELECT
            _state_id
        UNION
        SELECT
            c.parent_id
        FROM
            ancestors↻ AS a
            JOIN tree_children AS c
              ON c.child_id = a.state_id
    )
    INSERT INTO rewards(state_id, reward)
      SELECT a.state_id, _reward FROM ancestors↻ AS a
      RETURNING state_id, reward
$$ LANGUAGE sql;
```

As described above, MCTS continually explores the game tree until a given computation budget is spent. Implementing a strict time limit in pure SQL is not possible, but it can be approximated using `WITH RECURSIVE`: assume `epoch_now()` to be a function that returns the current epoch timestamp as an integer value. Dragging the initial timestamp from the non-recursive term through the recursion, we can check the elapsed time before each descent into the next recursion level, using it to stop the exploration when the budget is used up. This can lead to the query exceeding the budget by far, if the recursive term is a long-running query. Since there is no way to interrupt an already running pure SQL query, this technique may lead to unsatisfactory results but in practice is feasible if the number of unrolls per recursive step is not too large. See also Section 2.4.

```
CREATE FUNCTION mcts(_root INT, _budget INT, _unrolls INT DEFAULT 1)
RETURNS TABLE(state_id INT) AS $$
    WITH RECURSIVE
    exploration↻(state_id, start) AS (
        SELECT _root, epoch_now()
        UNION ALL
        SELECT
            explore(_root, _unrolls), -- always start again from the root
            last.start
        FROM
            exploration↻ AS last
        WHERE
            epoch_now() - last.start < _budget
    )
    SELECT state_id FROM exploration↻
$$ LANGUAGE sql;
```

Selecting a node for expansion is based on two pieces of information: the statistics for the nodes stored in `tree_statistics`, making sure we explore both untouched and promising regions of the tree, and what options we have to further explore a given node. The game developer must define which actions can still be applied to the existing states in a view `applicable`. For noughts and crosses for example, `applicable` contains all moves per node that have not yet been applied to the state itself or any ancestor node, which is manageable, since noughts and crosses features exactly 18 distinct moves. We can then unroll an arbitrary number of nodes (`_unrolls`) by applying an action to them. This unrolling will produce new child nodes in the game tree. Sorting the nodes that can be unrolled by their rating makes sure that urgent nodes are unrolled first. The UDF `rating` has to be implemented by the game developer and could for example be any strategy presented by Browne et al. in [13].

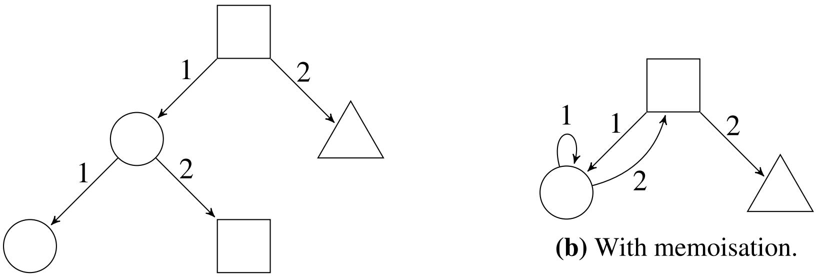
```
CREATE FUNCTION explore(_root INT, _unrolls INT DEFAULT 1)
RETURNS TABLE(state_id INT) AS $$
    WITH
    rollout(state_id, action_id) AS (
 86         SELECT
 87             app.state_id,
 88             app.action_id
 89         FROM
 90             subtree(_root) AS st
 91             JOIN applicable AS app
 92               ON st.state_id = app.state_id
 93             JOIN tree_statistics AS ts
 94               ON ts.state_id = st.state_id
 95         ORDER BY
 96             rating(app.state_id) DESC
 97         LIMIT
 98             _unrolls
 99     )
100     SELECT
101         ex.state_id
102     FROM
103         rollout AS r,
104         LATERAL expand(r.state_id, r.action_id) AS ex(state_id)
105 $$ LANGUAGE sql;
```

The actual unrolling takes place in expand, which creates a new state, the tree edges, and propagates the expected reward back up the tree. The reward is estimated by a UDF evaluate, which is specific to the game at hand. The implementation of the other steps is straightforward and therefore omitted. In the most basic case, the initialisation in line 110 creates a new state in init_new_state which is written to the table of states and then creates an edge between the new state and the state that is being unrolled. This particular place in the algorithm provides us with an opportunity to apply *memoisation* to the game tree, which is elaborated in Section 2.3.2.

```
106 CREATE FUNCTION expand(_state_id INT, _action_id INT)
107 RETURNS INT AS $$
108     WITH
109     child_state(state_id) AS (
110         SELECT COALESCE(find_similar_state(_state_id, _action_id),
111                         init_new_state(_state_id, _action_id))
112     ),
113     se_init_parent_relation(parent_id, child_id) AS (
114         INSERT INTO tree_children(parent_id, child_id, action_id)
115             SELECT
116                 _state_id,
117                 cs.state_id,
118                 _action_id
119             FROM
120                 child_state AS cs
121         RETURNING parent_id, child_id
122     ),
123     se_propagate_reward(state_id) AS (
124         SELECT
125             cs.state_id
126         FROM
127             child_state AS cs,
128             LATERAL propagate_reward(cs.state_id, (SELECT evaluate(cs.state_id)))
129     )
130     SELECT ns.state_id FROM se_propagate_reward AS ns
131 $$ LANGUAGE sql;
```

2.3.2 Memoisation

As hinted at in the last section, the process of expanding the game tree can be enhanced by reusing states that reappear throughout the game by applying memoisation to it. The term memoisation stems from Michie's description of "memo functions", where the result of functions for specific parameters can be stored and later retrieved without recalculating that particular function-parameter-combination [51]. The advantages become obvious when looking at the abstract call graph in Figure 2.7. Different shapes represent different results and edges are either labelled 1 or 2. $\delta(n, e) = n'$ is a function that takes a known node n from the graph and an edge label e and deterministically produces a new node n'. Thus δ is *referentially transparent* in that passing the same arguments to δ always produces the same resulting node [83].



(a) Without memoisation.

(b) With memoisation.

FIGURE 2.7: A tree where equal shapes represent equal states. Figure 2.7(b) shows the benefit of memoisation by having the circle point to already existing states that are in turn partially explored.

A naïve implementation could lead to indefinite or even infinite state expansion. See Figure 2.7(b), where each successive ○-node can produce another identical node by applying 1 to it. Instead, for edges that would result in a node n_i identical to a node n_j that already exists within the graph, the edge points to n_j, as can be seen in Figure 2.7(b). The two nodes n_i and n_j can be *condensed* into one node.
In the case of noughts and crosses (and a variety of other games) different parent states may lead to an identical child state. See Figure 2.8, where the state (3) can be the child node of both node (1) and (2). By combining the resulting states into a singular node (3), that exact state now only needs to be fully explored once. In fact, two nodes can already be condensed if they are *sufficiently similar*. The similarity of two states depends on the particular game at hand and must therefore be defined by the game developer. Line 110 incorporates a call to the UDF `find_similar_state` which gives the developer the opportunity to find a state that is sufficiently similar to the state that would be produced. Providing an implementation for `find_similar_state` also enables the reuse of (sub)trees from past rounds of the same game, or trees crafted by experts. Picture a state that occurs frequently throughout many rounds of a game. That state and its subtree can be included as a disjointed subgraph before the game starts. If some node is recognised within the `find_similar_state` process, that subtree is incorporated into the actual game graph by drawing an edge to it.

The implementation of noughts and crosses features a utility table `state_hashs`, which concatenates the three rows of the whole grid of one state into a string of nine characters (padded with underscores), making it the perfect fit for a hash index, allowing for rapid retrieval of existing states. The developer may choose to have `find_similar_state` always return `NULL` if states are always unique or no feasible way of finding similar states can be devised. In that case the produced graph takes on the shape of a traditional game tree. It should be noted that using general graphs with possible cycles instead of a tree poses no problem for MCTS, as both the descending function `subtree` (line 18) as well as the ascending table `ancestors` (line 51) automatically remove duplicates created by cycles and only continue their evaluation as long as unknown parts of the subgraph are found due to the use of `UNION`.

To ease some of the urgency during the early game, parts of game trees from former rounds can be employed as *opening books*, which are databases containing the first few actions for popular, recurring opening games. Since the whole game tree may remain within the database with only a pointer indicating the actual current game state, the sequence and distribution of moves can be analysed post-mortem. That information can be used to recreate play styles of the human opponent, serving as a basis for *player modelling* [28], [99][Chapter 5], which aims to emulate different characteristics of players to provide a more natural experience when

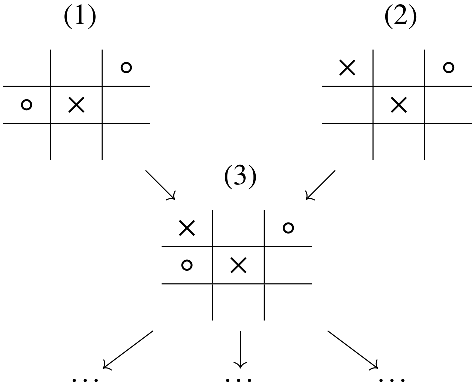
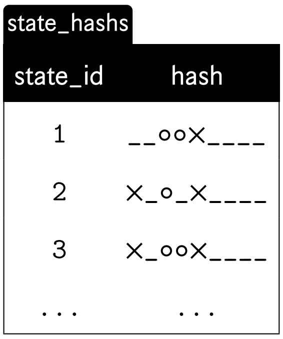


FIGURE 2.8: State (3) can be produced from either placing × in the top left corner of (1) or by placing ∘ in the centre row in the left column of (2). This makes (3) a candidate for memoisation.



state_hashs

state_id	hash
1	__∘∘×____
2	×_∘_×____
3	×_∘∘×____
...	...

FIGURE 2.9: Hash of the states seen in Figure 2.8, making it easy to find similar states.

playing against an AI or to (temporarily) replace players who left ongoing multiplayer matches or fail to provide input in a timely manner [60][Section C], [61].

2.4 Evaluation

2.4.1 MCTS

Expectations for an AI differ from genre to genre: as explained in Section 2.1, AI in video games is rarely expected to play perfectly or to always defeat the player by a landslide. At the same time, most games feature a difficulty setting to adjust the challenge based on player preference or skill, requiring the AI to play masterfully at times and be able behave like a novice at others. A similar variance goes for expected decision speed. While longer reaction times are deemed acceptable in *turn-based strategy* games like *Sid Meier's Civilization* [78], where the player may have to wait several seconds for the AI to finish the turn, many other genres like real time strategy games require swift response to give the player the impression of a vivid opponent. Generally, these requirements come as a trade-off, as more thorough exploration of options will be more likely to yield a promising strategy. The implementation of MCTS described in this chapter offers three major tuning knobs to influence the capability of the AI during runtime:

1. Adjusting the number of most promising nodes that are unrolled during MCTS exploration simultaneously.

2. The time budget the AI is given during which it repeatedly explores the tree.
3. The number of times the tree is revisited. This amounts to restarting the search entirely on the current state of the tree and is therefore the frequency with which the exploration of the tree is triggered.

The effects of increasing the allowed runtime (2 and 3) unsurprisingly leads to a more extensive exploration. Therefore, the effect of varying the number of simultaneous unrolls as the most relevant parameter in the context of DBMSs is observed in the following section on a game of noughts and crosses. A new game tree is created for each experiment which is then partially explored in a warm-up-phase. This is done to present the algorithm with enough unexplored nodes to choose from, which represents a typical state of the tree during the game, as opposed to having a blank slate on which the tree slowly unfolds during the first few iterations before fanning out. The number of existing edges is used to express how thoroughly explored the game tree is at any given time.

Figure 2.10 shows the effect of varying the number of nodes that are rolled out at once over a fixed time budget per iteration, harnessing the ability of the DBMS to apply operations to multiple rows at the same time. The time budget was set to 500 ms as a value that is reasonably small to give a human player the impression of “fast thinking”, while having experimentally proven as large enough to allow for more than one rollout cycle within the allowed time, as should be the case under real conditions. While the number of edges expectedly increases faster the more nodes are unrolled at once, the general overshoot of the allowed time budget also rises. This happens due to how an ongoing SQL query will always run to completion, and since extensive unrolling of multiple nodes will be more likely to take longer than the allowed time, the individual overshooting accumulates accordingly. This overshoot is further illustrated in Figure 2.11, where the average time for completing one iteration with differing rollout parameter is distributed evenly onto the number of nodes that are rolled out. This clearly shows that while the overshoot rises gradually with the number of simultaneous unrolls, the time per unrolled node reduces, which can be attributed to preparatory work that has to be performed by the DBMS in any case, but repays especially when applying the operation to more nodes at once. While the increasing overshoot seems to be a violation of the expectation of low latency, this is only a problem if there is a fixed order of exploring the tree first and then selecting a move afterwards, where the exploration could become a bottleneck and grind the reactivity of the AI to a halt. The well-established ACID properties of DBMSs, especially the isolation property, solve this issue: moves are selected from the tree “as is” whenever the AI needs to make a move while the tree is explored in the background. If the actual game state is near leaf-level, conveying the urgency to generate more edges, short-running explorations with a small rollout factor can be started to make sure the AI still maintains a bit of foresight. During idle times, where the game delves into well-explored areas of the tree, long running explorations can be started. As the proposed implementation offers a parameter to select a root to start the exploration from, multiple explorations with varying rollout-parameter can even be run at the same time in an isolated fashion. The benefit of rolling out multiple nodes at once is also visible in Figure 2.11, where the increase in time to complete a single iteration is compared to the ratio of time spent per iteration to the number of explored nodes. As the number of simultaneously unrolled nodes increases, the overhead of preparing and executing the query is distributed.

2.4.2 DFA

The performance of DFAs as a mean to implement simple behaviour was evaluated using the DFA from Figure 2.2, as it reflects a standard behaviour that is time-tested [38][75][Chapter 4.7.10] and can still be observed in modern video games. To test out the scalability of

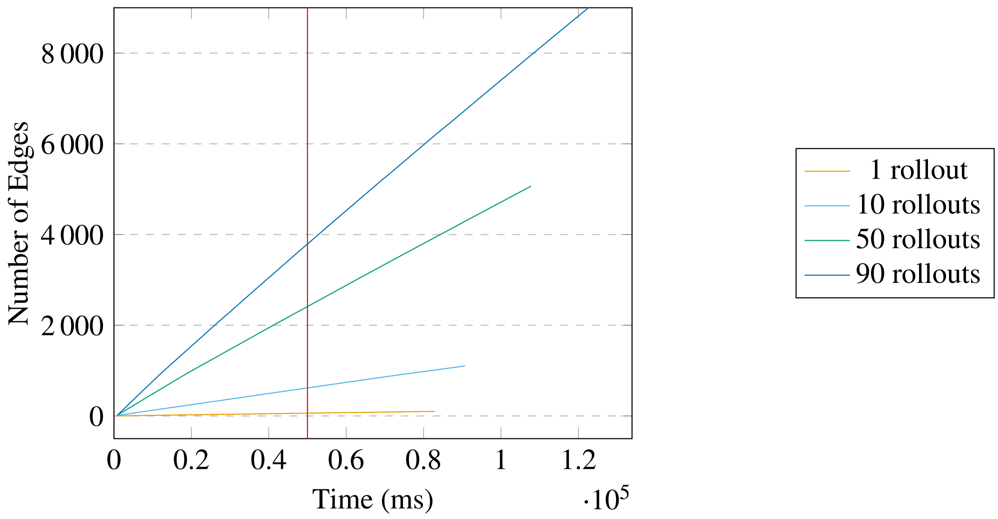


FIGURE 2.10: Different rollouts running with a budget of 500 ms for each iteration, 100 iterations each. The expected finish time is denoted as a red line at 100 × 500 ms = 50 s.

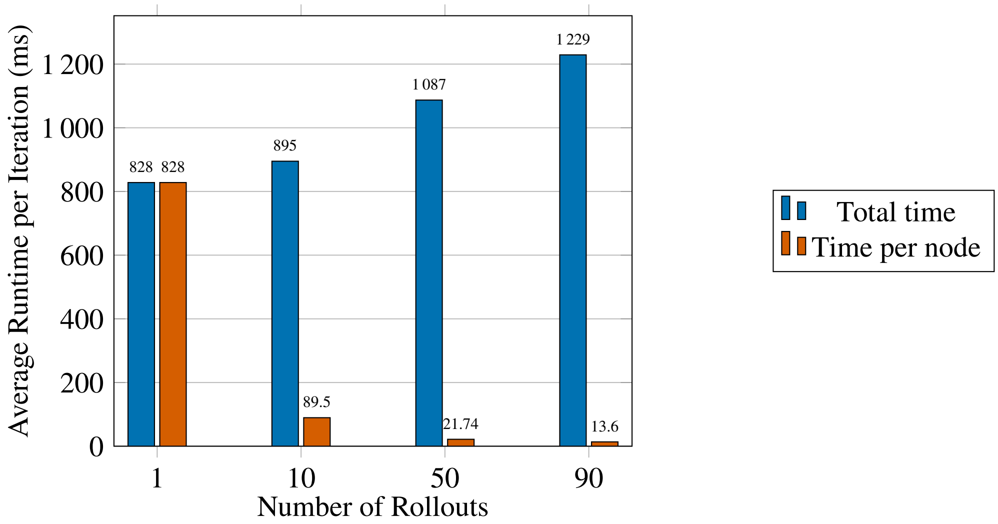


FIGURE 2.11: Average runtime per iteration compared to the average runtime per iteration divided by the number of nodes rolled out simultaneously.

this approach, three experiments were conducted in which 100, 1 000, and 10 000 actors were created on the initial state of the DFA, followed by 100 calls to `tick()` (line 106), shown in Figure 2.12. Runtime expectedly varied depending on what state the majority of actors were in. As the very first state *idle* features only one `else`-edge to the state *search*, where the evaluation of the condition is basically a no-op, the first transition was executed relatively quickly, peaking at a total 7 410 ms execution time for 10 000 actors. Following transitions in which more complex conditions had to be checked per-actor, such as a proximity-check for finding and approaching other actors, reached higher runtime for larger numbers of actors on average.
Figure 2.13 shows the plan of an exemplary execution when running `tick()` for 10 000 actors on the DFA of the running example after a warmup phase. The total execution time of the entire plan is at around 35 seconds (line 31), of which the majority is spent on generating the CTE `updates` which contains the new states for all actors and executes all updating side effects (line 2). Costs for evaluating conditions for edge traversal (line 24) and dispatching effects (line 5) ended up at around 1 ms per actor each. Combined with the measurements in Figure 2.12, linear runtime in dependence on the number of actors stands to reason. While these times sum up to unacceptable waiting times for a tick in a live game, the presented implementation does not make use of possible optimisations: as actors are updated independently from each other and based on the global state of the former iteration, the computations could be done in parallel. As the parallelisation mechanisms in the target version of PostgreSQL are rather rudimentary at the time of writing, this has not been explored further but leaves ample opportunity for improvements when running on a later version of PostgreSQL or any other DBMS that makes proper use of parallel computation. For the case of online games, where a dedicated server usually does the heavy lifting instead of the user's machine, *horizontal sharding*, where the tables are distributed onto multiple physical DBMS-servers, lends itself to this type of application.

In its current state, the approach turned out viable for driving a few hundred actors at once, but further improvements are required for the desired scale of large online games, in which the number of active objects could increase by several orders of magnitude. One possible extension to improve the overall performance is to group actors into urgency tiers. For example, when a player leaves an area, the actors in that section might not require the same update frequency to patrol the area and do idle tasks as when the player is near to actually notice their behaviour. This improvement was not explored in the scope of this thesis and could be subject to future work.

Actors	Min (ms)	Mean (ms)	Median (ms)	Max (ms)
100	29	101	104	162
1 000	264	1 478	1 445	8 303
10 000	7 409	31 714	21 135	769 490

FIGURE 2.12: Minimum, maximum, mean, and median runtime for a single update, determined by executing 100 consecutive updates on the DFA from Figure 2.2 for 100, 1 000, and 10 000 actors.

```
Update on dfa.actor_states current (cost=3777369.52..3777707.06 rows=10001 width=50)
                                   (actual time=35630.142..35630.201 rows=0 loops=1)
 CTE updates
  -> Nested Loop (cost=377.10..3776902.50 rows=10001 width=12)
                 (actual time=338.053..34949.837 rows=10001 loops=1)
     Output: states.actor_id, e.next_state, CASE ef.fname WHEN [...] ← effect dispatching
     -> Index Only Scan using actors_pkey on dfa.actors a (cost=0.29..587.50 rows=10001 width=4)
                                                          (actual time=0.017..39.711 rows=10001 loops=1)
        Output: a.actor_id
        Heap Fetches: 1226
     -> Limit (cost=376.81..376.82 rows=1 width=44)
              (actual time=1.272..1.274 rows=1 loops=10001)
        Output: states.actor_id, ef.fname, e.next_state, e.weight
        -> Sort (cost=376.81..376.82 rows=1 width=44)
                (actual time=1.267..1.267 rows=1 loops=10001)
           Output: states.actor_id, ef.fname, e.next_state, e.weight
           Sort Key: e.weight DESC
           Sort Method: quicksort Memory: 25kB
           -> Nested Loop Left Join (cost=1.46..376.80 rows=1 width=44)
                                    (actual time=0.760..1.254 rows=2 loops=10001)
              Output: states.actor_id, ef.fname, e.next_state, e.weight
              Inner Unique: true
              -> Nested Loop (cost=1.31..376.62 rows=1 width=16)
                             (actual time=0.746..1.225 rows=2 loops=10001)
                 Output: states.actor_id, e.next_state, e.weight, e.effect_id
                 Inner Unique: true
                 Join Filter: CASE c.fname [...] ← condition dispatching
                 [...]
[...]
Query Text: SELECT dfa.tick();
Result (cost=0.00..0.26 rows=1 width=4) (actual time=35635.502..35635.509 rows=1 loops=1)
```

FIGURE 2.13: Query plan for a typical run of `tick()` on the DFA from Figure 2.2 for 10 000 actors at once. Notable passages that caused the lion's share of the runtime are highlighted.

Chapter 3

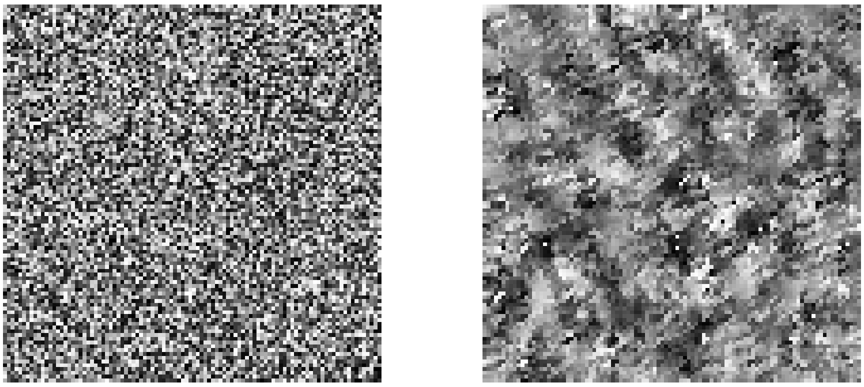
Map Generation

Modern video games usually[1] require a field on which they are being played. Akin to their analogous predecessors, playing fields in early video games would be called *boards*. Several terms have emerged over the years, such as *map*, *level*, and *stage*. Some of these terms are overloaded, as they may not only refer to the terrain, but also to the mission players need to accomplish, or actors and items that can be found during the play. Throughout this chapter, all the aforementioned terms will be used to refer to the terrain on which a game or part of a game takes place.
Designing stages for a game requires meticulous understanding of the game at hand. While some video games, such as early versions of *Asteroids* or *Raiden* (see Figure 3.3), conveyed their game mechanics purely through the actors on screen (i.e. the map could be removed without making the game unplayable), for many other games the map is a vital part of making the game playable and enjoyable. Real time strategy games call for large open spaces, in which units can be manoeuvred strategically (see Figure 3.4). Dungeon-crawling and some adventure games tend to comprise their maps of multiple rooms, that are connected through doors and passageways (see Figure 3.5). Games either come with a fixed set of maps, that are usually hand-crafted, or with tools to create arbitrary new maps. This can include map editors for end-users or generators that create maps based on certain rules. While Togelius et al. argue that allowing the end-users to fabricate content themselves is an enjoyable part of the gaming experience itself, they also raise points as to why having the game developer provide the content, or at least the rules for the content, can be preferable at times [90]. This chapter will therefore focus on how developers can be supplied with means to automatically generate spatial content for their games in the relational world.

Established approaches can operate using only a handful of parameters, like a seed for the random generator, to generate spatial information. *Perlin Noise* is a well-known algorithm for generating randomised terrain, originally devised to generate graphics for movies. To not just end up with complete randomness, a grid of random vectors is generated, which is then used to interpolate the values between the grid points [57, 58]. Depending on the smoothing functions and interpolation, Perlin Noise can yield quite diverse structures, such as textures, organic-seeming surfaces and topographical maps, as seen in Figure 3.1, which is for example used in *Minecraft* [52, 59]. While flexible and easy to implement in SQL[2], its outcome is still largely depended on the implementation details and used smoothing functions, which is, again, not always accessible to the people responsible for creating the content, as described in Section 1.1.5. Again, assuming a data-centric point of view on this problem facilitates the usage of database systems in this aspect of video games. Instead of procedurally placing elements, such as a piece of floor or a piece of wall, on a grid to fabricate a map (map editor), we can view these elements as data and their positioning as their relation to one another.

[1] There are exception to this, such as purely text-based educational games.

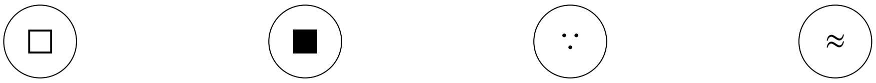
[2] Implementation attached in Appendix B, which was used to generate the data for Figure 3.1(b).



(a) Random noise.

(b) Perlin noise.

FIGURE 3.1: Perlin noise can be used as a technique to generate randomised terrain that is smoother than just random noise and therefore lends itself to creating topographical maps.



(a) Walkable terrain

(b) Walls

(c) Coast

(d) Water

FIGURE 3.2: Tiles used in the following examples.

The following section presents two such approaches. In both cases maps will be thought of as two-dimensional grids of arbitrary granularity. That means, parts of the map could be as coarse as a whole room or as small as a single pixel in the final rendering. Throughout all examples, a small set of *tiles*, which represent a certain terrain type each, is used to build the map from, which can be seen in Figure 3.2. Both algorithms make use of *modules*, which are compounds of size 3×3, built from the aforementioned tiles. While other dimensions are feasible, 3×3 has turned out to be convenient and suited for demonstration purposes. Although all examples are kept in two dimensions for simplicity, both approaches can easily be modified to work in three-dimensional games.

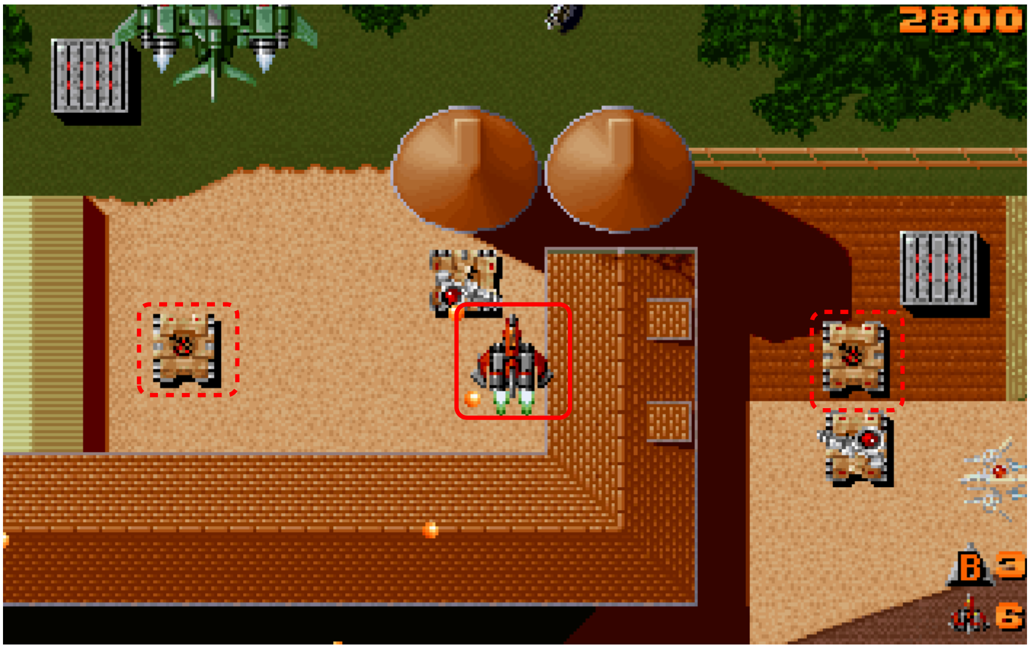


FIGURE 3.3: The 1990 game Raiden by Seibu Kaihatsu. While thematically fitting, the map purely serves as a background for aesthetic reasons. Game mechanics are conveyed through the actors. The player character has been outlined with a solid rectangle, two enemies are marked with dashed rectangles.

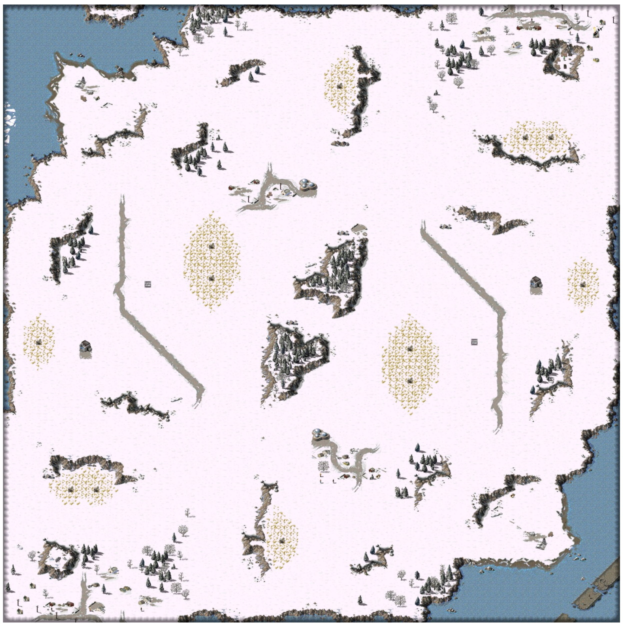


FIGURE 3.4: Overview of a map for a real time strategy game, taken from the open source engine OpenRA. Grey areas are walls, the white space is walkable snow. The open structure caters to the nature of the game.

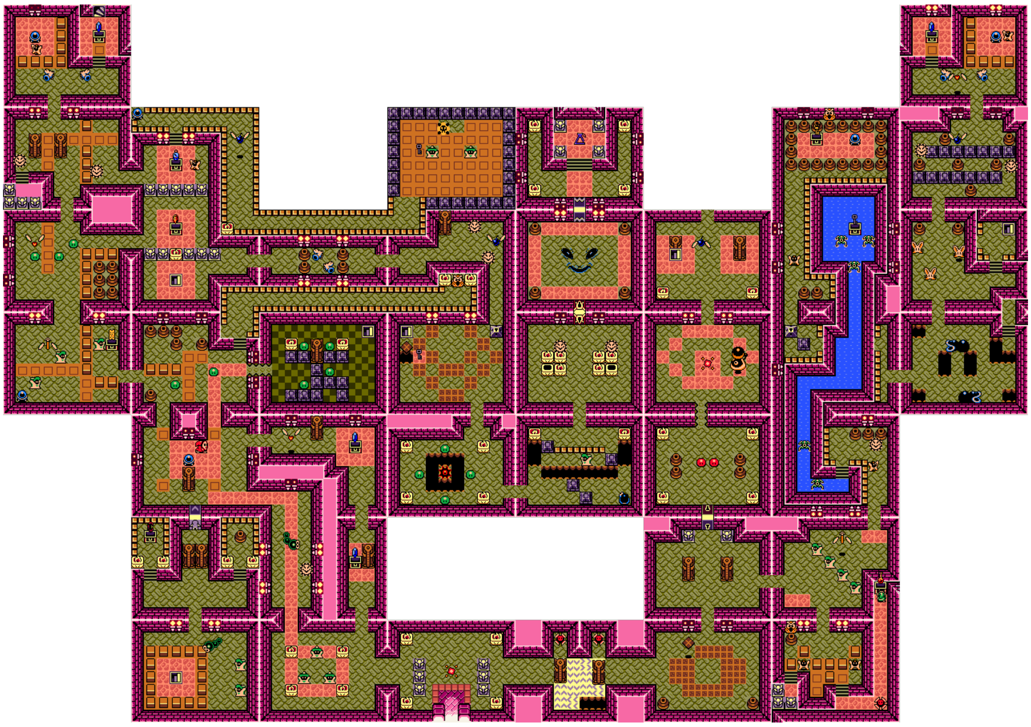


FIGURE 3.5: Map overview of the *Face Shrine* in Nintendo's *The Legend of Zelda: Link's Awakening*. The dungeon is divided into small rooms of which the player only sees one at a time. (Image retrieved from https://zelda.fandom.com/wiki/Face_Shrine_(dungeon) on 29.5.2019).

3.1 Rule-Based Map Generation

The first explored algorithm uses a recursive application of a *two-dimensional grammar* to a starting point, to expand each singular tile from earlier steps into a full module. Grammars with two dimension are already being used in image recognition and decomposition known as *picture languages* [73, Chapter 4 ff.][69]. The approach is also related to the idea of *shape grammars* presented by Stiny and Gips [88], as it repeatedly updates the structure of the generated grid with finer detail. Two-dimensional rules are being used in the field of *cellular automata* as well, albeit with inverse mapping, as rules do not unfold an element into its neighbours, but rather collapse the state of an element's neighbourhood to produce a new state for the element at hand [89, Chapter 5]. The two-dimensional grammars used in this approach yield fractal-like shapes, resembling interconnected rooms of varying size that are partitioned into smaller rooms themselves. Grammars have historically been used in video games to not only define the map, but also a series of objectives the player needs to achieve to advance through the game. Shaker et al. cover this secondary use of grammars for content generation in [77]. The following section focuses on grammars as a vehicle for map generation, although the way the grammar is represented and applied in relational form can also be used for other applications.

A two-dimensional grammar G^R is a tuple (Σ, V, δ, v_0), where

1. Σ is a set of terminal symbols (tiles).
2. V is a set of non-terminals.
3. δ is a set of rules mapping a non-terminal onto a module consisting of terminals or non-terminals.[3]
4. $v_0 \in V$ is the initial non-terminal the map starts out from.

An example for such a grammar is given in Figure 3.6, which is used throughout this section. Figure 3.8 gives an idea of how that grammar generates structures similar to the dungeon shown in Figure 3.5. The relational representation of the grammar is given in Figure 3.9.

$$\begin{aligned}
\Sigma &= (\blacksquare, \square)\\
V &= (A, B)\\
\delta &= \left(A \mapsto \begin{array}{c} BBB\\ BBB\\ BBB \end{array},\ B \mapsto \begin{array}{c} \blacksquare\blacksquare\blacksquare\\ \blacksquare\square\blacksquare\\ \blacksquare\blacksquare\blacksquare \end{array},\ B \mapsto \begin{array}{c} \blacksquare\square\blacksquare\\ \square B\square\\ \blacksquare\square\blacksquare \end{array} \right)\\
v_0 &= A
\end{aligned}$$

FIGURE 3.6: Simple recursive grammar G_1^R.

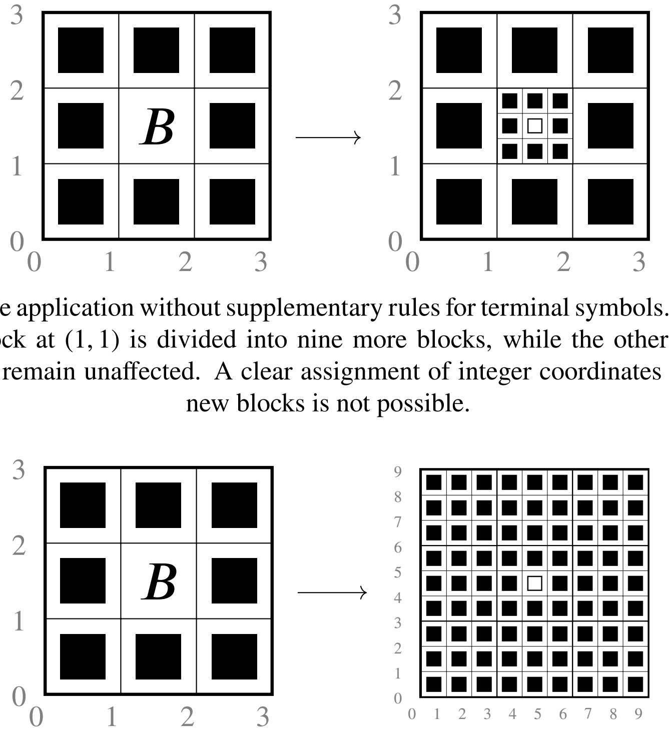
Starting with v_0 and then repeatedly applying the rules in δ to the existing tiles produces a map of increasingly finer granularity, as all tiles unfold with each iteration. However, this technique presents us with the issue of mapping a block with one coordinate in the two-dimensional space to nine blocks with one coordinate each when applying a rule. This problem especially occurs if for some blocks no rule exists to further unfold them, as is notably the case with terminal symbols, causing a *partial unfolding* of the map, which then has "holes" in it. The underlying issue is showcased in Figure 3.7, where rules can only be applied to non-terminals in Figure 3.7(a), resulting in an irregular grid. This issue can be solved by extending δ to δ' with supplementary rules:

$$\delta' = \delta \cup \left\{ \sigma \mapsto \begin{array}{c} \sigma\sigma\sigma\\ \sigma\sigma\sigma\\ \sigma\sigma\sigma \end{array} \mid \forall_{\sigma \in \Sigma} \right\}$$

These rules effectively scale each terminal up as if it was a non-terminal that was being unfolded and makes the assignment of coordinates clear. They do not change the terminal in any other way. The utilisation of δ' is shown in Figure 3.7(b), which enables the usage of a regular grid.

The SQL-implementation is explained in the following section in detail.

[3] If there are multiple applicable rules for a given non-terminal, an arbitrary mapping can be picked (pseudo-randomly or by any other means). In fact, having multiple rules for the same non-terminal makes the resulting map more interesting, as it varies in form.



(a) Rule application without supplementary rules for terminal symbols. Only the block at $(1, 1)$ is divided into nine more blocks, while the other eight blocks remain unaffected. A clear assignment of integer coordinates to the new blocks is not possible.

(b) Rule application with additional rules for terminal symbols. All blocks, including terminals, are expanded in each recursive step, making it easy to assign integer coordinates to each block.

FIGURE 3.7: Snippet of a generated map illustrating of how extending δ to δ' solves the issue of partial unfolding.

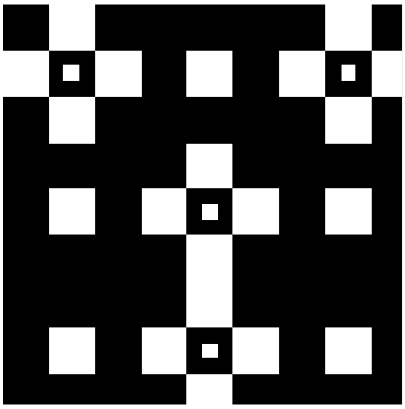


FIGURE 3.8: Exemplary map generated from the grammar in Figure 3.6.

δ

rule	input	x	y	output
1	*B*	0	0	■
1	*B*	0	1	■
1	*B*	0	2	■
1	*B*	1	0	■
1	*B*	1	1	□
1	*B*	1	2	■
1	*B*	2	0	■
1	*B*	2	1	■
1	*B*	2	2	■

δ

rule	input	x	y	output
2	*B*	0	0	■
2	*B*	0	1	□
2	*B*	0	2	■
2	*B*	1	0	□
2	*B*	1	1	*B*
2	*B*	1	2	□
2	*B*	2	0	■
2	*B*	2	1	□
2	*B*	2	2	■

δ

rule	input	x	y	output
3	*A*	0	0	*B*
3	*A*	0	1	*B*
3	*A*	0	2	*B*
3	*A*	1	0	*B*
3	*A*	1	1	*B*
3	*A*	1	2	*B*
3	*A*	2	0	*B*
3	*A*	2	1	*B*
3	*A*	2	2	*B*

δ

rule	input	x	y	output
4	□	0	0	□
4	□	0	1	□
4	□	0	2	□
4	□	1	0	□
4	□	1	1	□
4	□	1	2	□
4	□	2	0	□
4	□	2	1	□
4	□	2	2	□

δ

rule	input	x	y	output
5	■	0	0	■
5	■	0	1	■
5	■	0	2	■
5	■	1	0	■
5	■	1	1	■
5	■	1	2	■
5	■	2	0	■
5	■	2	1	■
5	■	2	2	■

ΣV

symbol	terminal
■	`true`
□	`true`
A	`false`
B	`false`

FIGURE 3.9: Grammar from Figure 3.6 in relational form. For viewing convenience the δ-table has been visually divided into multiple tables, each holding one rule. Rules 4 and 5 are the result of adding the scaling rules for the two terminal symbols. The contents of Σ and V are condensed into a single table ΣV for implementation reasons. An additional boolean flag distinguishes between terminals and non-terminals.

As preliminary step, δ is expanded to δ' by creating a supplementary rule for each terminal.

```
CREATE FUNCTION create_map(_υ0 TEXT, _max_generations INT)
RETURNS TABLE(x INT, y INT, output TEXT) AS $$
WITH RECURSIVE
δ'(rule, input, x, y, output, terminated) AS (
    SELECT
        rule   AS rule,
        input  AS input,
        x      AS x,
        y      AS y,
        output AS output,
        FALSE  AS terminated
    FROM
        δ
    UNION ALL
    (
        WITH
        offsets(x,y) AS (
            SELECT
                x,y
            FROM
                generate_series(0,2) AS x,
                generate_series(0,2) AS y
        ),
        -- give each terminal a single rule number,
        -- based on the current max rule number
        rule_numbers(n, symbol) AS (
            SELECT
                (SELECT MAX(rule) FROM δ) + ROW_NUMBER() OVER (),
                symbol
            FROM
                ΣV
            WHERE
                ΣV.terminal
            GROUP BY
                symbol
        )
        SELECT
            rn.n      AS rule,
            rn.symbol AS input,
            offsets.x AS x,
            offsets.y AS y,
            rn.symbol AS output,
            TRUE      AS terminated
        FROM
            rule_numbers AS rn
            CROSS JOIN offsets
    )
),
...
```

In this approach, a map generation is done by repeatedly joining the already generated portion of the map with δ', which produces an inward growing map. Each step unfolds all tiles simultaneously, making the algorithm perfect fit to run on a database system, which excel at putting large data sets in relation with each other (here: the map to unfold and the rule set). Blocks are therefore treated as possible input for all rules, which produces a set of possible rules for each block. Selecting a random matching rule per block using `DISTINCT ON`[4] then generates the relation `applicable`.

[4] `SELECT DISTINCT ON (x,y)` is a PostgreSQL specific extension of the SQL standard to force a distinction of the rows on `x` and `y`. Duplicates are handled by selecting the row that comes first in the defined ordering. This can be implemented more verbosely in standard SQL, which is omitted for brevity.

```
-- all rules, but grouped to one row per rule number
rules(rule, input, terminated) AS (
    SELECT DISTINCT ON (rule)
        rule,
        input,
        terminated
    FROM
        δ'
),
expand↻(generation, rule, input, output, x, y, terminated) AS (
   (VALUES (1, -1, '', _v0, 0, 0, FALSE))
   UNION ALL
   (
      WITH
      applicable(generation, input, x, y, rule) AS (
          SELECT DISTINCT ON (ex.x, ex.y)
              ex.generation + 1 AS generation,
              ex.output         AS input,
              ex.x              AS x,
              ex.y              AS y,
              r.rule            AS rule
          FROM
              expand↻ AS ex
              JOIN rules AS r
                ON ex.output = r.input
          ORDER BY
              ex.x, ex.y, random()
      ),
⋯
```

Finally, the selected rules are applied to the blocks, creating a new generation of blocks. Each original block produces nine new blocks to which the matching new coordinates are then assigned. The generation process ends once the map does not contain any more non-terminals. Additionally, a parameter `_max_generations` can be used to limit how many times the whole map is expanded, constraining the map to at most $9^{(\_max\_generations-1)}$ blocks.

```
      unfolded(generation, rule, input, output, x, y, terminated) AS (
        SELECT
            a.generation   AS generation,
            a.rule         AS rule,
            a.input        AS input,
            δ'.output      AS output,
            3 * a.x + δ'.x AS x,
            3 * a.y + δ'.y AS y,
            δ'.terminated  AS terminated
        FROM
            applicable AS a
            JOIN δ'
              ON a.rule = δ'.rule
      )
      SELECT
          unfolded.*
      FROM
          unfolded,
          (SELECT COUNT(*) FROM unfolded WHERE NOT terminated) applications(c)
      WHERE
          applications.c > 0
          AND generation <= _max_generations
   )
)
⋯
```

Note how the semantics of `WITH RECURSIVE` leave us with the intermediate results of all recursion steps as final result. That is, the initial v_0 is present in the result of the query, as well the blocks it has been expanded to on the same coordinates, and so on. We therefore finalise the result by applying `DISTINCT ON` to it, selecting only the tile from the youngest generation for each distinct coordinate.

```
SELECT DISTINCT ON (x,y)
  x, y, output
FROM
  expand
ORDER BY
  x, y, generation DESC
$$ LANGUAGE sql;
```

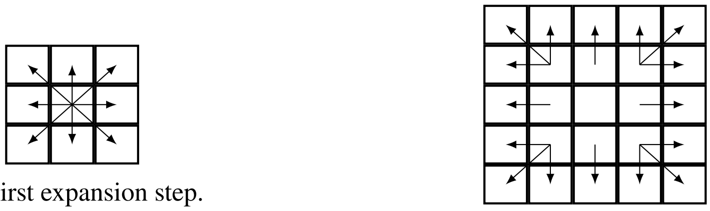
While this approach produces some interesting maps, their general shape of nested rooms and secluded areas suggest they are more apt to generate maps for adventure games, rather than for general games, as mentioned earlier. This observation also matches the findings of Green et al. [35]. On top of that, writing these kinds of rules for large boards that should have some sort of aesthetic to them has proven to be tedious. Maps generated this way could still be used in combination with the approach described in the following section to serve as a general layout that is then filled with more content. Generating maps in multiple passes where the rough outline is embellished further down the pipeline is not at all an unorthodox idea, as [27] and [35] suggest.

3.2 Module-Based Map Generation

The unwieldiness of the rule-based approach led to the exploration of another technique. This approach is based on the idea of having several small predefined map pieces (modules) which are stitched together in a modular fashion based on compatibility rules. This facilitates a data-centric view on map generation, where the user controls the output of the algorithm by providing appropriate pieces, while the algorithm itself stays the same at core. Modules can be grouped into multiple *modulesets* to create maps obeying different rule sets, like having maps following an arctic or a tropical theme. The required user input is a tuple $G^M = (\mathcal{T}, \mathcal{M}, C, \mathcal{S})$ consisting of

1. an alphabet of tiles $\mathcal{T}$;
2. a set of modules $\mathcal{M}$, where each module is a grid of 3×3 tiles $\in \mathcal{T}$. Each tile within a module is uniquely identified by a coordinate $(x, y) | x, y \in \{0, 1, 2\}$ specific to the module they are in;
3. C being a set of ordered triples $\{(x, y, f) | x, y \in \mathcal{T}, f \in \mathbb{N}^+\}$, describing the *compatibility* of x and y. Two tiles x and y are said to be compatible when y is allowed to be placed next to x on the map. f denotes the *frequency* of this compatibility, which is explained below;
4. an initial module $\mathcal{S} \in \mathcal{M}$.

This approach then produces maps based on predefined modules to grow the map outwards by repeatedly joining modules to the outsides of the map, as is shown in Figure 3.10.

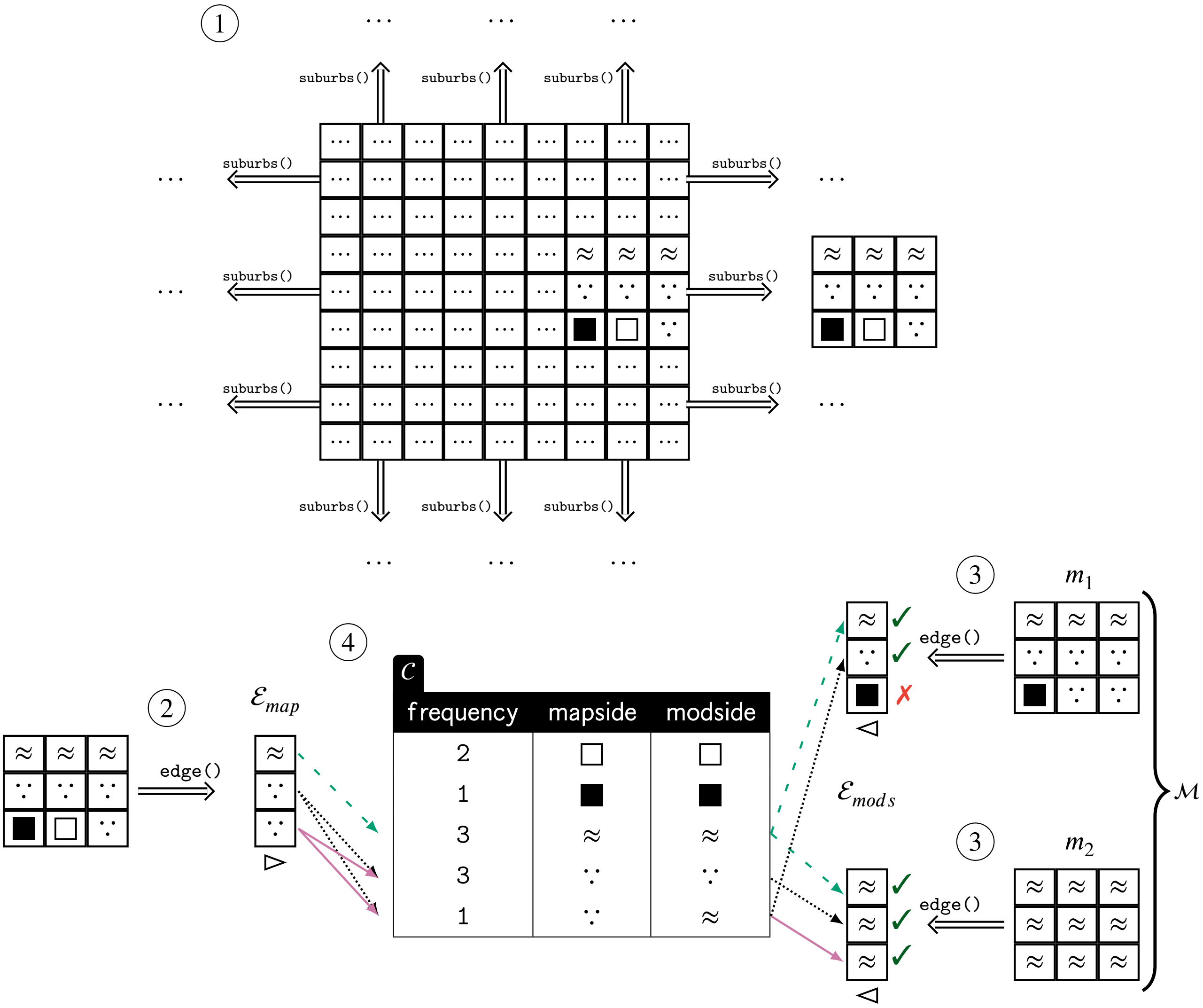


(a) First expansion step.

(b) Second expansion step.

FIGURE 3.10: Two steps of an expansion.

A relational schema as used for the implementation is given in Figure 3.12 in slightly simplified form. The following section explains the algorithm in-depth, accompanied by appropriate SQL snippets. Circled numbers mark regions in the visualisations presented in Figure 3.11.



frequency	mapside	modside
2	□	□
1	■	■
3	≈	≈
3	∵	∵
1	∵	≈

FIGURE 3.11: Illustration of the process of finding a compatible module to extend the map with. Module m_2 is compatible, as all three tiles can find a join partner in $\mathcal{C}$. Module m_1 lacks a join partner for the bottom tile, rendering the whole module incompatible.

modulesets

id	name
1	Tropical
2	Arctic

⋈

$\mathcal{T}$

id	symbol
1	□
2	■
3	∵
4	≈

⋈

modules

id	moduleset_id
1	1
2	1

⋈

module_contents

module_id	x	y	tile_id
1	0	0	4
1	0	1	3
1	0	2	2
1	1	0	3
1	1	1	3
1	1	2	3
1	2	0	4
1	2	1	3
1	2	2	3
2	0	0	4
2	0	1	4
2	0	2	4
2	1	0	4
2	1	1	4
2	1	2	4
2	2	0	4
2	2	1	4
2	2	2	4

=

$\mathcal{M}$

moduleset_name	module_id	x	y	symbol
Tropical	1	0	0	≈
Tropical	1	0	1	∵
Tropical	1	0	2	■
Tropical	1	1	0	∵
Tropical	1	1	1	∵
Tropical	1	1	2	∵
Tropical	1	2	0	≈
Tropical	1	2	1	∵
Tropical	1	2	2	∵
Tropical	2	0	0	≈
Tropical	2	0	1	≈
Tropical	2	0	2	≈
Tropical	2	1	0	≈
Tropical	2	1	1	≈
Tropical	2	1	2	≈
Tropical	2	2	0	≈
Tropical	2	2	1	≈
Tropical	2	2	2	≈

$\mathcal{C}$

frequency	mapside	modside	moduleset_id
2	□	□	1
1	■	■	1
3	≈	≈	1
3	∵	∵	1
1	∵	≈	1

FIGURE 3.12: Relational representation of G_1^M that is used in the implementation. The seed S is intentionally left out, as it does not need to be specified before the generation process takes place and instead can be freely chosen during runtime. Note that module sets bear no relevance for the implementation beyond filtering modules and rules that belong to a selected module set and are therefore omitted in the following explanation.

3.2.1 Preparations

The map generation can be executed during runtime either before the player dives into the game or even to extend the map during playing. The latter is done in games with seemingly infinite worlds, as is the case in Minecraft. Parts of these computations can be done offline and result in a trade-off between runtime computation and memory consumption, as the precomputed results are explicitly stored in relational form. This is actually an advantage in this case, as processing large amounts of data is a strong forte of database systems and presents us with an opportunity to create indices and auxiliary structures. The relations in this subsection are nonetheless presented as CTEs for simplicity reasons.

The preparations start with precomputing the edges of all modules, called *module-edges* subsequently. That is, the adjacent tiles along all four sides of all modules are combined into an edge together with the direction they are facing. This is done by grouping all tiles together that share a certain coordinate within the module. For example, all tiles that have the x-coordinate of 0 within a module m are part of the left module-edge of m, which will assign them ◁ as `junction`. For each edge we also determine `x_off` and `y_off` to denote how the position of a module attached to the edge at hand is calculated. If a module is attached to the left edge of the map, its position is the position of the module it was attached to but with the x-coordinate moved to the left by one, which is conveyed by the `x_off` of `-1` for that particular module-edge. This step only has to be repeated if the underlying data changes, i.e. new modules are created by a game designer or new compatibility rules are defined. `centroid_x` and `centroid_y` denote the coordinate of the centre of an edge. In this context, `any` refers to a special value that is equal to any integer, so basically a “do not care” value. `THE` is an aggregation operator defined by Peyton Jones and Wadler in [45, Section 3.4], which selects an arbitrary element from a list of identical values. We can safely use `THE` in this case, because all attributes selected this way are functionally dependent on the grouping criterion.

```
WITH
module_edges(module_id, offered_junction, required_junction, edge,
             x_off, y_off, centroid_x, centroid_y)
AS (
    WITH module_edge_indicators(required_x, required_y
                               , x_off, y_off
                               , centroid_x, centroid_y
                               , offered_junction, required_junction)
    AS (
        VALUES
        (  0, any,  1,  0, 0, 1, '◁', '▷'),
        (  2, any, -1,  0, 2, 1, '▷', '◁'),
        (any,   0,  0,  1, 1, 0, '△',  '▽'),
        (any,   2,  0, -1, 1, 2, '▽',  '△')
    )
    SELECT
        m.module_id              AS module_id,
        i.offered_junction       AS offered_junction,
        THE(i.required_junction) AS required_junction,
        array_agg(m.symbol ORDER BY m.x, m.y) AS edge,
        THE(i.x_off)             AS x_off,
        THE(i.y_off)             AS y_off,
        THE(i.centroid_x)        AS centroid_x,
        THE(i.centroid_y)        AS centroid_y
    FROM
        𝓜 AS m
        JOIN module_edge_indicators AS i
          ON m.x = i.required_x AND
             m.y = i.required_y
    GROUP BY
        m.module_id, i.offered_junction
),
...
```

To accommodate for having the tiles grouped into triples, we create $C\_edges = \mathcal{T} \times \mathcal{T} \times \mathcal{T}$ to join the edges on later, restricting them to all combinations that actually form an edge.

This leads to an explosion in possible combinations to find matches later on, as can be seen in Figure 3.13. But again, creating these permutations offline is actually desirable as opposed to having to create them during the map generation process.

```
C_edges(mapside, modside, frequencies) AS (
    SELECT
        ARRAY[c1.mapside, c2.mapside, c3.mapside]       AS mapside,
        ARRAY[c1.modside, c2.modside, c3.modside]       AS modside,
        ARRAY[c1.frequency, c2.frequency, c3.frequency] AS frequencies
    FROM
        C AS c1,
        C AS c2,
        C AS c3
        module_edges AS mapside,
        module_edges AS modside
    WHERE
        ARRAY[c1.mapside, c2.mapside, c3.mapside] = mapside.edge
        AND ARRAY[c1.modside, c2.modside, c3.modside] = modside.edge
        AND (c1.relative_x, c1.relative_y) = (c2.relative_x, c2.relative_y)
        AND (c2.relative_x, c2.relative_y) = (c3.relative_x, c3.relative_y)
),
...
```

C_edges

number	mapside	modside	frequencies
1	{□,■,∵}	{□,■,∵}	{2,1,3}
2	{□,■,≈}	{□,■,≈}	{2,1,3}
3	{□,■,∵}	{□,■,≈}	{2,1,1}
4	{□,■,□}	{□,■,□}	{2,1,2}
5	{□,■,■}	{□,■,■}	{2,1,1}
6	{□,■,∵}	{□,■,∵}	{2,1,3}
...	...	...	...
2725	{∵,≈,∵}	{≈,≈,∵}	{1,3,3}
2726	{∵,≈,≈}	{≈,≈,≈}	{1,3,3}
2727	{∵,≈,∵}	{≈,≈,≈}	{1,3,1}
2728	{∵,≈,□}	{≈,≈,□}	{1,3,2}
2729	{∵,≈,■}	{≈,≈,■}	{1,3,1}
2730	{∵,≈,∵}	{≈,≈,∵}	{1,3,3}

FIGURE 3.13: Compatible edges derived from C. The additional column `number` is only included to emphasise the combinatory explosion.

The actual map generation process hinges on a three-way join between the map that has been created up to a certain point, the compatibility table C, and the available modules $\mathcal{M}$ that can be appended to the edges, as is illustrated in Figure 3.11. To speed up the generation process, we can prepare an auxiliary table `semiconnected`, in which $\mathcal{M}$ has already been joined on C to reduce the three-way join into a two-way join during runtime. Visually this step subsumes ③ and ④ of Figure 3.11. In addition to these runtime savings we can also create a hash index over this auxiliary table for additional speed gain.

```
semiconnected(module_id, offered_junction, required_junction
              , x_off, y_off
              , offered_edge, required_edge
              , centroid_x, centroid_y, edge_frequencies)
    SELECT
        me.module_id         AS module_id,
        me.offered_junction  AS offered_junction,
        me.required_junction AS required_junction,
        me.x_off             AS x_off,
        me.y_off             AS y_off,
        c.mapside            AS offered_edge,
        c.modside            AS required_edge,
        me.centroid_x        AS centroid_x,
        me.centroid_y        AS centroid_y,
        c.frequencies        AS edge_frequencies
    FROM
        module_edges AS me
        JOIN 𝒞_edges AS c
            ON c.modside = me.edge
),
...
```

This concludes the preparations and the actual map generation process can now be started.

3.2.2 During Runtime

The process commences with the seed $\mathcal{S}$, which can either be chosen by the user or randomly selected, indicated by `init.id`, which is not explicitly shown here. The final map will consist of tiles, each being represented by a `symbol` $\in \mathcal{T}$, a coordinate (`block_x`|`block_y`) to identify the 3×3 block they are part of, and another coordinate (x|y) with $\mathtt{x} \in \{0, 1, 2\}$, $\mathtt{y} \in \{0, 1, 2\}$ to show where in that block the tiles are situated.

```
expand↻(block_x, block_y, x, y, symbol) AS (
    (
        SELECT
            0        AS block_x,
            0        AS block_y,
            mc.x     AS x,
            mc.y     AS y,
            t.symbol AS symbol
        FROM
            init
            JOIN module_contents AS mc
              ON mc.module_id = init.id
            JOIN 𝒯 AS t
              ON mc.tile_id = t.id
    )
    UNION ALL
    (
...
```

In each recursive step, the algorithm inspects the modules on the borders of the map created so far, marked with ① in Figure 3.11. These modules are the *suburbs* of that iteration. Their edges facing away from the map created up to this point are the current *map-edges* (② in Figure 3.11). We also assign each such map-edge a `required_junction` to indicate what directional symbol a module-edge should carry to be considered appendable.

```
WITH
bounds(min_x, max_x, min_y, max_y) AS (
    SELECT
        MIN(e.block_x) AS min_x,
        MAX(e.block_x) AS max_x,
        MIN(e.block_y) AS min_y,
        MAX(e.block_y) AS max_y
    FROM
        expand↻ AS e
),
edge_indicators(required_block_x, required_block_y, required_x, required_y, required_junction) AS (
    SELECT E.* FROM bounds AS b, LATERAL (VALUES
        (b.min_x, any,       0, any, '▷'),
        (b.max_x, any,       2, any, '◁'),
        (any,     b.min_y, any,   0, '▽'),
        (any,     b.max_y, any,   2, '△'),
        (b.min_x, b.min_y,   0,   0, '◁'),
        (b.max_x, b.min_y,   2,   0, '▷'),
        (b.min_x, b.max_y,   0,   2, '◁'),
        (b.max_x, b.max_y,   2,   2, '▷')
    ) E(required_block_x, required_block_y, required_x, required_y, required_junction)
),
suburbs(block_x, block_y, required, edge) AS (
    (SELECT
        e.block_x                AS block_x,
        e.block_y                AS block_y,
        THE(i.required_junction) AS required,
        array_agg(e.symbol ORDER BY e.x, e.y) AS edge
    FROM
        expand↻          AS e,
        edge_indicators AS i
    WHERE
        i.required_block_x = e.block_x AND
        i.required_block_y = e.block_y AND
        i.required_x = e.x AND
        i.required_y = e.y
    GROUP BY
        i.symbol, e.block_x, e.block_y
    )
),
...
```

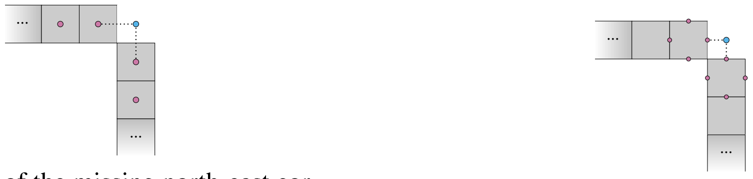
For each map-edge we look for modules that have an opposite-facing module-edge that is compatible with regard to all three tiles. We call such a combination *edge-compatible*. Each edge-compatible module is temporarily placed on the map based on the coordinates of the module it was appended to and the corresponding offset. The maximum desired size, contained in absolute_limits, consisting of min_x, max_x, min_y, and max_y, specified by the user, is accounted for, in case a rectangular map is being generated, where we need more horizontal than vertical expansion steps, or vice versa. So far, we are only performing the horizontal and vertical (straight) expansion. Should multiple matching modules be found, a semi-random ordering selects one match for each coordinate using DISTINCT ON.

```
matches(module_id, block_x, block_y, edge_frequencies) AS (
    SELECT DISTINCT ON (new_block_x, new_block_y)
        semi.module_id           AS module_id,
        sub.block_x + semi.x_off AS new_block_x,
        sub.block_y + semi.y_off AS new_block_y,
        semi.edge_frequencies    AS edge_frequencies
    FROM
        absolute_limits AS l,
        suburbs AS sub
        JOIN semiconnected AS semi
            ON semi.edge = sub.edge AND semi.junction = sub.required_junction
    WHERE
        (sub.block_x + semi.x_off BETWEEN l.min_x AND l.max_x) AND
        (sub.block_y + semi.y_off BETWEEN l.min_y AND l.max_y)
    ORDER BY
        new_block_x, new_block_y, weighted_sort(edge_frequencies) DESC
)
SELECT
    mat.block_x AS block_x,
    mat.block_y AS block_y,
    mc.x        AS x,
    mc.y        AS y,
    mc.symbol   AS symbol
FROM
    matches AS mat
    JOIN module_contents AS mc
      ON mc.module_id = mat.module_id
),
straight_expansion(module_id, block_x, block_y, x, y, symbol) AS (
  SELECT
      mat.module_id AS module_id,
      mat.block_x   AS block_x,
      mat.block_y   AS block_y,
      mc.x          AS x,
      mc.y          AS y,
      mc.symbol     AS symbol
  FROM
      matches AS mat
      JOIN module_contents AS mc
        ON mc.module_id = mat.module_id
  ORDER BY mat.block_x, mat.block_y, weighted_sort(edge_frequencies) DESC
),
...
```

The ordering is influenced by the *frequency* of all tiles contained in the edge in question. The higher their collective frequency, the more likely is it for the edge they make up to be sorted as the first item and subsequently to be selected as distinct element.

```
-- orders a value randomly but with respect to its accumulated probability values.
CREATE FUNCTION weighted_sort(ps INT[])
RETURNS DOUBLE PRECISION AS $$
  SELECT SUM(RANDOM() * p) FROM unnest(ps) AS ups(p);
$$ LANGUAGE SQL VOLATILE;
```

Based on the straight expansion, we can now fill in the diagonal gaps on the corners of the map. This step ensures each iteration to create a proper rectangular map that can be built on in subsequent steps. Not doing this diagonal fixup would lead to maps of the shape of a plus. To find modules for the diagonal fixup, we first determine the blocks from the straight expansion that are adjacent to any of the corner blocks. We can find those blocks by determining the closest neighbours for the corner position based on the Euclidean distance of their centroid from the corner's centroid, visualised in Figure 3.14(a). From each of those blocks, we extract the edge facing the corner in question by selecting the edge with the centroid closest to the corner's centroid, as shown in Figure 3.14(b).



(a) Neighbours of the missing north-east corner C_{NE} can be determined by looking for the module whose centroid (●) is closest to the centroid of C_{NE} (●).

(b) The edges facing C_{NE} can then be found by looking for edges whose centroids (●) are closest to the centroid of C_{NE} (●).

FIGURE 3.14: Visualisation of centroids for modules and borders to determine relevant neighbouring edges for the corner modules. The closest centroids are marked with dotted lines in both cases.

```
diagonal_expansion(module_id, block_x, block_y, x, y, symbol) AS (
    WITH
    corner_coordinates(block_x, block_y) AS (
        SELECT
            xs.x,
            ys.y
        FROM
            (SELECT unnest(ARRAY[min_x-1,max_x+1]) FROM bounds) AS xs(x),
            -- compensate for straight expansion
            (SELECT unnest(ARRAY[min_y-1,max_y+1]) FROM bounds) AS ys(y)
    ),
    corner_neighbours(module_id, corner_x, corner_y, neighbour_x, neighbour_y, c) AS (
        SELECT
            THE(se.module_id),
            cc.block_x,
            cc.block_y,
            se.block_x,
            se.block_y, ABS(cc.block_x - se.block_x) + ABS(cc.block_y - se.block_y)
        FROM
            straight_expansion AS se,
            corner_coordinates AS cc
        WHERE
            ABS(cc.block_x - se.block_x) + ABS(cc.block_y - se.block_y) = 1
        GROUP BY
            cc.block_x,
            cc.block_y,
            se.block_x,
            se.block_y
    ),
    corner_neighbour_edges(module_id, offered_junction, required_junction, edge
                          , corner_x, corner_y
                          , neighbour_x, neighbour_y) AS (
        WITH all_neighbour_edges(module_id
                          , offered_junction, required_junction
                          , edge, corner_x, corner_y
                          , neighbour_x, neighbour_y, distance) AS (
            SELECT
                cn.module_id         AS module_id,
                me.offered_junction  AS offered_junction,
                me.required_junction AS required_junction,
                me.edge              AS edge,
                cn.corner_x          AS corner_x,
                cn.corner_y          AS corner_y,
                cn.neighbour_x       AS neighbour_x,
                cn.neighbour_y       AS neighbour_y,
                euklidian_distance(cn.neighbour_x * 3 + me.centroid_x, cn.neighbour_y * 3 + me.centroid_y,
                                   cn.corner_x * 3 + 1, cn.corner_y * 3 + 1) -- (+1,+1) -> corner centroid
                                     AS distance
            FROM
                corner_neighbours AS cn
                JOIN module_edges AS me
                    ON me.module_id = cn.module_id
        )
        SELECT
            ane.module_id,
            ane.offered_junction,
            ane.required_junction,
            ane.edge,
            ane.corner_x,
            ane.corner_y,
            ane.neighbour_x,
            ane.neighbour_y
        FROM
            all_neighbour_edges AS ane
        WHERE
            ane.distance = 2
    ),
...
```

Based on the edges facing the corners, we can now find candidates for the corner positions through the same selection process we used for the straight expansion. With the slight difference of giving each candidate a rating $\in \{0, 1, 2\}$, reflecting if they are compatible to both their (horizontal and vertical) neighbours, just one of them, or none. By selecting candidates with the highest rating, corner modules will fit perfectly between the already generated neighbours at best, or at least be compatible with one of their neighbours. Providing tilesets with either high or unique compatibility enhances the quality of selected corner modules.

```sql
corner_candidates(module_id, corner_x, corner_y, neighbour_x, neighbour_y, edge_frequencies) AS (
    SELECT DISTINCT ON (cne.corner_x, cne.corner_y, cne.neighbour_x, cne.neighbour_y, semi.module_id)
        semi.module_id,
        cne.corner_x,
        cne.corner_y,
        cne.neighbour_x,
        cne.neighbour_y,
        semi.edge_frequencies,
        ROW_NUMBER() OVER (PARTITION BY cne.corner_x, cne.corner_y, semi.module_id) AS rating -- 1 or 2
    FROM
        corner_neighbour_edges AS cne
        JOIN semiconnected AS semi
          ON cne.required_junction = semi.offered_junction AND cne.edge = semi.offered_edge
),
corner_modules(module_id, corner_x, corner_y) AS (
    SELECT DISTINCT ON (cc.corner_x, cc.corner_y)
        cc.module_id,
        cc.corner_x,
        cc.corner_y
    FROM
        corner_candidates AS cc
    ORDER BY
        cc.corner_x, cc.corner_y, cc.neighbour_x, cc.neighbour_y, cc.rating DESC,
        weighted_sort(cc.edge_frequencies) DESC
),
corner_tiles(module_id, block_x, block_y, x, y, symbol) AS (
    SELECT
        cm.module_id AS module_id,
        cm.corner_x AS block_x,
        cm.corner_y AS block_y,
        mc.x AS x,
        mc.y AS y,
        mc.symbol AS symbol
    FROM
        corner_modules AS cm
        JOIN module_contents AS mc
          ON cm.module_id = mc.module_id
)
SELECT * FROM corner_tiles
```

The expansion step is finalised as `straight_expansion` ∪ `diagonal_expansion`.

This usage of frequencies is particularly useful if certain transitions should occur more or less frequently than others. For example, Figure 3.12 contains the rows (1, ∵, ≈) and (3, ∵, ∵). That makes it far more likely to have edges comprised of ∵ placed next to other such edges than having ≈ next to ∵. Visually, this amounts to having long stretches of coastlines with transitions to water after some blocks. The significance of different frequencies is indicated in Figure 3.15.

As described above, the map is expected to be of rectangular shape. This can only be guaranteed if for each module we can find at least one other module to attach to every edge, or the generation might produce maps with jagged edges where no module could be appended. The database system provides us with this information by looking for modules that do not have at least one open connection in every direction in the `semiconnected` table, allowing the game designers to perform sanity checks on their data.

$\mathcal{C}_2$

frequency	tile	with
1	≈	≈
1	∵	∵
1	∵	≈
1	≈	∵

$$\mathcal{M}_2 = \mathcal{M} \bigcup \left\{ \begin{array}{|c|c|c|}\hline \because & \because & \because \\ \hline \because & \because & \because \\ \hline \because & \because & \because \\ \hline \end{array} \right\}$$

(a) Extending G_1^M from Figure 3.12 to $G_2^M = (\mathcal{T}, \mathcal{M}_2, \mathcal{C}_2, \mathcal{S})$. Note how every row has the same frequency of 1.

≈	≈	≈	∵	∵	∵	≈	≈	≈	∵	∵	∵	≈	≈	≈	
≈	≈	≈	∵	∵	∵	≈	≈	≈	∵	∵	∵	≈	≈	≈	…
≈	≈	≈	∵	∵	∵	≈	≈	≈	∵	∵	∵	≈	≈	≈	

(b) Possible output when using G_2^M. Not utilising the frequency may lead to a repeated alternation between coast and water.

$\mathcal{C}_3$

frequency	tile	with
10	≈	≈
1	∵	∵
1	∵	≈
1	≈	∵

$$\mathcal{M}_3 = \mathcal{M} \bigcup \left\{ \begin{array}{|c|c|c|}\hline \because & \because & \because \\ \hline \because & \because & \because \\ \hline \because & \because & \because \\ \hline \end{array} \right\}$$

(c) Extending G_1^M from Figure 3.12 to $G_3^M = (\mathcal{T}, \mathcal{M}_3, \mathcal{C}_3, \mathcal{S})$.

∵	∵	∵	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	
∵	∵	∵	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	…
∵	∵	∵	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈	

(d) Possible output when using G_3^M. The increased frequency of `10` for connecting ≈ with ≈ leads to a higher probability of having long stretches of water.

FIGURE 3.15: Two examples of the impact of the frequency `frequency`. In both examples, the leftmost module is the seed and only the horizontal expansion to the right side is shown. For reading convenience, a more visual notation for $\mathcal{M}_x$ has been chosen.

```
WITH
mod_dirs(module_id, junction) AS (
    SELECT
        m.id       AS module_id,
        dir.symbol AS junction
    FROM
        𝓜 AS m,
        (VALUES ('◁'),('▷'),('△'),('▽'),('◿'),('◺'),('◹'),('◸'))
          AS dir(symbol)
)
SELECT
    md.module_id,
    md.junction
FROM
    mod_dirs AS md
    LEFT JOIN semiconnected AS sc
      ON md.module_id = sc.module_id AND md.junction = sc.junction
WHERE
    sc.module_id IS NULL
```

It should also be noted that while it may seem restrictive to demand elements of the map to be broken up into normalised chunks (i.e. modules), this approach is actually versatile enough to express structures with dimensions that exceed that of a single module. For example, Figure 3.16 shows a pond structure that does not match the required dimensions of 3 × 3 and thus is split into multiple modules. To avoid the resulting modules from being arranged differently or composed with other modules, artificial tiles are introduced to the junctions between the modules that make up the original structure (Figure 3.16(e)). A compatibility for each pair of neighbouring artificial junction tiles is introduced in $\mathcal{C}$ (Figure 3.16(f)). This forces the generation algorithm to compose bigger structures exactly as they looked before being separated into smaller modules.

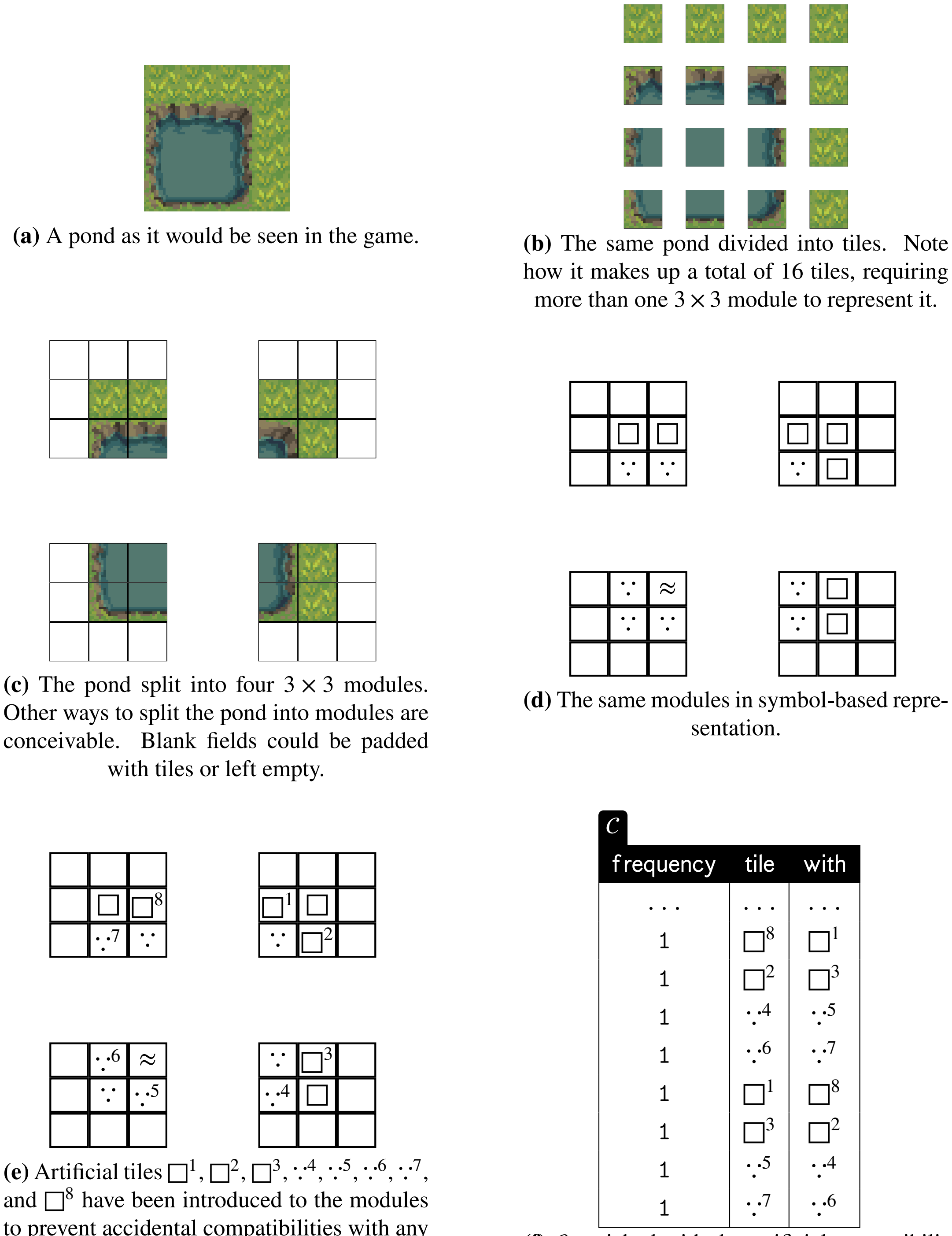
3.2.3 Map Generation by Example

Composing maps from a set of pieces also offers the opportunity to *decompose* maps that have already been crafted, even if they have been created manually. These pieces can in turn be used to generate new maps from it. Game developers can use this approach to provide small, hand-crafted input maps, which can serve as example for large maps, as is shown in Figure 3.17.

We assume the map that serves as input to be available in the following simple schema, where each coordinate of the map is given with a `tile_id` referencing a distinct terrain type. We assume only one map to be present during the process in the following explanation, but multiple maps could be processed at once by introducing a `map_id` by which intermediate results are grouped.

```
CREATE TABLE mapdata(
    id         SERIAL PRIMARY KEY,
    x          INT,
    y          INT,
    tile_id    INT,
    UNIQUE(x, y)
);
...
```

The tiles can be grouped into modules by mapping their global coordinate on the input map to a module. The mapping is done in line 12 in the following listing.



(a) A pond as it would be seen in the game.

(b) The same pond divided into tiles. Note how it makes up a total of 16 tiles, requiring more than one 3×3 module to represent it.

(c) The pond split into four 3×3 modules. Other ways to split the pond into modules are conceivable. Blank fields could be padded with tiles or left empty.

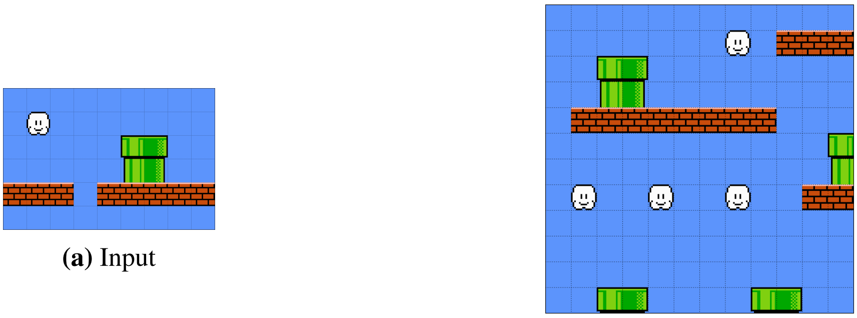
(d) The same modules in symbol-based representation.

(e) Artificial tiles $\square^1$, $\square^2$, $\square^3$, $\because^4$, $\because^5$, $\because^6$, $\because^7$, and $\square^8$ have been introduced to the modules to prevent accidental compatibilities with any other modules.

frequency	tile	with
. . .	. . .	. . .
1	$\square^8$	$\square^1$
1	$\square^2$	$\square^3$
1	$\because^4$	$\because^5$
1	$\because^6$	$\because^7$
1	$\square^1$	$\square^8$
1	$\square^3$	$\square^2$
1	$\because^5$	$\because^4$
1	$\because^7$	$\because^6$

(f) C enriched with the artificial compatibilities, allowing ponds to only be composed as it is seen in Figure 3.16(a).

FIGURE 3.16



(a) Input

(b) Output

FIGURE 3.17: Exemplary usage of map generation by example. A relatively small sample was given as input in Figure 3.17(a) to produce the output seen in Figure 3.17(b). While different dimensions have deliberately been chosen to showcase how a small example map can be used to generate a bigger map with different aspect ratio as well, structures from the input can still be found in the output.

```
9  CREATE VIEW read_modules(module_id, tile_id, module_x, module_y, x, y, global_x, global_y)
10 AS (
11     SELECT
12         ⌊md.x / 3⌋ + ⌈(SELECT MAX(x) + 1 FROM mapdata) / 3⌉ * ⌊md.y / 3⌋
13                       AS module_id,
14         md.tile_id    AS tile_id,
15         md.x / 3      AS module_x,
16         md.y / 3      AS module_y,
17         md.x % 3      AS x,
18         md.y % 3      AS y,
19         md.x          AS global_x,
20         md.y          AS global_y
21     FROM
22         dim,
23         mapdata AS md
24 );
25 ...
```

This process may result in having duplicate modules. Imagine a long stretch of beach that accommodates many tiles of sand, all of them being grouped into 3 × 3 modules, but effectively representing the same piece of a map. Each such module will have the same compatibilities, resulting in a potential explosion of candidates when expanding the map. To stint this explosion, duplicates of modules can be eliminated and one representative module of the group is given a higher frequency during the expansion as compensation. Equal modules can be found by hashing the contents of each module and grouping the modules by hash. In this implementation, the `tile_ids` of each module are concatenated as string, shown in line 34, which enables easy comparison of modules. Note that in line 42 we are using `MAX` in this case instead of `THE` to pick an arbitrary module. Having different identifiers per module renders `module_id` invalid for the use of `THE`, but the modules are the same content-wise, so we can pick an arbitrary module using any available aggregate function that yields a single identifier.

```
26 CREATE VIEW unique_modules(module_id, tile_id,
27                            module_x, module_y, x, y,
28                            global_x, global_y, occurrences)
29 AS (
30 WITH
31     hashed_modules(module_id, hash) AS (
32         SELECT
33             m.module_id AS module_id,
34             string_agg(m.tile_id, '|' ORDER BY m.x,m.y) AS hash
35         FROM
36             read_modules AS m
37         GROUP BY
38             m.module_id
39     ),
40     module_set(module_id, occurrences) AS (
41         SELECT
42             MAX(m.id) AS module_id,
43             COUNT(*)  AS occurrences
44         FROM
45             hashed_modules AS m
46         GROUP BY
47             m.hash
48     )
49     SELECT
50         mod.id         AS module_id,
51         mod.tile_id    AS tile_id,
52         mod.module_x   AS module_x,
53         mod.module_y   AS module_y,
54         mod.x          AS x,
55         mod.y          AS y,
56         mod.global_x   AS global_x,
57         mod.global_y   AS global_y,
58         ms.occurrences AS occurrences
59     FROM
60         read_module_set AS ms
61         JOIN read_modules AS mod
62           ON ms.module_id = mod.id
63 );
64 ...
```

With all the distinct modules at hand, we can again determine compatibilities from how the modules were arranged in the original map. Tiles that fall into adjacent modules and were aligned next to each other in the original map are noted down as compatible, which is shown in the following listing.

```
65 CREATE VIEW read_compatibilities(tile_id1, tile_id2, relative_x, relative_y, frequency) AS (
66     WITH
67     neighbours(tile_id1, tile_id2, relative_x, relative_y) AS (
68         SELECT
69             m1.tile_id                AS tile_id1,
70             m2.tile_id                AS tile_id2,
71             m2.global_x - m1.global_x AS relative_x,
72             m2.global_y - m1.global_y AS relative_y
73         FROM
74             read_modules AS m1,
75             read_modules AS m2
76         WHERE
77             (m2.global_x - m1.global_x, m2.global_y - m1.global_y) IN ((0,1),(1,0))
78     ),
79     compatible(tile_id1, tile_id2, relative_x, relative_y, frequency) AS (
80         SELECT
81             n.tile_id1   AS tile_id1,
82             n.tile_id2   AS tile_id2,
83             n.relative_x AS relative_x,
84             n.relative_y AS relative_y,
85             COUNT(*)     AS frequency
86         FROM
87             neighbours AS n
88         GROUP BY
89             n.tile_id1, n.tile_id2, n.relative_x, n.relative_y
90     ),
91 ...
```

As we only look for neighbours directly to the right and directly below each module, as per the condition in line 77, we only have half the compatibilities. They can easily be made bidirectional:

```
 92         bidirectional(tile_id1, tile_id2, relative_x, relative_y, frequency) AS (
 93             SELECT
 94                 c.tile_id1   AS tile_id1,
 95                 c.tile_id2   AS tile_id2,
 96                 c.relative_x AS relative_x,
 97                 c.relative_y AS relative_y,
 98                 c.frequency  AS frequency
 99             FROM
100                 compatible AS c
101             UNION ALL
102             SELECT
103                 c.tile_id2   AS tile_id1,
104                 c.tile_id1   AS tile_id2,
105                -c.relative_x AS relative_x,
106                -c.relative_y AS relative_y,
107                 c.frequency  AS frequency
108             FROM
109                 compatible AS c
110         )
111         SELECT
112             b.tile_id1       AS tile_id1,
113             b.tile_id2       AS tile_id2,
114             b.relative_x     AS relative_x,
115             b.relative_y     AS relative_y,
116             SUM(b.frequency) AS frequency
117         FROM
118             bidirectional AS b
119         GROUP BY
120             b.tile_id1, b.tile_id2, b.relative_x, b.relative_y
121     );
122     ...
```

The final grouping in line 120 compacts the resulting compatibility table. Say the original determined compatibility consisted of $[(x, y, 2), (y, x, 3)]$, which would produce the bidrectional compatibility $[(x, y, 2), (y, x, 3), (x, y, 3), (y, x, 2)]$, which obviously contains unnecessary duplicates, which are eliminated and condensed into a higher frequency for that compatibility: $[(x, y, 5), (y, x, 5)]$.

3.3 Evaluation

The following sections evaluates both, the rule-based, and the module-based approach in that order.

3.3.1 Rule-Based

Assessing query plans for the rule-based approach showed that a considerable share of the total runtime (around $7 - 11\%$) was spent on determining whether any non-terminals were unfolded during the previous step of the unfolding process (line 97 when selecting from the CTE `unfolded`). This was initially meant as an early exit when a map consisted only of terminals at any given time, but was temporarily disabled for the measurements in Figure 3.19. This change does not introduce any change in functionality that would be relevant for the following experiments, but makes other bottlenecks more obvious.
To conduct the measurements, a grammar consisting of either 100 or 1 000 rules, each rule mapping the non-terminal A onto 3×3 copies of itself, was used to examine the effects of having highly unselective joins as a worst case scenario. A single non-terminal A was then unfolded up to five times by calling the UDF `create_map` with the appropriate parameters. Increasing the number of rules in δ slowed down the unfolding process considerably, as the intermediate table to select the rule from grew accordingly, even prompting the DBMS to resort to external merge sort on disk, as can be seen in Figure 3.18, which explains the surge in time shown in Figure 3.19(a). Note that the exponential growth of the required time does not come as a surprise, as the number of generated tiles grows exponentially as well. It can be approximated as $max(\{\#r \mid (l \mapsto r) \in \delta\})^i$, where i is the number of unfold steps and $\#r$ is

```
CTE applicable
  -> Unique  (cost=926.76..926.83 rows=10 width=60)
             (actual time=8272.219..16576.595 rows=1476 loops=5)
        Output: ((ex.generation + 1)), ex.output, ex.x, ex.y, "δ’".rule_number, (random())
        ->  Sort (cost=926.76..926.78 rows=10 width=60)
                (actual time=8271.067..12751.816 rows=1476200 loops=5)
              Output: ((ex.generation + 1)), ex.output, ex.x, ex.y, "δ′".rule_number, (random())
              Sort Key: ex.x, ex.y, (random())
              Sort Method:  external merge Disk:  269672kB
              ->  Hash Join (cost=873.24..926.59 rows=10 width=60)
                            (actual time=45.864..2997.932 rows=1476200 loops=5)
                    Output: (ex.generation + 1), ex.output, ex.x, ex.y, "δ’".rule_number, random()
                    Hash Cond: ("δ’".input = ex.output)
                    [...]
```

FIGURE 3.18: Part of a query plan for generating a map from a grammar with 100 rules running for 5 iterations. The plan shows part of the subplan used to evaluate the CTE `applicable` shown in Section 3.1.

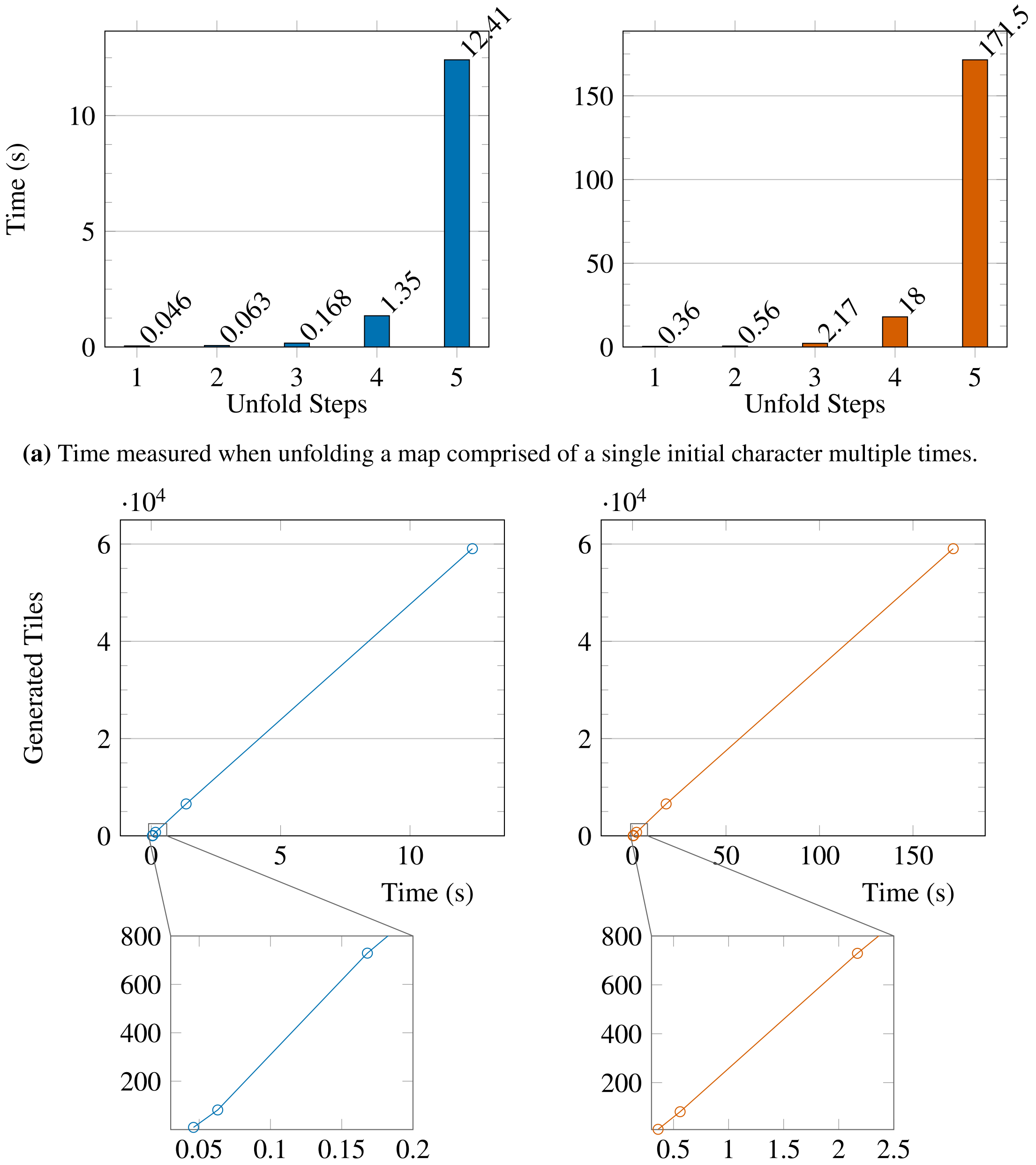
the number of tiles in the right hand side of a rule $l \mapsto r$. As all rules in the experiment map a non-literal onto a 3×3 block, unfolding the initial tile five times produces $9^5 = 59049$ tiles. Since both the number of generated tiles and the runtime grow exponentially, the number of tiles depends linearly on the invested time. This becomes more palpable when depicting the number of generated tiles as a function of the passed time during the generation process in Figure 3.19(b).

The resulting maps did indeed resemble structures that could be used for dungeon generation, but the rule based approach was deemed unfit for excessive use beyond creating the general shape of a map early on in development. Its major disadvantage remains the poor predictability of how new rules affect the generated map. This makes the conception of large rule sets tedious, rendering this approach hardly usable beyond small rule sets for the rough shaping of worlds that then need to be enhanced using other approaches.

3.3.2 Module-Based

To evaluate the modular map generation process, maps of varying size were created using module sets of increasing cardinality. To simulate a worst case scenario, the module sets were comprised entirely of modules of 3×3 walkable tiles (□). Adding $(□, □, x) \in C$ for an arbitrary $x > 0$ guaranteed full compatibility between all modules, making the expansion of each step as expensive as possible. As shown in Figure 3.22, the runtime grew linearly dependent on the module count. Note that increasing the number of possible connectors would have the same effect, as both module count and connector count ultimately influence the relation `semiconnected` on which the intermediate map is joined in each iteration of the generation process. Examining the query plan for generating a 700×700 map from one module reveals a gross underestimation of the number of output rows by the planner, especially when predicting the number of rows added per expansion, as can be seen in Figure 3.23. While this likely contributed to an inefficient execution strategy, a drop in computational speed was to be expected, as the generation hinges on a line of CTEs, where index usage or predicate pushdown becomes harder or even impossible to conduct [67]. In this instance for example, enforcing the usage of an index to perform a hash join through a PostgreSQL switch cut the runtime of the operation seen in Figure 3.23 in half. This sort of manual meddling with the planner, however, is undesirable, as it offloads part of the work the DBMS is supposed to do onto the user.

The proposed map generation can still be used effectively, albeit the seemingly poor performance; large maps are unlikely to be generated on the fly, but rather offline, when the player is either not involved in the process at all (i.e. during compile time), or at the start of a game, during which the user is used to seeing loading screens anyway. Small parts of a map on



(a) Time measured when unfolding a map comprised of a single initial character multiple times.

(b) Number of generated tiles as a function of passed time during the unfolding process.

FIGURE 3.19: Measurements for unfolding a map up to five times using a grammar with 100 (left) and 1 000 rules (right) respectively.

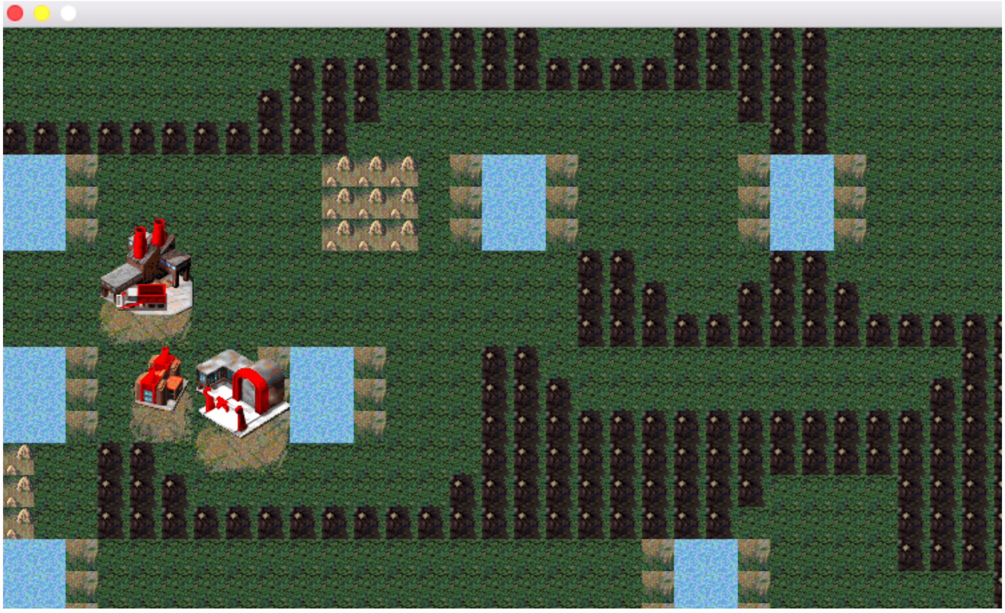


FIGURE 3.20: Screenshot of a generated map inside of OpenRA, showing areas enclosed by mountains and several ponds with a coastline.

the other hand can be generated while the game is taking place. This practice is for example used in Minecraft, where the map is segmented into *chunks* of 16 × 16 × 256 blocks (the 3D equivalent of the 2D tiles used in this chapter). Whenever the player reaches the edge of the map, chunks are generated and appended to the map, as shown in Figure 3.21. Given adequate parameters for the size, chunks can be gradually generated in SQL as well. The main advantage of the module-based approach is the convenience for level designers. The conception of modules can be enhanced with appropriate visual editors, making it much easier for uninitiated users to come up with building blocks for their terrain, instead of grappling with formal definitions of rules or modifications of an algorithm.

The module-based map generation was included into OpenRA to complement the engine's ability to import hand-crafted maps from the map editor that comes bundled with the framework. To avoid drastic changes to any peripheral components, like the way the engine generally stores, loads, and organises maps, generated maps are written to the file format used by OpenRA. This format consists of a `.bin` file, which contains the terrain information for all cells of the map, and a `.yaml` file, containing meta information, like positions players start out on. This process was easily included by adding another button to the map selection interface to trigger the generation process in the database, convert the resulting map information to the expected format and reload all map files from the respective directory. This change amounted to around 300 lines of C# code[5]. An example of a map generated directly from within OpenRA with a modest tileset is shown in Figure 3.20.

[5] Lines of code is obviously a very handwavy metric, especially in the light of C#'s verbosity. This number is therefore only meant to give a general idea of the implementation overhead.

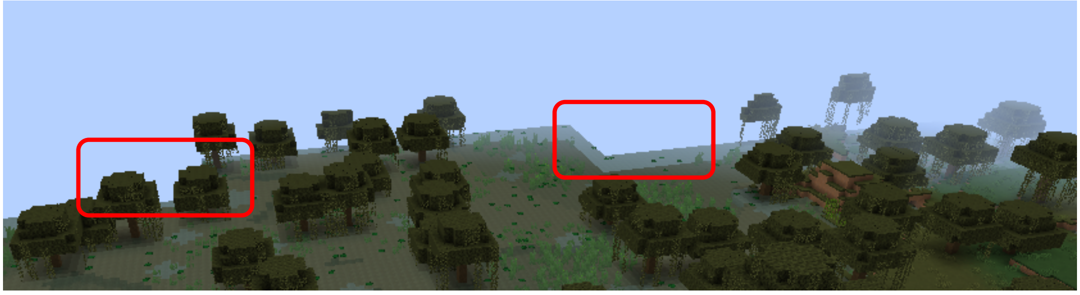


FIGURE 3.21: Screenshot of the edge of a map in Minecraft. Chunks are gradually generated as the player approaches the edge. Two places where chunks are obviously still missing are highlighted.

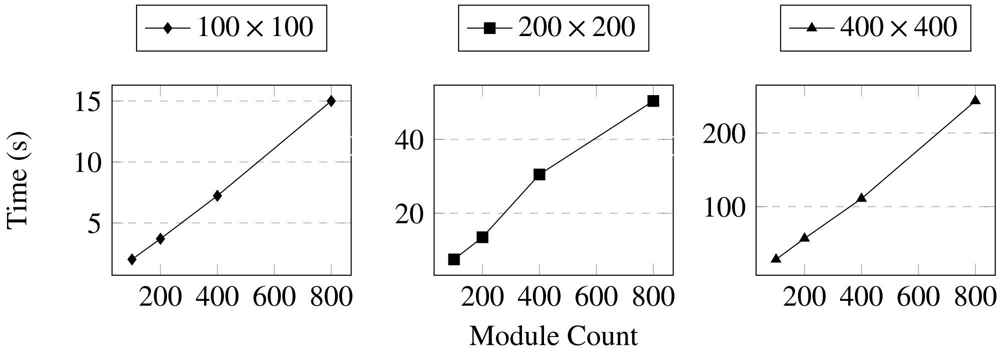


FIGURE 3.22: Measured duration in seconds to generate maps of size 100 × 100, 200 × 200, and 400 × 400 tiles with module sets of 200, 400, 600, and 800 3 × 3 modules each.

```
CTE straight_expansion
  [...]
  ->  Nested Loop (cost=1.30..8.10 rows=6 width=60)
                  (actual time=508.867..1149.138 rows=4141 loops=118)
      Output: mat.module_id, mat.block_x, mat.block_y, mc.x, mc.y,
              mc.symbol, mapgen.weighted_sort(mat.edge_frequencies)
      ->  CTE Scan on matches mat (cost=0.00..0.02 rows=1 width=44)
                                  (actual time=508.549..521.628 rows=460 loops=118)
            Output: mat.module_id, mat.block_x, mat.block_y, mat.edge_frequencies
      ->  Bitmap Heap Scan on mapgen.module_contents mc (cost=1.30..6.52 rows=6 width=44)
                                                        (actual time=0.019..0.043 rows=9 loops=54290)
            Output: mc.module_id, mc.symbol, mc.x, mc.y
            Recheck Cond: (mc.module_id = mat.module_id)
            Heap Blocks: exact=54290
            ->  Bitmap Index Scan on module_contents_module_id_idx (cost=0.00..1.30 rows=6 width=0)
                                                                   (actual time=0.010..0.010 rows=9 loops=54290)
                  Index Cond: (mc.module_id = mat.module_id)
```

FIGURE 3.23: Discrepancy between anticipated and actual number of rows when generating a 700 × 700 map (dimensions chosen arbitrarily) from a single module. The plan shows a single node from the subplan for calculating `straight_expansion` in Section 3.2.2. Around 40% of the total runtime for the entire map generation is spent executing this node.

Chapter 4

Pathfinding

4.1 Path Finding in Video Games

Pathfinding plays a central role in many video games in which actors are at least partially autonomous. It encompasses a family of algorithms which, given a weighted graph, and two points A and B, searches for a sequence of hops to move from A to B. As such, actors that are entirely controlled by an AI need to find their way through the map just like a unit in a strategy game that has been given a destination to reach by the player, making pathfinding indispensable for many games. While Algfoor et al. describe several algorithms that have been used in recent times in [4], they still identify the well-known A* algorithm as one of the most popular and well-researched algorithms for this task, which is hence subject to the investigations in this chapter. A* is a variation of *Dijkstra's pathfinding algorithm* [24, 40], making it an *informed* pathfinding algorithm, as it utilises additional knowledge to avoid exploring the entire graph in a brute-force fashion. Obviously, path finding is tightly coupled with how the map is represented and primed.

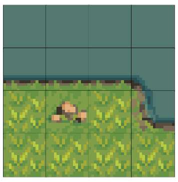
Due to their ability to store information in large quantity, even redundantly if needed, storing the spatial information about the in-game world in a DBMS brings several benefits to the table:

1. Easy implementation of additional constraints to avoid collisions upon planning, which is explored in Section 4.5.
2. Preliminary reduction of the search space.
3. Preparatory analysis of the search space, which are done in Section 4.3.
4. Performing the path search completely within the DBMS close to the data, instead of moving the data to external implementations, as shown in Section 4.4 and Section 4.6.

The following sections explore these advantages and suggest ways of executing path search directly within the DBMS. PostgreSQL can be enhanced using the *pgRouting* extension [62] to offer path finding capabilities through a variety of path finding algorithms, including A*, through calls to UDFs. The pgRouting extension has been used as a baseline in this work, but as its implementation is done in C++, it does not match the ambition to fully realise components of video games in SQL in order to not be tied down to a specific DBMS. Still, it offers a way of keeping the computation of paths entirely in the world of the DBMS and is therefore considered in this chapter. Two additional custom implementations that run entirely within the DBMS have been explored in this work. The required representation for each of them might differ slightly, especially since pgRouting requires weights to be on the edges while the other implementations expect weights to be written on the vertices. But as Skienne notes, vertex-weighted graphs can easily be transformed into edge-weighted graphs [82, p. 209 f.].

The other way around can also be achieved by transforming all vertices of the input graph into edges and vice versa.

The following chapter assumes the map on which the game is played to already be in a relational representation. More specifically, the representation shown in Figure 4.1 is used throughout all examples related to path finding.



(a) Rendered ingame.

	1	2	3	4
1	≈	≈	≈	≈
2	∵	∵	∵	≈
3	□	■	□	∵
4	□	□	□	□

(b) Representation with terrain tiles.

cells

id	x	y
1	1	1
2	1	2
3	1	3
4	1	4
5	2	1
6	2	2
7	2	3
8	2	4
9	3	1
10	3	2
11	3	3
12	3	4
13	4	1
14	4	2
15	4	3
16	4	4

⋈

cell_contents

cell_id	terrain_id
1	3
2	2
3	1
4	1
5	3
6	1
7	4
8	1
9	3
10	2
11	1
12	1
13	3
14	3
15	2
16	1

⋈

terrain_types

id	symbol
1	□
2	∵
3	≈
4	■

=

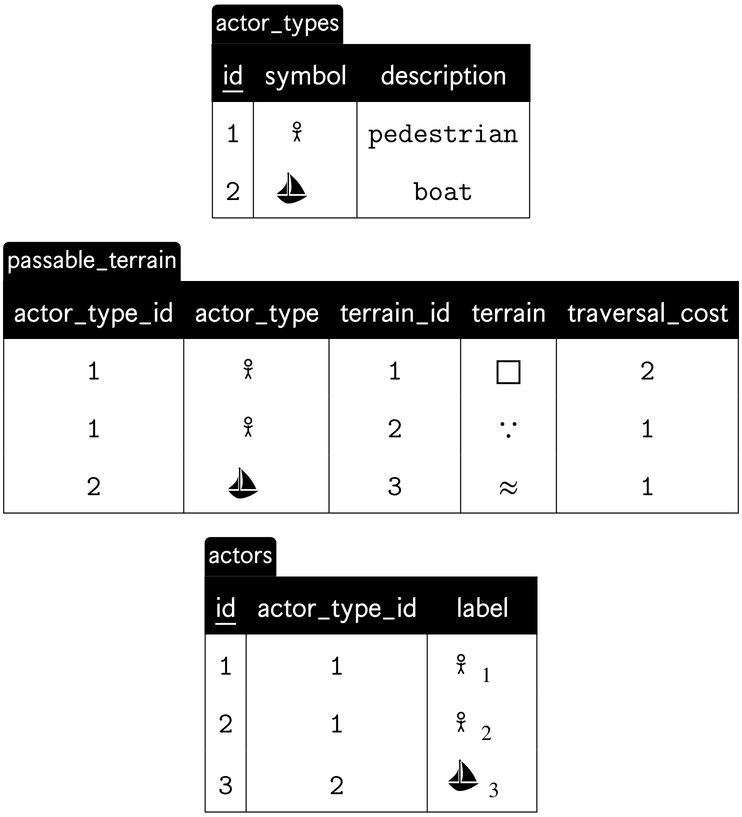
map

cell_id	x	y	terrain
1	1	1	≈
2	1	2	∵
3	1	3	□
4	1	4	□
5	2	1	≈
6	2	2	□
7	2	3	■
8	2	4	□
9	3	1	≈
10	3	2	∵
11	3	3	□
12	3	4	□
13	4	1	≈
14	4	2	≈
15	4	3	∵
16	4	4	□

(c) Relational representation. Note that one cell can actually hold multiple terrain types, hence the separation into multiple tables. This increases flexibility later on when defining what types of terrain actors can tread on.

FIGURE 4.1: Segment of a map in three representations. Figure 4.1(c) is the final representation as assumed in this section.

Additionally, the ability of actors to pass over types of terrain is encoded relationally as well. Actors can usually be grouped into *classes* (or *types*), which encompass certain attributes this group has in common, as is done in OOP. These classes of actors are denoted by specific symbols, such as 🚶 which refers to the whole class of pedestrians. In practice, they could of course be implemented as components, as explained in Chapter 1. Instances of such classes (actors) feature a subscripted ID, such as 🚶$_{42}$ being the pedestrian with unique ID 42. The encoding is exemplified in Figure 4.2.



actor_types

id	symbol	description
1	🚶	pedestrian
2	⛵	boat

passable_terrain

actor_type_id	actor_type	terrain_id	terrain	traversal_cost
1	🚶	1	□	2
1	🚶	2	∵	1
2	⛵	3	≈	1

actors

id	actor_type_id	label
1	1	🚶$_{1}$
2	1	🚶$_{2}$
3	2	⛵$_{3}$

FIGURE 4.2: Relational representation of classes of actors and instances thereof, and information on what types of terrain can be traversed by which types of actors.

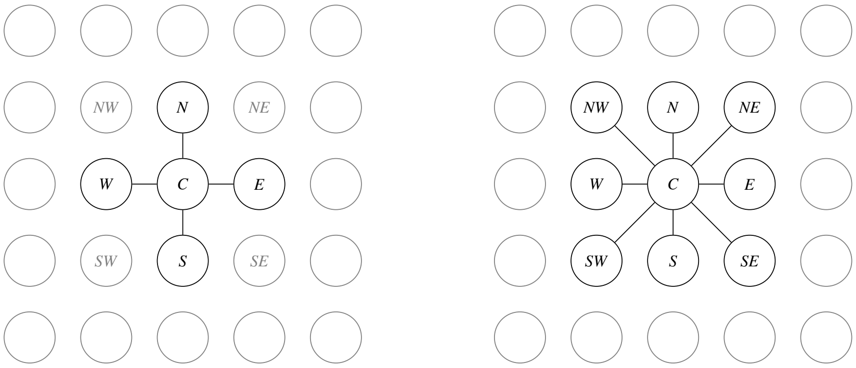
4.1.1 Different Neighbourhoods

When traversing a map, actors go from node to node strictly following through the *neighbourhood* of the node they are currently standing on.[1] A neighbourhood of a node n is the subset of the graph $G = (V, E)$ with $n \in V$ that is directly adjacent to n. While path finding is viable on any kind of graph, we assume maps to generally be decomposable into grids of arbitrary granularity. Neighbourhoods in a grid are either *Von Neumann neighbourhoods* or *Moore neighbourhoods* [48]. In Von Neumann neighbourhoods, each cell has up to[2] four neighbours, being the horizontally and vertically adjacent cells. A Moore neighbourhood additionally consists of up to[2] four diagonal neighbours. The kind of neighbourhood that is being used therefore determines whether actors may move diagonally. Both neighbourhoods are visualised in Figure 4.3. As the used type of neighbourhood largely depends on the game

[1] Games could feature mechanics that allow actors to teleport between nodes of arbitrary distance, but this goes beyond the scope of regular path finding.

[2] Cells on an edge or a corner of a lattice can obviously have fewer neighbours.

in question, a boolean pseudo-function `neighbouring(a, b)` is used in listings in this chapter to signify a check if two coordinates `a=(ax,ay)` and `b=(bx,by)` are neighbours.



(a) Von Neumann neighbourhood. **(b)** Moore neighbourhood.

FIGURE 4.3: Both types of neighbourhoods for the centre node *C* next to each other. Nodes that are part of the neighbourhood of *C* are connected via an edge and outlined in black. All nodes directly adjacent to *C* are labelled by their cardinal direction relative to *C*: north (*N*), north-east (*NE*), east (*E*), south-east (*SE*), south (*S*), south-west (*SW*), west (*W*), and north-west (*NW*).

4.2 Reducing the Search Space

A classic approach to navigating an actor 🯅$_x$ through a map is to start that actor off at a start position *s* and look in the direct neighbourhood of *s* for edges 🯅$_x$ can traverse. In a modest map of 256 × 256 cells, using Von Neumann neighbourhood, we end up with 130 560 bidirectional edges. With no preliminary knowledge, the pathfinding component might need to check the accessibility for 🯅$_x$ of every edge.
Accessibility denotes whether an actor is allowed to pass an edge and can usually be expressed for classes of actors instead of for individuals, as can be seen in Figure 4.2. For example, we can declare that forest terrain is not accessible to (any) boats, and therefore no node consisting of forest is accessible to boats either. Actors can only tread on nodes which hold no terrain they can not traverse. Ergo, we can find the appropriate subgraph for each actor by counting how many terrain types per node he can traverse. If that number coincides with the total number of terrain types on that node, it is traversable for the actor. Costs for traversing a cell can be determined by an arbitrary aggregate function (e.g. `SUM` or `MAX`) over the costs of all terrain types present on that cell.

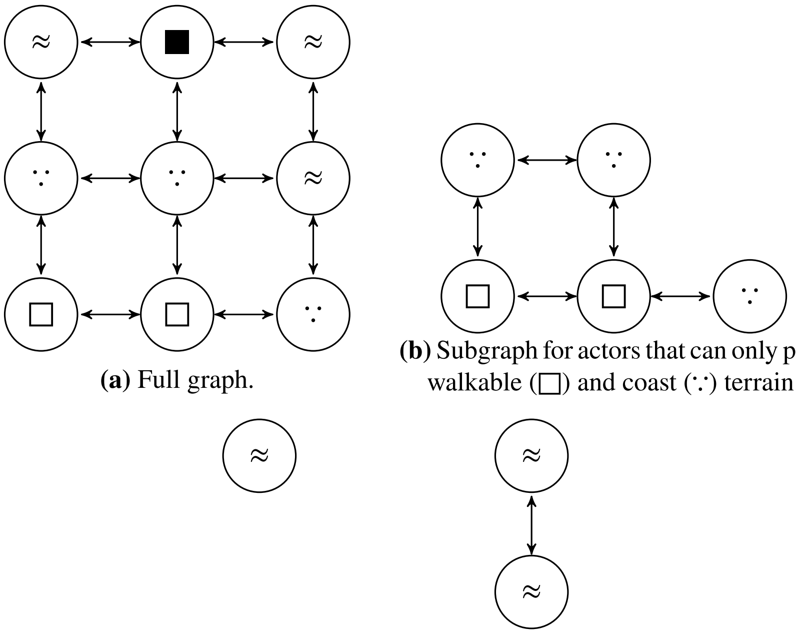
By only selecting the appropriate parts of the graph, the search space can be reduced before starting the pathfinding. Such a selection of subgraphs can be seen in Figure 4.5 for pedestrians and boats with Von Neumann neighbourhoods. This effort can in fact be done offline, by redundantly storing all accessible nodes for each class of actors. Having these subgraphs available allows us to exclude parts of the map before even starting the path search. It also paves the way for additional analyses on the graph that are only relevant for specific types of actors, as is described in Section 4.3. This selection is rather trivial and therefore not elaborated, but the view `passable_cells` which is outlined in Figure 4.4 is assumed in the following listings.

`actor_types` ⋈ `passable_terrain` ⋈ `terrain_types` ⋈ `map` =

passable_cells

cell_id	x	y	actor_type	actor_type_id	traversal_cost
1	1	1	⛵	2	1
2	1	2	🚶	1	1
3	1	3	🚶	1	2
4	1	4	🚶	1	2
5	2	1	⛵	2	1
...	...	...	...	...	...

FIGURE 4.4: View `passable_cells` which holds the parts of a map that are accessible to a certain type of actor.



(a) Full graph.

(b) Subgraph for actors that can only pass walkable (□) and coast (∵) terrain.

(c) Subgraph for boats that can only pass water (≈).

FIGURE 4.5: Preselected subgraphs for pedestrians and boats next to the full graph they are generated from.

4.3 Exploiting Connectivity to Speed up Pathfinding

Redundantly storing multiple subgraphs of the full graph also enables us to find all *unilaterally connected components* within each subgraph to speed up pathfinding in some cases. A unilaterally connected component $G' = (E', V')$ is part of a graph $G = (E, V)$ such that $E' \subseteq E$ and $V' \subseteq V$ where for each pair of nodes $(u, v) \in V'$ we can find a way through E' from u to v or from v to u [39, p. 199]. If start s and destination d of a path are not part of the same connected component, there is definitely no path connecting s and d and we can bail from the path search early. Note that we do not require *strongly connected components*, where we can find a way from u to v and from v to u, as that would give false positives as to a possible path search from s to d where no path from d to s can be found [39, p. 199].
Finding these connected components can be done by *flood filling* the graph. A naïve imperative implementation of flood filling would label a randomly picked unlabelled node and propagate that label through the graph as far as possible through neighbouring nodes. If the propagation stops, a new unlabelled node is picked. The algorithm would stop if all nodes have been labelled. All nodes with the same label are part of the same connected component. A more declarative approach would instead incrementally update the component each node is in at once, stopping as soon as the graph has reached a stable state:

1. Initially treat each node as its own component with a unique ID.
2. For all neighbourhoods, that is: each node n and all directly reachable nodes, label n with the largest component ID present within that neighbourhood.
3. Repeat step 2 until no node changes their component ID any more.

This algorithm can be expressed in pure SQL, using the `WITH RECURSIVE` construct:

First, we determine the neighbours for all nodes of the graph. `map` is the part of the full map that is traversable by a particular type of actor, as described in Section 4.2. Each neighbourhood consists of all adjacent nodes *plus the node the neighbourhood is being found for*, to cover the case where a cell is already part of a component with a larger ID than all of its neighbours. Initially, each node is a separate component.

```
CREATE FUNCTION connected_components(_actor_type_id INT)
RETURNS TABLE(cell_id INT, component_id INT) AS $$
    WITH RECURSIVE
    map(cell_id, x, y) AS (
        SELECT
            p.cell_id AS cell_id,
            p.x       AS x,
            p.y       AS y
        FROM
            passable_cells AS p
        WHERE
            p.actor_type_id = _actor_type_id
    ),
    neighbours(this_id, neighbour_id) AS (
        SELECT
            this.cell_id AS this_id,
            ns.cell_id   AS neighbour_id
        FROM
            map AS this,
            map AS ns
        WHERE
            (this.x, this.y) = (ns.x, ns.y)
            OR neighbouring((this.x, this.y), (ns.x, ns.y))
    ),
    components↻(cell_id, component_id) AS (
        (SELECT
            m.cell_id AS cell_id,
            m.cell_id AS component_id
        FROM
            map AS m
        )
...
```

In each recursive step we update the component ID for each cell by partitioning the graph into the neighbourhoods and selecting the largest value within that neighbourhood.

```
        UNION
        (
            SELECT
                c1.cell_id AS cell_id,
                MAX(c2.component_id) OVER (PARTITION BY c1.cell_id) AS component_id
            FROM
                components↻ AS c1
                JOIN neighbours AS ns
                  ON c1.cell_id = ns.this_id
                JOIN components↻ AS c2
                  ON c2.cell_id = cs.neighbour_id
        )
...
```

Due to the semantics of `UNION` the recursion stops once the largest component ID for each cell has been determined and the join yields no unique rows that have not already been discovered. In the final step, we select only the latest version of each cell, holding their largest and final component ID.

```
    SELECT DISTINCT ON (cell_id)
        cell_id,
        MAX(new_component_id) OVER (PARTITION BY cell_id)
    FROM
        components AS c
$$ LANGUAGE sql;
```

Figure 4.6 further illustrates this process in which all cells are flooding their neighbours at once until a stable state has been reached and the connected components within the graph have been established. The `neighbouring(a, b)` function introduced in Section 4.1.1 is used for this task as well. Since the connected components derived with this method are always specific to one actor type, the type of neighbourhood must reflect how that particular actor type would traverse the map.

Note that this computation can be done offline which takes some additional load off the DBMS during runtime.

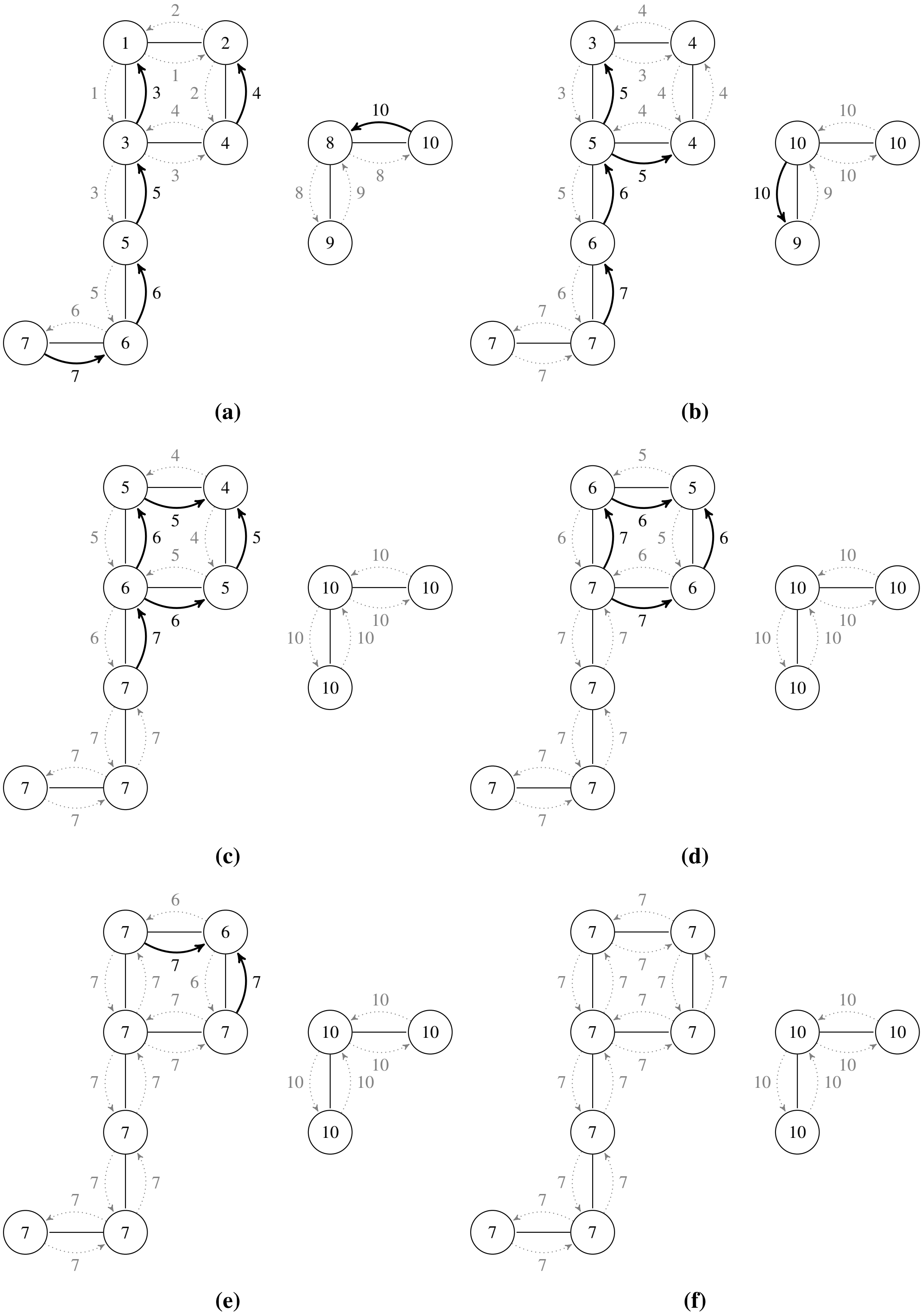


FIGURE 4.6: Flood filling over a graph with two connected components. Each node starts out as its own component with a running ID in Figure 4.6(a). In each step nodes are labelled with the highest component ID that has been propagated to them up to this point. Propagation from neighbours is shown with arrows: dotted grey arrows are being superseded, thick black arrows prevail and become the node's label in the next iteration. The flood filling ends as soon as no changes happen any more, resulting in two connected components with IDs 7 and 10 in Figure 4.6(f).

4.4 Spatial A* – in Pure SQL

A* is an informed algorithm, building on the *Dijkstra path finding* algorithm on graphs, meaning it does not necessarily explore the full graph. It finds the least expensive path from a starting node to a destination node by exploring the most promising nodes known up to that point. This well known algorithm can be implemented in SQL by utilising repeated updates of the map on which the path finding takes place. An imperative pseudo code implementation is given in Figure 4.7.

The following section explains the algorithm in detail. Since A* is well-known and the SQL implementation may look unusual to the reader, the algorithm is broken down into core sections with circled numbers, which are also attached at the corresponding SQL snippets to allow for a rough mapping of the SQL code to the imperative counterpart. Note that some implementation details in the SQL code have been omitted for the sake of brevity and readability.

```
def AStar(s, d, nodes):
  closedlist := {}
  openlist := {s}
  parents := {}
  gs     := {(n,∞)| ∀n ∈ nodes}   ①
  fs     := {(n,∞)| ∀n ∈ nodes}
  gs[s]  := 0
  fs[s]  := h(s,d)

  while openlist not empty: ⑥
      cheapest := pop_cheapest(openlist) ②
      if cheapest = d:
        return Path(s, d, parents)
      closedlist := closedlist ⋃ {cheapest}
      ns := neighbours(cheapest) \ closedlist ③
      for each n in ns:
        g' := gs[cheapest] + c(cheapest, n) ⑤
        if n ∉ openlist:
          openlist := openlist ⋃ {n}               ④
        if g' < gs[n]:
          parents[n] := cheapest
          gs[n] := g'
          fs[n] := g' + h(n, destination)
  return "no path"

⑦
def Path(s, d, parents):
  -- reconstruct the path by following the parents back to s
  path := []
  p := d
  while p ≠ s:
    append(path, p)
    p := parents[p]
  append(path, s)
  return path
```

FIGURE 4.7: Pseudocode implementation of imperative A* where pop_cheapest(xs) determines the element in xs with the entry with the lowest cost in fs and neighbours(n) finds all nodes that are directly reachable from n. append(xs,x) appends x to a list xs.

Throughout the search for a path from $s = (sx, sy)$ to $d = (dx, dy)$, A* maintains information about nodes that are yet to be visited (`openlist`) and which nodes have already been visited (`closedlist`). For every node n we track the costs $g(n)$ for travelling from s to n. Also, we can calculate $h(n)$, an estimation of the cost for travelling from n to d. This estimation must never exceed the actual costs. A more detailed explanation of this requirement is given in Section 4.4.1. Also, each visited node references a parent node from which it is reachable at the lowest cost.
Initially, g is infinite for all nodes and no node has a parent. The starting node is an exception to this, being the only node in the `openlist`, having precalculated costs, and a special parent with ID $= -1$, indicating that it has no parent. We also maintain a counter `iteration` to signify for each row in which step it received its most recent update.
The relation `map(id, x, y)` is the subset of the full map which the actor we are trying to find a path for can pass. ①

```
WITH RECURSIVE
map(cell_id, x, y, traversal_cost) AS (
    SELECT
        p.cell_id        AS cell_id,
        p.x              AS x,
        p.y              AS y,
        p.traversal_cost AS traversal_cost
    FROM
        passable_cells AS p
    WHERE
        p.actor_type_id = _actor_type_id -- id specified by caller
),
astar↻(iteration, cell_id, x, y, f, g, open, closed, parent, traversal_cost) AS (
    SELECT
        0                                AS iteration,
        m.cell_id                        AS cell_id,
        m.x                              AS x,
        m.y                              AS y,
        CASE (m.x,m.y)
          WHEN (sx,sy) THEN h(x,y, sx,sy)
          ELSE 0.0
        END                              AS f,
        CASE (m.x,m.y)
          WHEN (sx,sy) THEN 0
          ELSE ∞
        END                              AS g,
        (m.x,m.y) = (sx,sy)              AS open,
        FALSE                            AS closed,
        CASE (m.x,m.y)
          WHEN (sx,sy) THEN -1
          ELSE NULL
        END                              AS parent,
        m.traversal_cost                 AS traversal_cost
    FROM
        map AS m
...
```

At each step of the algorithm, the node *cheapest* from the `openlist` with the lowest $f(n) = g(n) + h(n)$ cost is removed and put into the `closedlist`, which can easily be achieved in our relational representation through `ORDER BY` and `LIMIT`: ②

```
cheapest(iteration, cell_id, x, y, f, g, open, closed, parent, traversal_cost) AS (
    SELECT
        iteration + 1  AS iteration,
        cell_id        AS cell_id,
        x              AS x,
        y              AS y,
        f              AS f,
        g              AS g,
        FALSE          AS open,    -- move from open...
        TRUE           AS closed,  -- ...to closed list
        parent         AS parent,
        traversal_cost AS traversal_cost
    FROM
        astar↻
    WHERE
        open
    ORDER BY
        f ASC
    LIMIT 1
),
...
```

The search is then extended `cheapest`'s spatial neighbours that are not yet part of the `closedlist`. For each neighbour n we calculate a *tentative g* that is $g(cheapest) + c(g,n)$, where $c(g,n)$ are the costs to move from g to n. ③

```
spatial_neighbours(iteration, cell_id, x, y, f, g, open, closed, parent, tentative_g, traversal_cost) AS (
    SELECT
        a.iteration              AS iteration,
        a.cell_id                AS cell_id,
        a.x                      AS x,
        a.y                      AS y,
        a.f                      AS f,
        a.g                      AS g,
        a.open                   AS open,
        a.closed                 AS closed,
        a.parent                 AS parent,
        ch.g + a.traversal_cost  AS tentative_g,
        a.traversal_cost         AS traversal_cost
    FROM
        cheapest AS ch,
        astar↻   AS a
    WHERE
        neighbouring((a.x, a.y), (ch.x, ch.y))
)
...
```

Of those neighbours we select only those that are not yet part of the `closedlist` and have a lower *tentative g* than their current g-value. Their parent is also set to *cheapest*. ④, ⑤

```
neighbours(iteration, cell_id, x, y, f, g, open, closed, parent,
           traversal_cost, checkin, checkout) AS (
SELECT
    ns.iteration + 1   AS iteration,
    ns.cell_id         AS cell_id,
    ns.x               AS x,
    ns.y               AS y,
    ns.tentative_g + h(ns.x, ns.y, _x2, _y2) AS f,
    ns.tentative_g     AS g,
    TRUE               AS open,
    ns.closed          AS closed,
    ch.cell_id         AS parent,
    ns.traversal_cost  AS traversal_cost,
FROM
    cheapest           AS ch,
    spatial_neighbours AS ns
    JOIN map
      ON ns.cell_id = map.cell_id
WHERE
    NOT ns.closed
    AND (NOT ns.open OR ns.tentative_g < ns.g)
)
...
```

Unifying these newly found candidates for expansion with the updated cheapest node and the former state of the map generates the new state. Note that at this point, all rows contributed by `neighbours` and `cheapest` (rows, which received an update and therefore form a subset of the former state) appear as duplicates of rows in `astar` (the complete former state). This issue is visualised in Figure 4.8. We will fix that in the next step.

```
updated(iteration, cell_id, x, y, f, g, open, closed, parent,
  traversal_cost, checkin, checkout) AS (
  TABLE neighbours
  UNION ALL
  TABLE cheapest
  UNION ALL
  TABLE astar↻
)
...
```

By ordering all rows by the iteration they were generated in and selecting them distinct on their ID, we can eliminate the duplicates caused by the last step and only have the latest version of each row. We continue the recursion until we have either reached our destination or the `openlist` is empty. ⑥

```
SELECT DISTINCT ON (u.id)
    u.*
FROM
    updated  AS u,
    cheapest AS ch,
    (SELECT COUNT(*) FROM updated WHERE open) AS olc(cnt)
WHERE
    olc.cnt > 0
    AND (ch.x,ch.y) <> (dx,dy)
ORDER BY id, iteration DESC
...
```

When the algorithm has finished, each node n knows another node (parent) from which n can be accessed with the least cost.

After completing the path search, the path can be constructed by recursively following the parent nodes from d to s. Since we initially assigned s a parent ID of -1 we can use that to terminate the reconstruction of the path. ⑦

```
path_steps↻(cell_id, x, y, parent, step) AS (
  SELECT
    a.cell_id AS cell_id,
    a.x       AS x,
    a.y       AS y,
    a.parent  AS parent,
    0         AS step
  FROM
    astar↻ AS a
  WHERE
    (a.x,a.y) = (dx,dy)

  UNION ALL

  SELECT
    a.cell_id   AS cell_id,
    a.x         AS x,
    a.y         AS y,
    a.parent    AS parent,
    ps.step + 1 AS step
  FROM
    astar↻ AS a
    JOIN path_steps↻ AS ps
      ON a.id = ps.parent
  WHERE
    ps.parent > -1
)
```

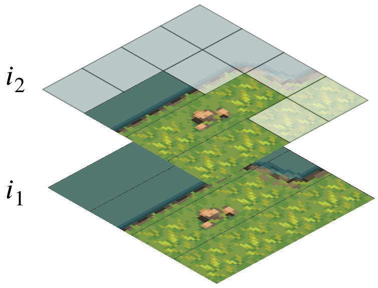


FIGURE 4.8: Update step of A*. Only some of the nodes in the first step labelled i_1 receive an update in the second step labelled i_2, rendered opaque. That makes those exact nodes appear twice in the final result. Ordering them descending on the step number they stem from and selecting them distinct on their coordinate within the grid ensures that we only carry the most recent version of each cell.

4.4.1 Heuristics

Heuristics for A* need to be *admissible*, meaning that they may not overestimate the actual costs. Which heuristic is applicable for a path search depends on which kind of *neighbourhood* is being used.
When using a Von Neumann neighbourhood, the *Manhattan distance* is a commonly used heuristic, which sums up the differences between the coordinate components of s and d. More formally, the Manhattan distance between two n-dimensional coordinates $p = (p_1, p_2, \cdots, p_{n-1}, p_n)$ and $q = (q_1, q_2, \cdots, q_{n-1}, q_n)$ is defined as

$$Manhatten\_Distance(p, q) = \sum_{i=1}^{n} |p_i - q_i|$$

For Moore neighbourhoods, the Manhattan distance would overestimate the cost, as moving diagonally can decrease the costs compared to moving in a stair-like fashion. Instead we can use the *Euclidean distance*, which is defined as

$$Euclidian\_Distance(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

or the *Chebyshev distance*, which assumes a cost of 1 for moving onto any of the eight adjacent cells:

$$Chebyshev\_Distance(p, q) = max(\{|q_i - p_i| | \forall i \in \{1..n\}\})$$

All three heuristics are visualised in Figure 4.9.

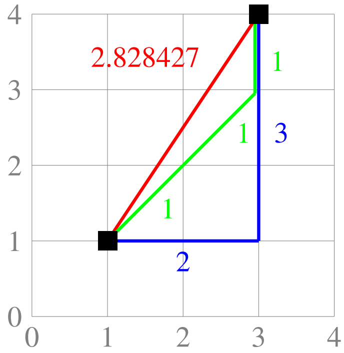
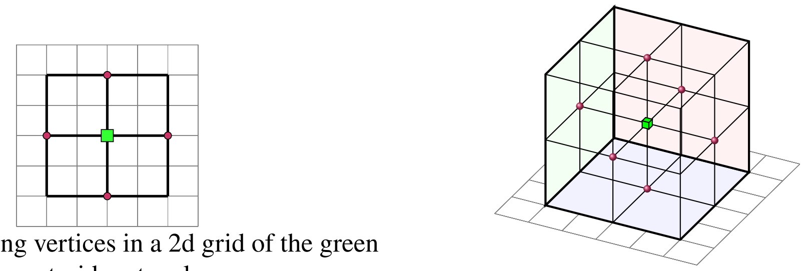


FIGURE 4.9: Manhattan distance of $2 + 3 = 5$ between $(1, 1)$ and $(3, 4)$ in blue, Chebyshev distance of $1 + 1 + 1 = 3$ in green and Euclidian distance of 2.828427 in red.

4.4.2 Dimensionality of A*

As the algorithm is generally oblivious towards the dimensionality of the map that is being traversed, A* is used for both two-dimensional and three-dimensional path finding scenarios. Applying A* to a map of higher dimension is easy in most cases. It usually suffices to adjust `neighbouring(a, b)` to retrieve neighbours from the new dimension as well. See Figure 4.10 to understand how the vertex in question (green rectangle or green cube respectively) gains more neighbours (magenta circles or magenta spheres respectively) when lifted from two-dimensional to three-dimensional space. The path finding algorithm stays the same at core.



(a) Neighbouring vertices in a 2d grid of the green centroid rectangle.

(b) Neighbouring vertices in a 3d grid of the green centroid cube.

FIGURE 4.10: Lifting `neighbouring(a, b)` from two to three dimensions to acquire the Von Neumann neighbourhood.

4.5 Temporal A* – Avoiding Collisions with a Booking System

Path finding allows computer controlled actors to navigate through a map with static obstacles, like walls or boulders. But in most scenarios, actors are also dynamic obstacles to each other, meaning that they can not occupy the same space at the same time. Any game that does not allow such states must implement a collision avoidance or resolution mechanism. Games usually resolve collisions when they occur by having their colliding actors back up by a random amount to give their adversary room to navigate around them. But conflicts like this could actually be avoided in the planning phase: making the path search aware that parts of the map can be temporarily occupied, and where in time-space the planning takes place, enables the algorithm to choose paths presciently.

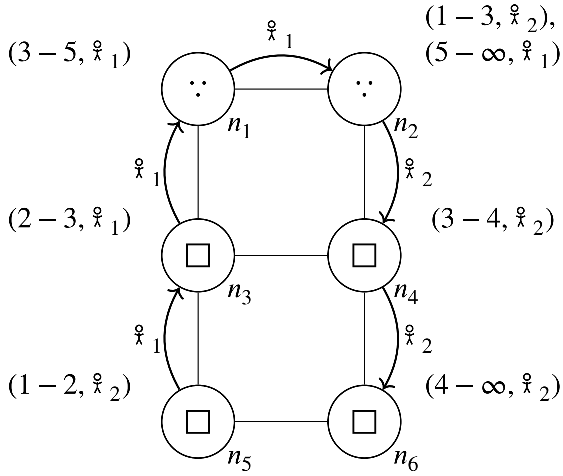
passable_terrain′

actor_type	terrain	duration	...
웃	□	1	...
웃	∵	2	...

reservations

cell_id	checkin	checkout	resident
1	3	5	웃$_1$
2	1	3	웃$_2$
2	5	∞	웃$_1$
3	2	3	웃$_1$
4	3	4	웃$_2$
5	1	2	웃$_2$
6	4	∞	웃$_2$

(a) Tables holding how many time steps it takes each actor to traverse terrain of a certain type and to store reservations for cells of a resolved path. Foreign key relations have been resolved in this rendering.



(b) Spatial paths for 웃$_1$ from n_5 to n_2 and for 웃$_2$ from n_2 to n_6. Each cell has a list of reservations next to it, consisting of tuples of time period during which it is occupied and the actor which occupies the cell.

FIGURE 4.11: Example of how actors move through a graph of varying terrain type, affecting the reservations on the nodes they visit.

While making for some interesting behaviour, just giving each coordinate $(c_1, c_2, \cdots, c_{n-1}, c_n)$ in the graph another component t to represent time does not give us the desired results for navigating through the temporal dimension: in most games time strictly moves forward[3], so we must only include neighbours of future time steps. We also do not try to minimise costs (that is in regards to time: finding the quickest way between two points) and therefore can leave the heuristic unaffected. Instead it is more of an additional constraint embedded in `neighbouring(a, b)`: to avoid collisions, we need to determine for each neighbour of n how long it would take our actor 웃$_x$ in question to travel through it, and whether that neighbour is already being traversed by another actor during that time frame. If any two actors 웃$_x$ and 웃$_y$ want to travel through any node in overlapping time frames, we have foreseen a collision and can look for an alternative path. Instead of resolving this conflict when it happens, we can avoid collisions during the path finding itself. To support this, each step in the path search produces an additional column holding the timestamp at which the actor we search the path for would start off at a given cell. We also define a relation `reservations`, to track when an actor (`resident`) occupies a cell during a time frame, defined by a start (`checkin`) and an end point (`checkout`). We also need to know for each actor how long it takes them to traverse a cell of a certain type. For this, we can extend the `passable_terrain` relation from Figure 4.2 with a `duration` column. Both tables are exemplified in Figure 4.11(a).

[3] The game *Braid* by Jonathan Blow makes a delightful exception to this.

When the search for spatial neighbours is done, we can already infer for each neighbour how long it would take the travelling actor to traverse said neighbour if it was not occupied (specified by `start` and `until`), based on the underlying terrain, seen in Figure 4.11(a). We can use that information to remove occupied cells from the spatial neighbours by performing an *anti join* with the `reservations` table. An anti join $\rhd$ is a binary operation on two relations R and S which produces only the tuples of R for which no join partner in S can be found. It can also be expressed as $R \rhd S = R - (R \ltimes S)$. This presents us with the temporal neighbours:

$$temporal\_neighbours := spatial\_neighbours \underset{(checkin,checkout)\text{ not overlaps }(start,until)}{\rhd} reservations$$

Take Figure 4.11(b) for example. 🧍$_1$ wants to travel from n_5 to n_2. A path for 🧍$_2$ from n_2 to n_6 has already been scheduled. Both actors want to start their journey at timestamp $t = 1$. 🧍$_1$ could spatially travel from n_3 to n_4, but that would mean they would need to occupy n_4 from $t = 3$ to $t = 4$. This overlaps with 🧍$_2$ already having a reservation during that time, so 🧍$_1$ instead travels via n_1.

Adjusting the code in Section 4.4 would be rather easy: two new columns `checkin` and `checkout` are added to the CTE `map`, which are dragged through the chain of CTEs to keep track of hypothetical occupation of each cell by the actor for which the path is being found. That is: change the CTE starting at line 3 to contain `checkin` and `checkout`, based on the current timestamp in the game (`now()`) and the travel duration over each cell:

```
3   SELECT
4       p.cell_id         AS cell_id,
5       c.x               AS x,
6       c.y               AS y,
7       p.traversal_cost  AS traversal_cost
```

→

```
3   SELECT
4       p.cell_id               AS cell_id,
5       c.x                     AS x,
6       c.y                     AS y,
7       p.traversal_cost        AS traversal_cost,
8       p.traversal_time        AS traversal_time,
9       now()                   AS checkin,
10      now()+p.traversal_time  AS checkout
```

Maintain temporal information about the stay of the actor when determining the neighbours in line 59:

```
59  SELECT
60      a.iteration       AS iteration,
...     ...
71      a.traversal_cost  AS traversal_cost
72  FROM
73      cheapest AS c,
74      astar⟲   AS a
```

→

```
59  SELECT
60      a.iteration       AS iteration,
...     ...
71      a.traversal_cost  AS traversal_cost,
72      c.checkout        AS checkin,
73      c.checkout + a.traversal_time AS checkout
74  FROM
75      cheapest AS c,
76      astar⟲   AS a
```

Finally, introduce a new CTE `temporal_neighbours` after `spatial_neighbours` which removes all spatial neighbours for which a reservation is found for the timeframe it would take the actor to traverse that neighbour:

```
77    , temporal_neighbours(iteration, cell_id, x, y, f, g,
78                          open, closed, parent, tentative_g, traversal_cost)
79      AS (
80        SELECT
81            sn.*
82        FROM
83            cheapest AS c,
84            spatial_neighbours AS sn
85              JOIN map
86                ON sn.cell_id = map.cell_id
87              LEFT JOIN reservations AS r
88                ON sn.cell_id = r.cell_id
89        WHERE
90            -- these are not reserved at all
91            r.cell_id IS NULL
92            -- these are reserved, but do not overlap
93            OR NOT ((r.start, r.until) OVERLAPS (c.checkout + map.traversal_time,
94                                                 c.checkout + 2 * map.traversal_time))
95  )
```

Then base the selection of neighbours on those temporal neighbours, instead of the spatial neighbours in line 93:

```
91  FROM                                            91  FROM
92      cheapest           AS c,                    92      cheapest            AS c,
93      spatial_neighbours AS ns            →       93      temporal_neighbours AS ns
94      JOIN map                                    94      JOIN map
95        ON ns.cell_id = map.cell_id               95        ON ns.cell_id = map.cell_id
```

With this additional information, actors for which a path has been resolved can make reservations ahead of their actual presence on each cell of the path, making them unavailable for other actors during the respective time frame. Note that, while not shown in the accompanying listings, this booking system also makes the implementation of a *wait*-operation fairly simple, by making a cell its own spatial and temporal neighbour, enabling actors to wait for other actors to pass if that is cheaper than finding a way around a temporarily blocked passage.

4.6 Iterative Path Finding

The implementation described in Section 4.4 maintains the interface programmers are used to when utilising a path finding algorithm: a start and a destination position in an n-dimensional space are specified, alongside a metric to properly calculate the traversal costs – the actor for which a path is sought in this case. While algorithmically faithful to reference implementations, it comes with a certain disconnect from the set-based philosophy behind SQL, which excels at specifying operations on many elements at once. So instead of following a single actor through the whole process of the path search in a *depth-first* approach, the following section proposes an alternative implementation which gradually finds paths for multiple agents. Not only is this more in line with how DBMSs work natively, it also gives us the opportunity to insert additional path finding processes into potentially long-running searches. While the first approach relies on `WITH RECURSIVE` to drive the entire path search, the following approach needs to be called repeatedly to advance all ongoing path searches. These calls could come from a rudimentary tick-loop that is not part of the SQL code. Not relying on `WITH RECURSIVE` to drive the search also gives us the opportunity to make use of independent writing queries, such as `INSERT`-, `UPDATE`-, or `DELETE`-statements, where we do not need to drag intermediate results through a chain of CTEs.

Instead of having a CTE representing the state of the ongoing path search, we introduce an additional table `node_lists`:

```
CREATE TABLE node_lists(
    node_list_id              SERIAL PRIMARY KEY,
    actor_id                  INT,
    actor_type_id             INT,
    cell_id                   INT,
    start                     POINT,
    destination               POINT,
    position                  POINT,
    traversal_cost            DOUBLE PRECISION,
    predecessor               INT DEFAULT NULL,
    g                         DOUBLE PRECISION DEFAULT ∞,
    f                         DOUBLE PRECISION DEFAULT ∞,
    closed                    BOOLEAN DEFAULT FALSE,
    open                      BOOLEAN DEFAULT FALSE,
    FOREIGN KEY(actor_type_id) REFERENCES actor_types(actor_type_id),
    FOREIGN KEY(actor_id)      REFERENCES actors(actor_id),
    FOREIGN KEY(cell_id)       REFERENCES cells(cell_id),
    FOREIGN KEY(predecessor)   REFERENCES cells(cell_id),
    UNIQUE(cell_id, actor_id)
);
```

While a traditional imperative implementation would gradually add newly discovered nodes to the closed and open list, each path finding process initialises all nodes for their respective actor here. As before, only the part of the map the actor in question can actually pass is considered, which is determined by their `_actor_type_id`. It is always assumed to have at most one active path search per actor, so all nodes related to that search can be identified by the `actor_id`.

```
CREATE FUNCTION init_nodelist(_actor_id INT, _actor_type_id INT, _start POINT, _destination POINT)
RETURNS VOID AS $$
    -- cancel stale searches and start path search: init closed/open list
    DELETE FROM node_lists WHERE actor_id = _actor_id;
    INSERT INTO node_lists(actor_id, actor_type_id, cell_id, position, traversal_cost, start, destination)
        SELECT
            _actor_id,
            _actor_type_id,
            pc.cell_id,
            POINT(pc.x, pc.y),
            pc.traversal_cost,
            _start,
            _destination
        FROM
            passable_cells AS pc
        WHERE
            pc.actor_type_id = _actor_type_id
$$ LANGUAGE sql;
...
```

The `node_list` that was formerly initialised for the actor now has the starting point of the search set as the cheapest available node in the open list.

```
CREATE FUNCTION init_search(_actor_id INT, _start POINT, _destination POINT)
RETURNS VOID AS $$
    SELECT init_nodelist(_actor_id,
                        (SELECT type_id
                         FROM actors
                         WHERE actor_id = _actor_id),
                        start,
                        destination);
    UPDATE
        node_lists AS nl
    SET
        g = 0.0,
        f = 0.0,
        open = TRUE
    WHERE
        nl.position = _start AND nl.actor_id = _actor_id
$$ LANGUAGE sql;
...
```

The most striking difference from the previous implementation is the fact that we do not look for the cheapest node in the open list upon expansion, but for the cheapest node *per ongoing search*, which can easily be achieved by using a window-based partitioning by actor.

```
CREATE FUNCTION advance() RETURNS VOID AS $$
    WITH
    ordered(node_lists_id, actor_id, cell_id, position, g, f_rank) AS (
        SELECT
            id       AS node_lists_id,
            actor_id AS actor_id,
            cell_id  AS cell_id,
            position AS position,
            g        AS g,
            ROW_NUMBER() OVER (PARTITION BY actor_id ORDER BY f ASC) AS f_rank
        FROM
            node_lists AS nl
        WHERE
            open
    ),
    cheapest(actor_id, cell_id, position, g) AS (
        UPDATE
            node_lists AS nl
        SET
            open = FALSE,
            closed = TRUE
        WHERE
            node_lists_id IN (SELECT node_lists_id FROM ordered WHERE f_rank = 1)
        RETURNING
            actor_id,
            cell_id,
            position,
            g
    ),
...
```

We can then proceed with the regular expansion step, in which the neighbours of the cheapest cells are incorporated into the search, if they have not already been fully explored. The findings of the exploration are propagated to the `node_list` as an `UPDATE`-statement.

```
    expand(actor_id, this_id, neighbour_id, neighbour_pos, open, tentative_g, g, destination) AS (
        SELECT
            this.actor_id              AS actor_id,
            this.cell_id               AS this_id,
            ns.cell_id                 AS neighbour_id,
            ns.position                AS neighbour_pos,
            ns.open                    AS open,
            this.g + ns.traversal_cost AS tentative_g,
            ns.g                       AS g,
            ns.destination             AS destination
        FROM
            cheapest AS this
            JOIN node_lists AS ns
              ON neighbouring(this.position, ns.position)
                 AND this.actor_id = ns.actor_id
        WHERE
            NOT ns.closed AND NOT (ns.open AND (this.g + ns.traversal_cost) >= ns.g)
    )
    UPDATE node_lists AS nl
    SET
        open = TRUE,
        g = e.tentative_g,
        f = e.tentative_g + h(e.neighbour_pos[0], e.neighbour_pos[1], e.destination[0], e.destination[1]),
        predecessor = e.this_id
    FROM
        expand AS e
    WHERE
        nl.cell_id = e.neighbour_id AND nl.actor_id = e.actor_id
$$ LANGUAGE sql;
···
```

Calling the `advance`-routine may produce finished paths that will just reside within the `node_list` table until retrieved. The UDF `resolve_paths` can then be either called every so often or after each `advance`-call to collect paths that are complete. Endpoints that have a predecessor are indicators for successful path searches, that can be reconstructed by following the `predecessors` up to the start node, which has no `predecessor`, producing an empty result, which stops the recursion.

```
CREATE FUNCTION resolve_paths()
RETURNS TABLE(steps INT, actor_id INT, cell_id INT, position POINT) AS $$
    WITH RECURSIVE
    endpoints(actor_id, cell_id, destination, position, predecessor) AS (
        SELECT
            actor_id,
            cell_id,
            destination,
            position,
            predecessor
        FROM
            node_lists
        WHERE
            destination = position AND predecessor IS NOT NULL
    ),
    complete_paths↻(steps, actor_id, cell_id, position, predecessor) AS (
        SELECT
            0,
            actor_id,
            cell_id,
            position,
            predecessor
        FROM
            endpoints
        UNION ALL
        SELECT
            p.steps + 1,
            nl.actor_id,
            nl.cell_id,
            nl.position,
            nl.predecessor
        FROM
            node_lists AS nl
            JOIN complete_paths↻ AS p
              ON nl.cell_id = p.predecessor
                 AND nl.actor_id = p.actor_id
    ),
···
```

Finally, we need to take into account that there might be no valid path between a desired start and destination for an actor. Those paths can be identified by having destinations with no predecessor, but no more nodes left to explore, which is implied by the open column. An invalid path can be signalled in various ways, in this implementation, we return a single row where all fields are NULL except for the actor_id of the actor we were searching a path for.

```
    paths(steps, actor_id, cell_id, position, predecessor) AS (
        SELECT * FROM complete_paths
        UNION ALL (
            -- unavailable paths
            WITH
            pending(actor_id) AS (
                SELECT
                    actor_id
                FROM
                    node_lists AS nl
                WHERE
                    position = destination
                    AND predecessor IS NULL
            ),
            empty(actor_id) AS (
                SELECT
                    nl.actor_id
                FROM
                    node_lists AS nl
                GROUP BY
                    nl.actor_id
                HAVING
                    COUNT(open) FILTER (WHERE NOT open) = COUNT(nl.actor_id)
            )
            SELECT
                NULL,
                p.actor_id,
                NULL,
                NULL,
                NULL,
            FROM
                pending AS p JOIN empty AS e ON p.actor_id = e.actor_id
        )
    ),
...
```

The node_list can then be cleaned up in an optional step by removing nodes for all actors for which a path has been resolved.

```
    se_cleanup(actor_id) AS (
        DELETE FROM
            node_lists AS nl
        WHERE
            nl.actor_id IN (SELECT DISTINCT actor_id FROM paths)
        RETURNING
            1
    )
    SELECT
        p.steps,
        p.actor_id,
        p.cell_id,
        p.position
    FROM
        paths AS p
$$ LANGUAGE sql;
```

4.7 Evaluation

Path finding in video games needs to be able to determine paths in a timely manner to uphold the illusion of autonomous actors. On the other hand, Claypool and Claypool found that path finding is among the actions in a video game where players are willing to accept higher latencies [18], which was especially confirmed for real time strategy games by Claypool [17], for which path finding is especially important. The following section therefore evaluates the performance of pathfinding within DBMSs as it was presented in the preceding sections. Only the spatial, two-dimensional versions of all implementations are considered to maintain comparability. Searches are always expected to resolve to a path. This is ensured through

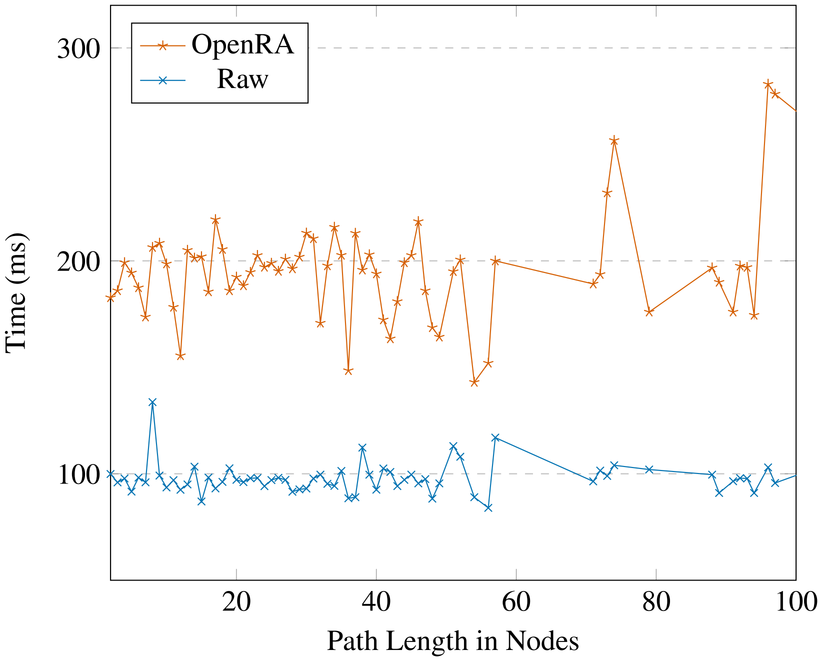


FIGURE 4.12: Comparison between times when calling pgRouting from OpenRA versus calling it from the command line (raw).

the use of a labyrinth as map which forms a single component in which actors can search for paths between random passable cells of the map. Therefore, no pre-emptive check for the connectedness of the components in which start and destination lie is performed, which would immediately terminate the path search if no path was available. To establish a baseline, path finding based on pgRouting (and later the custom implementation) was integrated into OpenRA: the entire map of a skirmish was written into the DBMS upon starting the game and a thin wrapper around the path finding component was implemented to replace calls to the native path finding of the engine with calls to pgRouting. To mitigate possible performance hits, the requests to the database were sent in a separate thread. Actors were sent towards their target coordinate in a straight line to give the player immediate palpable feedback for their input. The actual path was pursued once the background thread returned.

As data set, the path finding during a game between two computer-controlled opponents was recorded and timed, using the wall clock time between the moment the request to find a path was sent, and the moment the path was fully returned to the engine and converted into C# data-structures. Start, destination, and actor of each search were recorded and later replayed directly on the DBMS to determine the impact of not having the entire engine within the DBMS, but only swapping out the path finding itself. The results show that this split had a considerable hit on the runtime, ramping up the time to find a path from around 100 ms to around 200 ms. The found timings are shown in Figure 4.12. Naturally, this hit must be applied to all other approaches as well, as it is a general penalty of travelling between the DBMS and another system.

Then, the timings of running pgRouting from the command line were compared to the times when running the same path search in the native SQL implementation from Section 4.4. While the time stayed about the same throughout all calls when using pgRouting, the duration gradually increased in relation to the resulting length of the path when using the native implementation. This can be explained by how the intermediate table during the native path search grows in each step, resulting in more expensive intermediate joins.

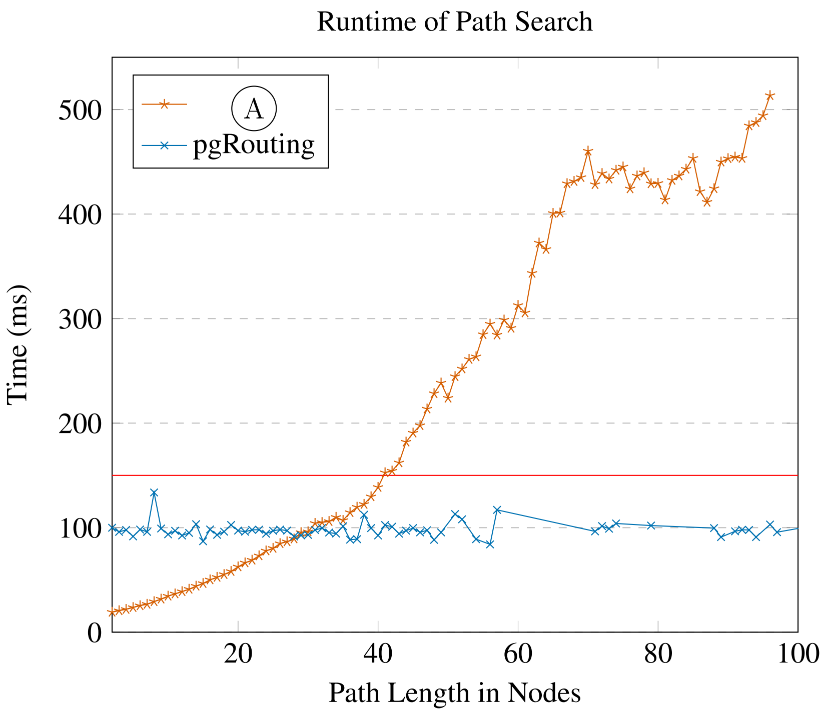


FIGURE 4.13: Comparison of path finding using pgRouting and the native implementation Ⓐ in SQL from Section 4.4 . A loose boundary of 150 ms as agreeable waiting time is delineated in red.

Shorter paths on the other hand can be found faster using the native implementation. This discrepancy can be attributed to the fact that the accessible part of map for each actor is readily available and indexed for the native implementation, whereas pgRouting needs to prepare the graph beforehand. As different types of games come with different time boundaries for how long players are willing for a path search to complete, we call on the measurements of competitive *StarCraft* [86] games conducted by Lewis et al. [49] as a soft upper bound. They found that professional gamers would execute up to 400 *actions per minute (APM)*, i.e. one action every 150 milliseconds, giving us a lose upper bound for preparatory work like determining a path between actions.

Plotting the distribution of path lengths of the aforementioned match in Figure 4.14 shows that a large share (57%) of all path searches were well within the range that resulted in agreeable time boundaries. In real-world scenarios, especially for real time strategy games, short path searches are the most time-critical ones, as they are part of the micromanagement of units that occurs during heated battles.

Figure 4.15 shows a comparison between the imperative approach Ⓐ from Section 4.4, the relational version Ⓑ from Section 4.6 and pgRouting. As advancing Ⓑ requires repeated calls from an outside driver and capitalises on its potential when applied to many path searches at once, it can hardly be compared to Ⓐ and pgRouting on singular searches. Instead, the comparison shows how long it takes for all three approaches to return results for 1 000 random path searches. Each search is started as soon as possible; that is: Ⓐ and pgRouting start each search consecutively whenever the result of the former call is yielded, while Ⓑ starts all searches at once.

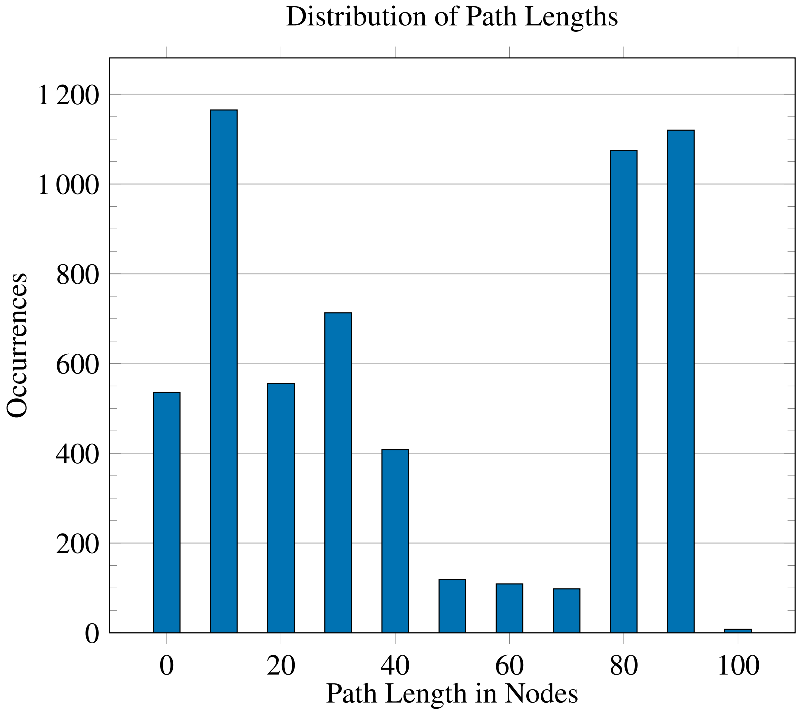


FIGURE 4.14: Distribution of resulting path lengths during a game between two computer opponents.

Since (B) is biased towards short-running path searches, returning their result as soon as possible, the curve flattens towards the top, representing the long-running searches that are returned last. The other two approaches expectedly produce a reliably linear curve.

Accordingly, the DBMS offers sufficiently fast means of finding paths for actors. While PostgreSQL's pgRouting offers an established extension, the two DBMS-agnostic approaches presented in this chapter perform well within the expected time-bounds. A reasonable speed-boost can be achieved if the developers are willing to stray from the customary interface of performing one entire path search at a time when using approach (B), favouring short path searches.

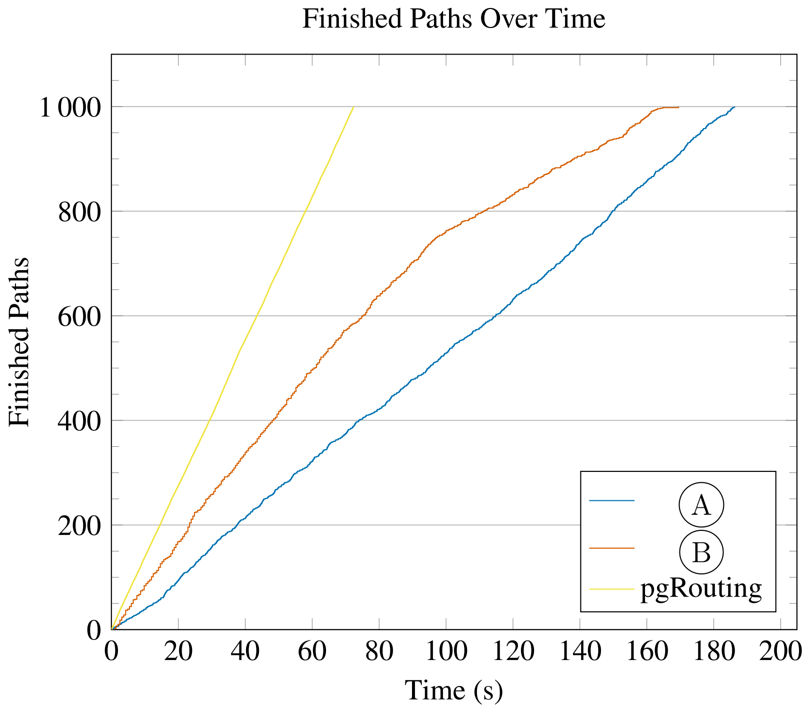


FIGURE 4.15: Running 1 000 path searches in imperative A* (A) from Section 4.4, the relational version (B) from Section 4.6, and on pgRouting.

Chapter 5

Discussion and Final Remarks

In this work we presented the implementation of typical components of a video game engine in pure SQL, namely *artificial intelligence* in the sense of video games (Chapter 2), *map generation* (Chapter 3), and *path finding* (Chapter 4). The components were purposefully selected to be applicable to a wide range of game genres, as opposed to concentrating on components that are only used in niche games, such as a component for parallax scrolling, which would be very specific for side scrolling games. Despite the intention of keeping the presented implementation of all components as general as possible, the proposed code is obviously not universally applicable to all games without a certain amount of tweaking and certainly bears the potential for further improvements. While one can hardly surpass the established finely tuned game engines – that sometimes work hand in hand with hardware on a low level [87], as is the case with *NVidia*'s *PhysX* [63] library – in all regards, the suggested implementations performed acceptably for most use cases by running reasonably fast despite the additional abstraction layer SQL has to deal with, or by making use of the *time–memory trade-off (TMTO)* by calculating data beforehand and storing it for efficient retrieval. This demonstrates that DBMS are indeed a viable vehicle for transporting even the more unlikely parts of video game engines. It is therefore important to note that the presented components are meant to be understood as an advancement in addition to what DBMSs already have to offer for video game developers! As mentioned in Chapter 1, game engines are already moving towards software design principles that are more suitable to how DBMSs work with data by reinventing techniques that are indigenous to the world of database systems. That means:

1. structuring of complex data in a modular fashion through foreign key relations;
2. indexing attributes with a multitude of readily available suited index structures;
3. aggregating attributes efficiently
4. having a persistent and consistent data storage
5. performing spatial checks, i.e. collision detection or finding actors within a certain proximity of each other
6. expressing updates as operations on sets, making them inherently suited for parallel execution
7. post-mortem analysis of persistently stored game data and replay capabitilties if the applied actions are preserved (see Section 2.3)
8. being able to shard computation and storage across servers, and so on

all come *for free* in addition to everything that was discussed in this work. Implementing established components in SQL that perform reasonably well can thereby be deemed satisfactory, as they serve as an enabler for the incorporation of benefits inherent to DBMSs. As discussed in Section 4.7, splitting the game engine between the imperative and the declarative world comes with a certain hit to the runtime when transferring data between those two realms. It stands to reason to try to move as many components of the game engine as possible to the DBMS to make the data that has to travel to and from the database as lightweight as possible. Obviously, there are some components that currently seem unlikely to ever be transferred entirely to a database. For instance, any component that receives and processes input from the user via keystrokes, mouse clicks, motion control, or other means will likely stay on the imperative side. Likewise, databases do not lend themselves to implement rendering – although utilising the GPU from within the DBMS is not unheard of [12]. As such, a middle ground between doing the heavy lifting of data transformation in a database-driven backend, paired with a lean, imperative frontend towards the user appears to be a worthwhile long-term goal.

The work presented in this thesis is just one step from the traditional use of databases towards the world of imperative video game engines that are already taking large strides towards the relational way of thinking.

Appendix A

DDL Statements

A.1 Introduction

A.1.1 Recursive SQL Queries

```
CREATE TABLE enemy (
    id     SERIAL PRIMARY KEY,
    x      INT,
    y      INT,
    health INT
);
```

```
CREATE TABLE platforms (
    x     INT,
    y     INT,
    width INT
);
```

```
CREATE TABLE position_components (
    id SERIAL PRIMARY KEY,
    x  INT,
    y  INT
);
```

A.2 Video Game AI

A.2.1 Tactics Through DFA

```
CREATE TABLE actors (
    id        SERIAL PRIMARY KEY,
    name      TEXT,
    position  VECTOR2D,
    speed     DOUBLE PRECISION,
    range     DOUBLE PRECISION,
    damage    INT,
    hitpoints INT,
    target_id INT REFERENCES actors(id)
);
```

```
CREATE TABLE actor_states (
    dfa_id   INT REFERENCES dfas(id),
    state_id INT REFERENCES states(id),
    actor_id INT REFERENCES actors(id)
);
```

```
CREATE TABLE conditions (
    id     SERIAL PRIMARY KEY,
    fname  TEXT
);
```

```
CREATE TABLE dfas (
    id            SERIAL PRIMARY KEY,
    initial_state INT REFERENCES states(id)
);
```

```
CREATE TABLE edges (
    id            SERIAL PRIMARY KEY,
    weight        INT,
    current_state INT REFERENCES states(id),
    condition_id  INT REFERENCES conditions(id),
    next_state    INT REFERENCES states(id),
    effect_id     INT REFERENCES effects(id)
);
```

```
CREATE TABLE effects (
    id    SERIAL PRIMARY KEY,
    fname TEXT
);
```

```
CREATE TABLE states (
    id SERIAL PRIMARY KEY
);
```

A.2.2 Monte Carlo Tree Search

```
CREATE TABLE actions(
    id SERIAL PRIMARY KEY,
    ... -- game specific
);
```

```
CREATE TABLE rewards(
    state_id INT REFERENCES states(id),
    reward   NUMERIC
);
```

```
CREATE TABLE states(
    id SERIAL PRIMARY KEY
);
```

```
CREATE TABLE tree_children(
    parent_id INT NOT NULL REFERENCES states(id),
    action_id INT NOT NULL REFERENCES actions(id),
    child_id  INT NOT NULL REFERENCES states(id),
    UNIQUE(parent_id, child_id), -- each tree can only be attached to another tree once
    UNIQUE(parent_id, action_id) -- per parent, each action may only be applied once
);
```

```
CREATE VIEW tree_statistics(state_id, total_reward, visit_count) AS (
    SELECT
        r.state_id      AS state_id,
        SUM(r.reward)   AS total_reward,
        COUNT(r.reward) AS visit_count
    FROM
        rewards AS r
    GROUP BY
        r.state_id
);
```

A.3 Map Generation

A.3.1 Rule-Based Map Generation

```
CREATE TABLE δ(
    rule   INT,
    input  TEXT NOT NULL REFERENCES ΣV(symbol)
    x      INT CHECK (x BETWEEN 0 AND 2),
    y      INT CHECK (y BETWEEN 0 AND 2),
    output TEXT NOT NULL REFERENCES ΣV(symbol)
);
```

```
CREATE TABLE ΣV(
    symbol   TEXT PRIMARY KEY,
    terminal BOOLEAN
);
```

A.3.2 Module-Based Map Generation

```
CREATE TABLE 𝒞(
    frequency    INT,
    mapside      TEXT REFERENCES 𝒯(symbol),
    modside      TEXT REFERENCES 𝒯(symbol),
    moduleset_id INT REFERENCES modulesets(id)
);
```

```
CREATE VIEW ℳ(moduleset_name, module_id, x, y, symbol) AS (
    SELECT
        ms.moduleset_name AS moduleset_name,
        m.id              AS module_id,
        mc.x              AS x,
        mc.y              AS y,
        t.symbol          AS symbol
    FROM
        modulesets AS ms
        JOIN modules AS m
          ON ms.id = m.moduleset_id
        JOIN module_contents AS mc
          ON m.id = mc.module_id
        JOIN 𝒯 AS t
          ON mc.tile_id = t.id
);
```

```
CREATE TABLE module_contents(
    module_id INT REFERENCES modules(id),
    x         INT CHECK (x BETWEEN 0 AND 2),
    y         INT CHECK (y BETWEEN 0 AND 2),
    tile_id   INT REFERENCES 𝒯(id),
    UNIQUE(module_id, x, y)
);
```

```
CREATE TABLE modules(
    id           SERIAL PRIMARY KEY,
    moduleset_id INT NOT NULL REFERENCES modulesets(id)
);
```

```
CREATE TABLE modulesets(
    id   SERIAL PRIMARY KEY,
    name TEXT UNIQUE
);
```

```
CREATE TABLE 𝒯(
    id     SERIAL PRIMARY KEY,
    symbol TEXT UNIQUE
);
```

Module-Based Map Generation

A.3.3 Map Generation by Example

```
CREATE TABLE mapdata(
    id      SERIAL PRIMARY KEY,
    x       INT,
    y       INT,
    tile_id INT,
    UNIQUE(x, y)
);
```

```
CREATE VIEW read_compatibilities(tile_id1, tile_id2, relative_x, relative_y, frequency) AS (
    WITH
    neighbours(tile_id1, tile_id2, relative_x, relative_y) AS (
        SELECT
            m1.tile_id,
            m2.tile_id,
            m2.global_x - m1.global_x,
            m2.global_y - m1.global_y
        FROM
            modules AS m1,
            modules AS m2
        WHERE
    ),
    compatible(tile_id1, tile_id2, relative_x, relative_y, frequency) AS (
        SELECT
            n.tile_id1,
            n.tile_id2,
            n.relative_x,
            n.relative_y,
            COUNT(*)
        FROM
            neighbours AS n
        GROUP BY
            n.tile_id1, n.tile_id2, n.relative_x, n.relative_y
    ),
    bidirectional(tile_id1, tile_id2, relative_x, relative_y, frequency) AS (
        SELECT
            c.tile_id1,
            c.tile_id2,
            c.relative_x,
            c.relative_y,
            c.frequency
        FROM
            compatible AS c
        UNION ALL
        SELECT
            c.tile_id2,
            c.tile_id1,
            -c.relative_x,
            -c.relative_y,
            c.frequency
        FROM
            compatible AS c
    )
    SELECT
        b.tile_id1,
        b.tile_id2,
        b.relative_x,
        b.relative_y,
        SUM(b.frequency)
    FROM
        bidirectional AS b
    GROUP BY
);
```

```
CREATE VIEW read_modules(module_id, tile_id, module_x, module_y, x, y, global_x, global_y)
AS (
    SELECT
        ⌊md.x / 3⌋ + ⌈(SELECT MAX(x) + 1 FROM mapdata) / 3⌉ * ⌊md.y / 3⌋
                      AS module_id,
        md.tile_id    AS tile_id,
        md.x / 3      AS module_x,
        md.y / 3      AS module_y,
        md.x % 3      AS x,
        md.y % 3      AS y,
        md.x          AS global_x,
        md.y          AS global_y
    FROM
        dim,
        mapdata AS md
);
```

```
CREATE VIEW unique_modules(map_id, module_id, tile_id,
                           module_x, module_y, x, y,
                           global_x, global_y, occurrences)
AS (
WITH
    hashed_modules(module_id, hash) AS (
        SELECT
            m.module_id AS module_id,
            string_agg(m.tile_id, '|' ORDER BY m.x,m.y) AS hash
        FROM
            read_modules AS m
        GROUP BY
            m.module_id
    ),
    module_set(module_id, occurrences) AS (
        SELECT
            MAX(m.id) AS module_id,
            COUNT(*)  AS occurrences
        FROM
            hashed_modules AS m
        GROUP BY
            m.hash
    )
    SELECT
        mod.id,
        mod.tile_id,
        mod.module_x,
        mod.module_y,
        mod.x,
        mod.y,
        mod.global_x,
        mod.global_y,
        ms.occurrences
    FROM
        module_set AS ms
        JOIN read_modules AS mod
          ON ms.module_id = mod.id
);
```

A.4 Pathfinding

A.4.1 Path Finding in Video Games

```
CREATE TABLE cells(
    id SERIAL PRIMARY KEY,
    x  INT,
    y  INT
    UNIQUE(x, y)
);
```

```
CREATE TABLE cell_contents(
    cell_id    INT REFERENCES cells(id),
    terrain_id INT REFERENCES terrain_types(id)
);
```

```
CREATE TABLE terrain_types(
    id     SERIAL PRIMARY KEY,
    symbol TEXT UNIQUE
);
```

```
CREATE TABLE actor_types(
    id          SERIAL PRIMARY KEY,
    symbol      TEXT UNIQUE,
    description TEXT
);
```

```
CREATE TABLE passable_terrain(
    actor_type_id  INT REFERENCES actor_types(id),
    actor_type     TEXT REFERENCES actor_types(symbol) -- only included for readability
    terrain_id     INT REFERENCES terrain_types(id),
    terrain        TEXT REFERENCES terrain_types(symbol), -- only included for readability
    traversal_cost INT CHECK (traversal_cost >= 0)
);
```

```
CREATE TABLE actors(
    id            SERIAL PRIMARY KEY,
    actor_type_id INT REFERENCES actor_types(id),
    label         TEXT
);
```

```
CREATE VIEW map(cell_id, x, y, terrain) AS (
    SELECT
        c.id      AS cell_id,
        c.x       AS x
        c.y       AS y,
        tt.symbol AS terrain
    FROM
        cells AS c
        JOIN cell_contents AS cc
          ON c.id = cc.cell_id
        JOIN terrain_types AS tt
          ON cc.terrain_id = tt.id
);
```

```
CREATE VIEW passable_cells(cell_id, x, y, actor_type, actor_type_id, traversal_cost) AS (
    SELECT
        map.cell_id            AS cell_id,
        THE(map.x)             AS x,
        THE(map.y)             AS y,
        THE(at.actor_type)     AS actor_type,
        at.actor_type_id       AS actor_type_id,
        SUM(pt.traversal_cost) AS traversal_cost
    FROM
        actor_types AS at
        JOIN passable_terrain AS pt
          ON at.id = pt.actor_type_id
        JOIN terrain_types AS tt
          ON tt.id = pt.terrain_id
        JOIN map
          ON map.terrain = pt.terrain
    GROUP BY
        map.cell_id, actor_type_id
);
```

A.4.2 Temporal A* – Avoiding Collisions with a Booking System

```
CREATE TABLE passable_terrain(
    actor_type_id  INT REFERENCES actor_types(id),
    actor_type     TEXT REREFENCES actor_types(symbol)
    terrain_id     INT REFERENCES terrain_types(id),
    terrain        TEXT REFERENCES terrain_types(symbol),
    traversal_cost INT CHECK (traversal_cost >= 0),
    duration INT CHECK (duration >= 0)
);
```

```
CREATE TABLE actors(
    id            SERIAL PRIMARY KEY,
    actor_type_id INT REFERENCES actor_types(id),
    label         TEXT
);
```

A.4.3 Iterative Path Finding

```
CREATE TABLE node_lists(
    node_list_id               SERIAL PRIMARY KEY,
    actor_id                   INT,
    actor_type_id              INT,
    cell_id                    INT,
    start                      POINT,
    destination                POINT,
    position                   POINT,
    traversal_cost             DOUBLE PRECISION,
    predecessor                INT DEFAULT NULL,
    g                          DOUBLE PRECISION DEFAULT ∞,
    f                          DOUBLE PRECISION DEFAULT ∞,
    closed                     BOOLEAN DEFAULT FALSE,
    open                       BOOLEAN DEFAULT FALSE,
    FOREIGN KEY(actor_type_id) REFERENCES actor_types(actor_type_id),
    FOREIGN KEY(actor_id)      REFERENCES actors(actor_id),
    FOREIGN KEY(cell_id)       REFERENCES cells(cell_id),
    FOREIGN KEY(predecessor)   REFERENCES cells(cell_id),
    UNIQUE(cell_id, actor_id)
);
```

Appendix B

Perlin Noise in SQL

```
CREATE TYPE vector AS (
    x DOUBLE PRECISION,
    y DOUBLE PRECISION
);

-----------------------------------------------------------------------------------------

CREATE TABLE grid(
    x INT,
    y INT,
    mesh INT,
    v vector,
    UNIQUE(x,y)
);

CREATE TABLE pixels(
    x INT,
    y INT,
    UNIQUE(x,y)
);

-----------------------------------------------------------------------------------------

-- determines the length of _v.
CREATE FUNCTION length(_v vector)
RETURNS DOUBLE PRECISION AS $$
    SELECT SQRT(_v.x * _v.x + _v.y * _v.y);
$$ LANGUAGE sql IMMUTABLE;

-- subtracts _v1 from _v2.
CREATE FUNCTION sub(_v1 vector, _v2 vector)
RETURNS vector AS $$
    SELECT row(_v2.x - _v1.x, _v2.y - _v1.y)::vector;
$$ LANGUAGE sql IMMUTABLE;

-- adds _v1 to _v2.
CREATE FUNCTION add(_v1 vector, _v2 vector)
RETURNS vector AS $$
    SELECT row(_v2.x + _v1.x, _v2.y + _v1.y)::vector;
$$ LANGUAGE sql IMMUTABLE;

-- calculates the distance between _v1 and _v2.
CREATE FUNCTION distance(_v1 vector, _v2 vector)
RETURNS DOUBLE PRECISION AS $$
    SELECT SQRT((_v1.x - _v2.x) * (_v1.x - _v2.x) + (_v1.y - _v2.y) * (_v1.y - _v2.y));
$$ LANGUAGE sql IMMUTABLE;

-- normalises _v to length 1. Vectors of length 0 will result in (0 0).
CREATE FUNCTION normalise(_v vector) RETURNS vector AS $$
    SELECT CASE WHEN length(_v) = 0
                THEN row(0,0)::vector
                ELSE row(
                    _v.x / length(_v),
                    _v.y / length(_v)
                )::vector
            END;
$$ LANGUAGE sql IMMUTABLE;

-- dot product between _v1 and _v2.
CREATE FUNCTIOn dot(_v1 vector, _v2 vector)
RETURNS DOUBLE PRECISION AS $$
    SELECT _v1.x * _v2.x + _v1.y * _v2.y;
$$ LANGUAGE sql IMMUTABLE;

-- linear interpolation of _x between _a and _b.
CREATE FUNCTION lerp(_a DOUBLE PRECISION, _b DOUBLE PRECISION, _x DOUBLE PRECISION)
```

```
RETURNS DOUBLE PRECISION AS $$
    SELECT _a + _x * (_b - _a);
$$ LANGUAGE sql IMMUTABLE;

-- fade function as defined by Ken Perlin: 6t^5-15t^4+10t^3
CREATE FUNCTION fade(_x DOUBLE PRECISION)
RETURNS DOUBLE PRECISION AS $$
   SELECT 6 * POWER(_x, 5) - 15 * POWER(_x, 4) + 10 * POWER(_x, 3);
$$ LANGUAGE sql IMMUTABLE;

-- scales _x between _lo and _hi. ie, scale(15, 10, 20) results in .5.
CREATE FUNCTION scale(_x INT, _lo INT, _hi INT)
RETURNS DOUBLE PRECISION AS $$
    SELECT (_x - _lo)::DOUBLE PRECISION / (_hi - _lo);
$$ LANGUAGE sql IMMUTABLE;

-- creates a random vector between (-1 -1) and (1 1).
CREATE FUNCTION randomv()
RETURNS vector AS $$
    SELECT row(RANDOM() * 2 - 1, RANDOM() * 2 - 1)::vector;
$$ LANGUAGE sql VOLATILE;

------------------------------------------------------------------------------------------

-- initialises pixels (image) and grid. The grid will have a mesh size of _mesh
-- and will hang over all sides of the image, so even the border pixels will
-- have grid points to be calculated from.
CREATE FUNCTION init(_w INT, _h INT, _mesh INT)
RETURNS BOOLEAN AS $$
    DELETE FROM pixels;
    INSERT INTO
        pixels(x, y)
        SELECT
            r.a,
            s.b
        FROM
            generate_series(1, _w) AS r(a),
            generate_series(1, _h) AS s(b)
    ;

    DELETE FROM grid;
    INSERT INTO
        grid(x, y, mesh, v)
        SELECT
            r.a * _mesh,
            s.b * _mesh,
            _mesh,
            randomv()
        FROM
            generate_series(0, (_w / _mesh)+1) AS r(a),
            generate_series(0, (_h / _mesh)+1) AS s(b)
        ;
    SELECT TRUE;
$$ LANGUAGE sql VOLATILE;

------------------------------------------------------------------------------------------

-- determines the 4 nearest corners from the grid for each pixel.
-- Those are exactly the four corners of the enclosing unit square.
CREATE VIEW corners(x, y, cx, cy, influence) AS (
    WITH corners(px, py, cx, cy, cv, drank) AS (
        SELECT
            p.x AS px,
            p.y AS py,
            g.x AS cx,
            g.y AS cy,
            g.v AS cv,
            -- pixels that are situated exactly on a gridline can produce incorrect neighbours when looking for the distance.
            -- We therefore nudge on pixel up/ to the left/ both so that they properly fall into a unit square.
            -- 1 - ceil(x % n) / n) produces 1 for multiples of n and 0 for everything else.
            ROW_NUMBER() OVER
                (PARTITION BY p.x, p.y
                 ORDER BY distance(row(p.x - (1 - CEIL((p.x % g.mesh) / g.mesh)), p.y - (1 - CEIL((p.y % g.mesh) / g.mesh)))::vector,
                                              row(g.x, g.y)::vector))
                AS drank
        FROM
            pixels AS p
            JOIN grid AS g
              -- this join condition is not really required, since we take the 4 nearest neighbours anyway,
              -- which are exactly the four corners of the unit square around the coordinate,
              -- but it produces a smaller intermediate result.
              ON distance(row(p.x, p.y)::vector,
                                 row(g.x, g.y)::vector) <= g.mesh * 1.4142 -- times sqrt(2) for the diagonal
    )
    SELECT
        px AS x,
        py AS y,
        cx AS cx,
        cy AS cy,
        dot(cv,
            sub(row(px, py)::vector,   -- \ distance vector
                row(cx, cy)::vector))  -- /
           AS influence
    FROM
        corners
    WHERE
        drank <= 4
);

-----------------------------------------------------------------------------------------

-- generates perlin noise for an image of size _w x _h with a mesh size of _mesh.
CREATE FUNCTION noise(_w INT, _h INT, _mesh INT = 10)
RETURNS TABLE(x INT, y INT, value DOUBLE PRECISION) AS $$
    SELECT init(_w, _h, _mesh);
    WITH cs(x, y, infs, minx, miny, maxx, maxy) AS (
        SELECT
            x,
            y,
            ARRAY_AGG(influence),
            MIN(cx) AS minx,
            MIN(cy) AS miny,
            MAX(cx) AS maxx,
            MAX(cy) AS maxy
        FROM
            corners AS c
        GROUP BY
            x,
            y
    )
    SELECT
        x,
        y,
        (lerp(lerp(
                infs[1],
                infs[2],
                fade(scale(x, minx, maxx)),
              lerp(
                infs[3],
                infs[4],
                fade(scale(x, minx, maxx)),
              fade(scale(y, miny, maxy))) + 1)/2 -- [-1,1] -> [0,1]
    FROM
        cs
    ;
$$ LANGUAGE sql VOLATILE;
```

Bibliography

[1] Mike Acton. *Data-Oriented Design and C++*. https://www.youtube.com/watch?v=rX0ItVEVjHc. Accessed: 30 January 2020. Insomniac Games, 2014.

[2] Tony Albrecht. *The Latency Elephant*. http://seven-degrees-of-freedom.blogspot.com/2009/10/latency-elephant.html. Accessed: 30 January 2020. 2009.

[3] Robert Albright et al. "SGL: A Scalable Language for Data-Driven Games". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, 1217–1222. ISBN: 9781605581026. DOI: 10.1145/1376616.1376739. URL: https://doi.org/10.1145/1376616.1376739.

[4] Zeyad A. Algfoor, Mohd S. Sunar, and Hoshang Kolivand. "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". In: *Int. J. Comput. Games Technol.* 2015 (Jan. 2015). ISSN: 1687-7047. DOI: 10.1155/2015/736138. URL: https://doi.org/10.1155/2015/736138.

[5] Louis V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. Den Haag, NL: CIP–Gegevens Koninklijke Bibliotheek, Jan. 1994. ISBN: 9789090074887. URL: https://books.google.de/books?id=c7FTAgAACAAJ.

[6] Entertainment Software Associaction. *2018 Essential Facts About the Computer and Video Game Industry*. http://www.theesa.com/esa-research/2018-essential-facts-about-the-computer-and-video-game-industry/. Accessed: 3 February 2020. 2018.

[7] Entertainment Software Associaction. *2019 Essential Facts About the Computer and Video Game Industry*. https://www.theesa.com/esa-research/2019-essential-facts-about-the-computer-and-video-game-industry/. Accessed: 3 February 2020. 2019.

[8] Bob Bates. *Game Design*. 2nd. Boston, MA, USA: Thomson Course Technology, 2004. ISBN: 9781592004935. URL: https://books.google.de/books?id=f7XFJnGrb3UC.

[9] Guy E. Blelloch et al. "Implementation of a Portable Nested Data-Parallel Language". In: PPOPP '93. San Diego, California, USA: Association for Computing Machinery, 1993, 102–111. ISBN: 0897915895. DOI: 10.1145/155332.155343. URL: https://doi.org/10.1145/155332.155343.

[10] Jonathan Blow. *Data-Oriented Demo: SOA, composition*. https://www.youtube.com/watch?v=ZHqFrNyLlpA. Accessed: 30 January 2020. 2015.

[11] Michael Booth. *The AI Systems of Left 4 Dead*. http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf. Accessed: 21 August 2019. Valve, 2009.

[12] Sebastian Breß et al. "GPU-Accelerated Database Systems: Survey and Open Challenges". In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events*. Ed. by Abdelkader Hameurlain et

al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 1–35. ISBN: 978-3-662-45761-0. DOI: 10.1007/978-3-662-45761-0_1. URL: https://doi.org/10.1007/978-3-662-45761-0_1.

[13] Cameron B. Browne et al. "Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (Mar. 2012), pp. 1–43. ISSN: 1943-068X, 1943-0698. DOI: 10.1109/TCIAIG.2012.2186810. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6145622.

[14] Michael J. Carey and David J. DeWitt. "Of Objects and Databases: A Decade of Turmoil". In: *Proceedings of the 22th International Conference on Very Large Databases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1996, 3—14. ISBN: 1558603824.

[15] Guillaume M. J-B. Chaslot et al. "Progressive Strategies for Monte-Carlo Tree Search". In: *New Mathematics and Natural Computation* 04.03 (Nov. 2008), pp. 343–357. ISSN: 1793-0057. DOI: 10.1142/s1793005708001094.

[16] *Chris Kohler interviewing Atsushi Inaba on "God Hand"*. https://www.wired.com/2006/10/interview-atsus/. Accessed: 21 August 2019. Oct. 2006.

[17] Mark Claypool. "The Effect of Latency on User Performance in Real-Time Strategy Games". In: *Computer Networks* 49 (Sept. 2005), pp. 52–70. DOI: 10.1016/j.comnet.2005.04.008.

[18] Mark Claypool and Kajal Claypool. "Latency and Player Actions in Online Games". In: *Communications of the ACM* 49.11 (Nov. 2006), 40–45. ISSN: 0001-0782. DOI: 10.1145/1167838.1167860. URL: https://doi.org/10.1145/1167838.1167860.

[19] Edgar F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. URL: http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.

[20] George Copeland and David Maier. "Making Smalltalk a Database System". In: *SIGMOD Rec.* 14.2 (June 1984), 316–325. ISSN: 0163-5808. DOI: 10.1145/971697.602300. URL: https://doi.org/10.1145/971697.602300.

[21] *Cry Engine*. https://docs.cryengine.com/. Accessed: 11 March 2020. Crytek.

[22] Mihaly Csikszentmihalyi. *Beyond Boredom and Anxiety*. San Francisco: Jossey-Bass, 1975. ISBN: 0787951404. DOI: 10.2307/2065805.

[23] Mihaly Csikszentmihalyi. *Flow and the Foundations of Positive Psychology*. New York, NY, USA: Springer, 2014. ISBN: 9789401790833. DOI: 10.1007/978-94-017-9088-8.

[24] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[25] *Doom 3 Engine*. https://github.com/id-Software/DOOM-3. Accessed: 11 March 2020. id Software.

[26] *DOOM Engine*. https://github.com/id-Software/DOOM. Accessed: 11 March 2020. id Software.

[27] *Dungeon Generation in Diablo 1*. https://www.boristhebrave.com/2019/07/14/dungeon-generation-in-diablo-1/. Accessed: 17 January 2020.

[28] Sehar S. Farooq and Kyung-Joong Kim. "Game Player Modeling". In: *Encyclopedia of Computer Graphics and Games*. Ed. by Newton Lee. New York City, NY, USA: Springer, Dec. 2015, pp. 1–5. ISBN: 978-3-319-08234-9. DOI: 10.1007/978-3-319-08234-9_14-1. URL: https://doi.org/10.1007/978-3-319-08234-9_14-1.

[29] Theresa Fleming et al. "Serious Games and Gamification for Mental Health: Current Status and Promising Directions". In: *Frontiers in Psychiatry* 7 (Jan. 2017). DOI: `10.3389/fpsyt.2016.00215`.

[30] *DuckDB Documentation on `WITH RECURSIVE`*. `https://github.com/cwida/duckdb/pull/404`. Accessed: 5 November 2020.

[31] *MariaDB Documentation on `WITH RECURSIVE`*. `https://jira.mariadb.org/browse/MDEV-9864`. Accessed: 5 November 2020. MariaDB Foundation.

[32] *MySQL Documentation on `WITH RECURSIVE`*. `https://dev.mysql.com/doc/refman/8.0/en/mysql-nutshell.html#mysql-nutshell-additions`. Accessed: 5 November 2020. Oracle Corporation.

[33] *PostgreSQL Documentation on `WITH RECURSIVE`*. `https://postgresql.org/docs/8.4/release-8-4.html`. Accessed: 5 November 2020. PostgreSQL Global Development Group.

[34] *SQLite Documentation on `WITH RECURSIVE`*. `https://sqlite.org/releaselog/3_8_3.html`. Accessed: 5 November 2020.

[35] Michael C. Green et al. "Two-step Constructive Approaches for Dungeon Generation". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. Vol. abs/1906.0. FDG '19. San Luis Obispo, California, USA: Association for Computing Machinery, Aug. 2019. ISBN: 9781450372176. DOI: `10.1145/3337722.3341847`. arXiv: `arXiv:1906.04660v1`. URL: `https://doi.org/10.1145/3337722.3341847`.

[36] Jason Gregory. *Game Engine Architecture, Second Edition*. 2nd. Boca Raton, FL, USA: CRC Press, 2014. ISBN: 1466560010.

[37] Torsten Grust, Nils Schweinsberg, and Alexander Ulrich. "Functions Are Data Too: Defunctionalization for PL/SQL". In: vol. 6. 12. Riva del Garda, Trento, IT: VLDB Endowment, Aug. 2013, pp. 1214–1217. DOI: `10.14778/2536274.2536279`. URL: `http://dx.doi.org/10.14778/2536274.2536279`.

[38] *Half Life Sourcecode*. `https://github.com/ValveSoftware/halflife/`. Accessed: 16 October 2020. Valve.

[39] Frank Harary. *Graph Theory*. Boston, MA: Addison-Wesley Publishing Company, Inc. ISBN: 0-201-02787-9.

[40] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107.

[41] Robin Hunicke and Vernell Chapman. "AI for Dynamic Difficulty Adjustment in Games". In: *Challenges in Game Artificial Intelligence AAAI Workshop* 2 (Jan. 2004), pp. 91–96.

[42] Stratos Idreos et al. "MonetDB: Two Decades of Research in Column-Oriented Database Architectures". In: *IEEE Data Engineering Bulletin* 35.1 (2012), pp. 40–45.

[43] Philip C. Jr. Jackson. *Introduction To Artificial Intelligence*. 2nd. Mineola, NY, USA: Dover Publications, 1985. ISBN: 048624864X.

[44] *James Brightman interviewing Warren Spector*. `https://www.gamesindustry.biz/articles/2012-03-16-warren-spector-a-lifetime-of-achievements`. Accessed: 27 Feburary 2020. Mar. 2012.

[45] Simon P. Jones and Philip Wadler. "Comprehensive Comprehensions". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: ACM, Sept. 2007, pp. 61–72. ISBN: 978-1-59593-674-5. DOI: `10.1145/1291201.1291209`. URL: `http://doi.acm.org/10.1145/1291201.1291209`.

[46] Korina Katsaliaki and Navonil Mustafee. "A Survey of Serious Games on Sustainable Development". In: *Proceedings of the Winter Simulation Conference*. WSC '12. Berlin, Germany: Winter Simulation Conference, Dec. 2012.

[47] Steven L. Kent. *The Ultimate History of Video Games: From Pong to Pokemon — The Story Behind the Craze That Touched Our Lives and Changed the World*. New York City, NY, USA: Crown, 2001. ISBN: 0761536434.

[48] Tobias Kretz and Michael Schreckenberg. "Moore and More and Symmetry". In: *Pedestrian and Evacuation Dynamics 2005*. Ed. by Nathalie Waldau et al. Berlin, Heidelberg: Springer Berlin Heidelberg, May 2007, pp. 297–308. ISBN: 978-3-540-47064-9.

[49] Joshua M. Lewis, Patrick Trinh, and David Kirsh. "A Corpus Analysis of Strategy Video Game Play in Starcraft: Brood War". In: *Cognitive Science* 33 (2011). ISSN: 1069-7977.

[50] Adam Martin. *Entity Systems are the future of MMOG development*. `http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/`. Accessed: 31 January 2020. Sept. 2007.

[51] Donald Michie. ""Memo" Functions and Machine Learning". In: *Nature* 218.5138 (Apr. 1968), p. 306. ISSN: 1476-4687. DOI: `10.1038/218306c0`. URL: `https://doi.org/10.1038/218306c0`.

[52] *Minecraft*. `https://www.minecraft.net/`. Accessed: 4 April 2020. Mojang Studios.

[53] *OpenRA Engine*. `https://www.openra.net/`. Accessed: 10 March 2020.

[54] *Oracle Documentation on `WITH RECURSIVE`*. `https://docs.oracle.com/cd/E11882_01/server.112/e41360/chapter1.htm#NEWFT107`. Accessed: 5 November 2020. Oracle Corporation.

[55] Carlos Ordonez and Ladjel Bellatreche. "A Survey on Parallel Database Systems From a Storage Perspective: Rows Versus Columns". In: *Database and Expert Systems Applications*. Ed. by Mourad Elloumi et al. New York City, NY, USA: Springer International Publishing, 2018, pp. 5–20. ISBN: 978-3-319-99133-7.

[56] Bruce Pandolfini. *Kasparov and Deep Blue: The Historic Chess Match Between Man and Machine*. A Fireside Book. New York City, NY, USA: Touchstone, 1997. ISBN: 9780684848525. URL: `https://books.google.de/books?id=4eZh2giYL5MC`.

[57] Ken Perlin. "An Image Synthesizer". In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. Vol. 19. SIGGRAPH '85 3. New York, NY, USA: Association for Computing Machinery, July 1985, 287–296. ISBN: 0897911660. DOI: `10.1145/325334.325247`. URL: `https://doi.org/10.1145/325334.325247`.

[58] Ken Perlin. "Improving Noise". In: SIGGRAPH '02. San Antonio, Texas: Association for Computing Machinery, July 2002, 681–682. ISBN: 1581135211. DOI: `10.1145/566570.566636`. URL: `https://doi.org/10.1145/566570.566636`.

[59] *Perlin Noise in Minecraft*. `https://github.com/UnknownShadow200/ClassiCube/wiki/Minecraft-Classic-map-generation-algorithm/6234239c9429d69dd9f945cef12d28c59e632cc7`. Accessed: 20 July 2020. Dec. 2015.

[60] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. "Towards Deep Player Behavior Models in MMORPGs". In: CHI PLAY '18. Melbourne, VIC, Australia: Association for Computing Machinery, Oct. 2018, 381—392. ISBN: 9781450356244. DOI: `10.1145/3242671.3242706`. URL: `https://doi.org/10.1145/3242671.3242706`.

[61] Johannes Pfau et al. "Bot or Not? User Perceptions of Player Substitution with Deep Player Behavior Models". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, Apr. 2020, 1—10. ISBN: 9781450367080. DOI: `10.1145/3313831.3376223`. URL: `https://doi.org/10.1145/3313831.3376223`.

[62] *pgRouting*. `https://pgrouting.org/`. Accessed: 7 September 2020.

[63] *PhysX*. https://developer.nvidia.com/gameworks-physx-overview. Accessed: 11 November 2020. NVidia.

[64] *Postgresql Configuration Recommendations*. https://wiki.postgresql.org/wiki/Simple_Configuration_Recommendation#PostgreSQL_Configuration_Recommendations. Accessed: 1 December 2020. PostgreSQL Global Development Group.

[65] *Postgresql Documentation on Planner*. https://www.postgresql.org/docs/current/planner-optimizer.html. Accessed: 28 October 2020. PostgreSQL Global Development Group.

[66] *Postgresql Documentation on Planner Configuration*. https://www.postgresql.org/docs/current/runtime-config-query.html. Accessed: 28 October 2020. PostgreSQL Global Development Group.

[67] *Postgresql Documentation on Predicate Pushdown in Common Table Expressions*. https://www.postgresql.org/docs/12/queries-with.html. Accessed: 24 November 2020. PostgreSQL Global Development Group.

[68] *Postgresql Documentation on EXPLAIN*. https://www.postgresql.org/docs/current/using-explain.html. Accessed: 28 October 2020. PostgreSQL Global Development Group.

[69] Daniel Průša. "Two-Dimensional Languages". PhD thesis. Charles University Prague, 2004.

[70] *Quake 2 Engine*. https://github.com/id-Software/Quake-2. Accessed: 11 March 2020. id Software.

[71] *Quake Engine*. https://github.com/id-Software/Quake. Accessed: 11 March 2020. id Software.

[72] *Quake III Arena Engine*. https://github.com/id-Software/Quake-III-Arena. Accessed: 11 March 2020. id Software.

[73] Azriel Rosenfeld. *Picture Languages: Formal Models for Picture recognition*. English. New York City, NY, USA: Academic Press New York, 1979, xiii, 225 p. ISBN: 0125973403.

[74] Fabien Sanglard. *Game Engine Black Book: Doom: v1.1*. 1.1. Independently published, 2019. ISBN: 978-1099819773. URL: http://fabiensanglard.net/gebbdoom_v1.1.pdf.

[75] Fabien Sanglard. *Game Engine Black Book: Wolfenstein 3D*. Game Engine Black Book. Scotts Valley, CA, USA: CreateSpace Independent Publishing Platform, 2017. ISBN: 9781539692874.

[76] Frederik C. Schadd. "Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis". MA thesis. Maastricht University, Netherlands, 2009.

[77] Noor Shaker et al. "Constructive generation methods for dungeons and levels". In: *Procedural Content Generation in Games*. New York City, NY, USA: Springer, Oct. 2016. Chap. 3, pp. 31–55. ISBN: 978-3-319-42714-0. DOI: 10.1007/978-3-319-42716-4_3.

[78] *Sid Meier's Civilization*. https://civilization.com/. Accessed: 9 October 2020. Firaxis Games.

[79] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *Computing Research Repository* abs/1712.01815 (Dec. 2017). arXiv: 1712.01815.

[80] David Silver et al. "Mastering the Game of Go Without Human Knowledge". In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[81] Herbert A. Simon and Toshinori Munakata. "The Implications of Kasparov Vs. Deep Blue". In: *Communications of the ACM* 40.8 (1997), pp. 21–25. ISSN: 15577317. DOI: 10.1145/257874.257878.

[82] Steven S. Skiena. *The Algorithm Design Manual*. London, UK: Springer, Sept. 2008. ISBN: 9781848000704 1848000707 9781848000698 1848000693. DOI: 10.1007/978-1-84800-070-4.

[83] Harald Søndergaard and Peter Sestoft. "Referential Transparency, Definiteness and Unfoldability". In: *Acta Informatica* 27.6 (May 1990), pp. 505–517. ISSN: 00015903. DOI: 10.1007/BF00277387.

[84] Benjamin Sowell et al. "From Declarative Languages to Declarative Processing in Computer Games". In: *Computing Research Repository* abs/0909.1770 (2009). arXiv: 0909.1770. URL: http://arxiv.org/abs/0909.1770.

[85] International Organization for Standardization. *ISO/IEC 9075-2:1999: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. Geneva, CH: International Organization for Standardization, 1999, p. 1121. URL: http://www.iso.ch/cate/d26197.html.

[86] *StarCraft*. https://starcraft.com/. Accessed: 8 September 2020. Blizzard Entertainment.

[87] Cardon Stéphane and Jacopin Éric. "Binary GPU-Planning for Thousands of NPCs". In: *2020 IEEE Conference on Games (CoG)*. Osaka, JP: IEEE, Aug. 2020, pp. 678–681. DOI: 10.1109/CoG47356.2020.9231696.

[88] George Stiny and James Gips. "Shape Grammars and the Generative Specification of Painting and Sculpture". In: *Information Processing, Proceedings of IFIP Congress*. Ed. by Charles V. Freiman, John E. Griffith, and Jack L. Rosenfeld. Vol. 2. Amsterdam, NL: North Holland Publishing, Jan. 1971, pp. 1460–1465.

[89] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling*. Cambridge, MA, USA: MIT Press, Apr. 1987. ISBN: 0262200600.

[90] Julian Togelius et al. "What is Procedural Content Generation? Mario on the Borderline". In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: Association for Computing Machinery, Jan. 2011. ISBN: 9781450308724. DOI: 10.1145/2000919.2000922. URL: https://doi.org/10.1145/2000919.2000922.

[91] Alexander Ulrich and Torsten Grust. "The Flatter, the Better: Query Compilation Based on the Flattening Transformation". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, May 2015, 1421–1426. ISBN: 9781450327589. DOI: 10.1145/2723372.2735359. URL: https://doi.org/10.1145/2723372.2735359.

[92] *Unity Blog*. https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/. Accessed: 11 March 2020. Unity Technologies, Aug. 2017.

[93] *Unity Engine*. https://unity.com. Accessed: 10 March 2020. Unity Technologies.

[94] *Unreal Engine*. https://www.unrealengine.com/. Accessed: 11 March 2020. Epic Games.

[95] *Unreal Engine 4 – First Look*. https://web.archive.org/web/20120524062935/http://gameindustry.about.com/od/trends/a/Unreal-Engine-4-First-Look.htm. Accessed: 11 March 2020. Epic Games, May 2012.

[96] Guozhang Wang et al. "Behavioral Simulations in MapReduce". In: *Computing Research Repository* abs/1005.3773 (2010). arXiv: 1005.3773. URL: http://arxiv.org/abs/1005.3773.

[97] Walker M. White et al. "Scaling Games to Epic Proportions". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: Association for Computing Machinery, June 2007, 31—42. ISBN: 9781595936868. DOI: 10.1145/1247480.1247486. URL: https://doi.org/10.1145/1247480.1247486.

[98] Georgios N. Yannakakis and John Hallam. "Evolving Opponents for Interesting Interactive Computer Games". In: *From Animals to Animats 8: Proceedings of the 8th International Conference on Simulation of Adaptive Behavior*. Santa Monica, CA, USA: MIT Press, Jan. 2004, pp. 499–508. ISBN: 9780262291446.

[99] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. http://gameaibook.org. New York, NY, USA: Springer, 2018.

[100] Michael Zyda. "From Visual Simulation to Virtual Reality to Games". In: *Computer* 38.9 (Sept. 2005), pp. 25–32. ISSN: 1558-0814. DOI: 10.1109/MC.2005.297.