Database-Supported Video Game Engines: Data-Driven Map Generation

Daniel O'Grady[1]

Abstract: Video game engines can benefit greatly from being tightly coupled with database systems. To make this point and exemplify the similarities in database and game engine technology, we demonstrate a data-driven approach to generate maps for video games, expressed purely in *SQL*. The demonstration will feature such a live database-supported game that is playable on-site.

1 A Marriage of Game and Database Engines

Early video games were conceived as little more than a pastime, but the video game industry has grown immensely since its beginnings in the mid-twentieth century [Ke01]. Video games have since seeped into many aspects of our lifes, such as education, simulations, and serious gaming in our work life. Games further hold a significant economic impact on the world, grossing billions of dollars every year, and employing thousands of workers [Fa18]. Contrasting the humble beginnings of video games, today's games are becoming ever more computationally demanding, with hundreds of players playing the game at once, huge persistent worlds to explore, and large numbers of objects to interact with.
To avoid implementing common components over and over, numerous *video game engines* have been developed throughout the years, which offer commonly required functionality to game developers. Possible game engine components include, but are not limited to: (1) Simulation of physics, (2) collision detection, (3) pathfinding, (4) control of non-player characters (NPCs or "AI"), (5) network communication, (6) video and audio output, (7) processing input from the player, (8) creating and managing game worlds and objects. Most of these components have to deal with large amounts of information that has to be processed rapidly – a true forte of database systems. Unfortunately, databases are predominantly used as little more than dumbed-down persistent storage in the context of video games. This strongly contrasts with the database community's creed to *move the computation closer to the data*. Indeed, in this paper we propose a stronger connection between video games and database systems, by moving parts of a game's internals to the database system, to benefit from database system technology: (1) Selecting a subset of objects is a task where databases excel. This is interesting for finding objects in close vincinity to a player, visual clipping during rendering ("culling"), or collision detection.

[1] University of Tübingen, Department of Computer Science, daniel.ogrady@uni-tuebingen.de

doi:10.18420/btw2019-34

(2) Updating the state of many game objects in bulk, as proposed by Gehrke et al. [Wh07].
(3) Guaranteeing consistent state through transactions.

As we are aware that implementing algorithms in *SQL* may seem foreign to many game developers, we aim to *couple the database tightly with the video game engine* while not exposing developers to the intricacies of database internals.
To exemplify our claim, this demo showcases *declarative map generation*, which is the automatic generation of a playing field for games. The generation either happens before the player dives into the game or when they reach the border of the field to create the illusion of an infinite world. This paper focuses on *real-time strategy* (RTS) games. In RTS games, players control multiple game figures (so-called *units*) on a playing field, the *map*. Units are instructed to collect resources, attack enemy units or just move to a new position. This calls for large open spaces to maneuver units properly, so maps can not just be completely random but need certain properties to be compelling to the player.

2 Data-Driven Iterative Map Generation

The maps we create are comprised of *tiles*, arranged in a two-dimensional grid. Each tile represents a certain type of terrain specific for the game the map is generated for. Such a set of tiles could be *walkable* □, *wall* ■, *coast* ∵, and *water* ≈. Our approach combines these into *modules*, which are predefined clusters of 3×3 tiles.[2] Fig. 1 shows an example of a module representing a piece of horizontal shoreline on the left. The map generation starts with a seed of one module and then *joins* modules to the outer edges repeatedly. The input for the algorithm is a tuple $(\mathcal{T}, \mathcal{M}, C, \mathcal{S})$ with

1. $\mathcal{T}$: a set of tiles,
2. $\mathcal{M}$: a relational representation of modules[3], each consisting of 3×3 tiles $\in \mathcal{T}$,
3. a relation $C \subseteq \mathcal{T} \times \mathcal{T}$ describing the *compatibility* of tiles (explained below) and
4. $\mathcal{S} \in \mathcal{M}$: an initial seed module.

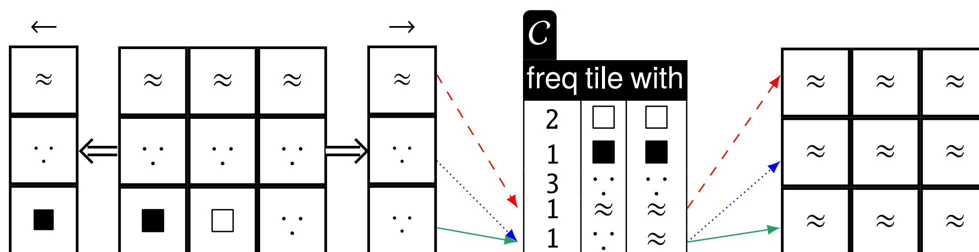
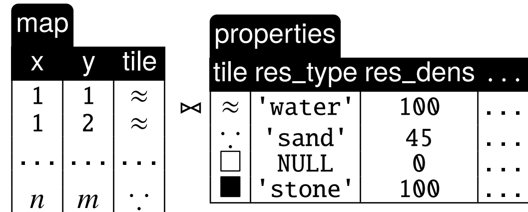


Fig. 1: Two compatible modules. Each pair of neighbouring tiles in the adjacent edges can be joined on the compatibility table C after extracting the edges (only shown for two edges of the left module).

[2] While those dimensions have turned out to be rather convenient for defining modules, any other rectangular dimensions work as well.
[3] Many tabular module encodings are conceivable and we abstract from these here.

```
1  WITH RECURSIVE map(x,y,tile) AS (
2    (SELECT 𝒮)
3    UNION ALL
4    (SELECT ...
5     FROM   C, map, ℳ
6     WHERE
7      C.tile = edge(map) AND
8      C.with = edge(ℳ) AND
9      NOT(<termination condition>)
10 ))
```

Fig. 2: Pseudocode for the map generation. It creates a table `map` of tiles `tile` positioned at coordinates `x,y`. `edge(...)` is a function that produces edges as shown in Fig. 1 on the left.



map

x	y	tile
1	1	≈
1	2	≈
...	...	...
n	m	⋰

⋈

properties

tile	res_type	res_dens	...
≈	'water'	100	...
⋰	'sand'	45	...
□	NULL	0	...
■	'stone'	100	...

Fig. 3: Joining a map of size $n \times m$ on a table of properties. Here, each tile type is associated with a resource type, a resource density, and possibly other information.

The algorithm selects the *edges* of all modules in $\mathcal{M}$ and of the outermost modules of the map together with the direction they are facing. An edge is comprised of the three adjacent tiles of either side on a module. Take Fig. 1, where two of the four edges of the left module are extracted (denoted by $\Leftarrow$ and $\Rightarrow$). Each edge of the map is then joined with a compatible module in $\mathcal{M}$. Two modules are said to be *compatible* if their adjacent edges (the ones facing opposite directions) are compatible in all neighbouring tiles. Tiles, in turn, are compatible with each other if they can be joined on C, as is shown in Fig. 1. An additional value freq per compatibility rule denotes the frequency, which allows us to control how likely a rule is selected in ambiguous situations where more than one compatibility rule qualifies. A formulation of this iterative map expansion in pseudo *SQL* is shown in Fig. 2. Starting with a small seed $\mathcal{S}$, the intermediate result is gradually expanded by recursively joining matching modules to it. Usually, the termination condition limits the map size, but can also be bound to other constraints. This type of iterative map generation is a perfect match for a recursive common table expression.

The data-driven nature of this algorithm enables game developers to easily control this process with their own modules and compatibilities. For example, adding a rule (5, ≈, ■) to C in Fig. 1 would allow water to be generated next to walls. In fact, this would occur more often than, say, water next to water, because of the higher frequency of 5 of the rule.

To finalise the map, it is joined with a tile property table to attach additional semantics that are required for a particular game. This is shown in Fig. 3, where the result of the recursive CTE map is joined with a table properties which contains resource information for each tile type. The map is subsequently converted into a format that is understood by the game engine. Then, actual gameplay commences.

3 Demonstration Setup

The on-site demonstration will showcase the discussed map generation algorithm by tapping into the OpenRA[4] engine, an engine specifically tailored to RTS games. OpenRA is written

[4] https://www.openra.net

in C#, and is still under active development since 2007. The map generation is implemented in pure *SQL* and runs on a *PostgreSQL 10* database system. The process of generating a map is triggered from within a game implemented on top of OpenRA by sending a query to the database and receiving a stream of tuples. Those are then used to create the native data structure provided by the engine, just as if the map was read from disk.[5]

To demonstrate the capabilities of our approach, we will bring canned tilesets with us. These tilesets will have realistic size in terms of modules and compatibilities and will be used to produce maps of typical dimensions for OpenRA, between 40×40 and 200×200 tiles. But the audience is invited to propose changes to the input data to explore how it affects the generated maps. The generation process takes about 60 milliseconds,[6] which allows for rapid re-generation of maps to compare them with each other.[7] We will also bring the source code of the modified OpenRA engine for interested audience members to inspect the details of our implementation and convince themselves that the fingerprint we left on the engine is rather small.

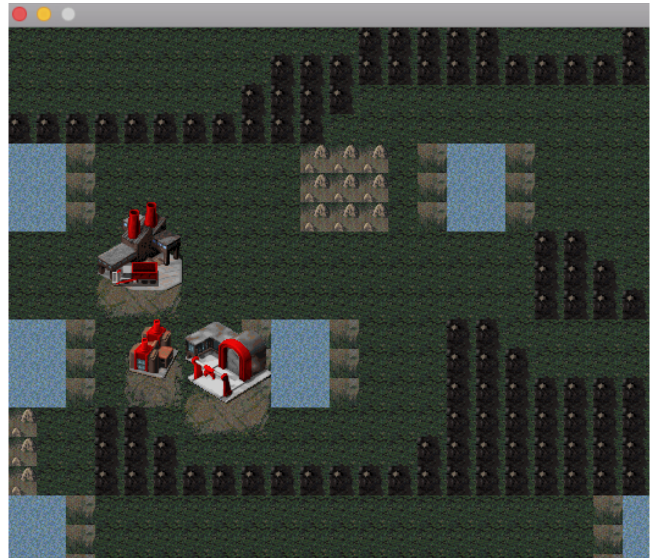


Fig. 4: Screenshot of a generated map with pools of water with shores and enclosed walkable territory. The game will be playable on site.

References

[Fa18] Facts, E.: Essential Facts About the Computer and Video Game Industry, `http://www.theesa.com/wp-content/uploads/2018/05/EF2018_FINAL.pdf`, Accessed: 2018-09-13, 2018.

[Ke01] Kent, S. L.: The Ultimate History of Video Games: From Pong to Pokemon–the Story Behind the Craze That Touched Our Lives and Changed the World. Prima Communications, Inc., Rocklin, CA, USA, 2001, ISBN: 0761536434.

[Wh07] White, W.; Demers, A.; Koch, C.; Gehrke, J.; Rajagopalan, R.: Scaling Games to Epic Proportions. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. SIGMOD '07, ACM, Beijing, China, pp. 31–42, 2007, ISBN: 978-1-59593-686-8, URL: `http://doi.acm.org/10.1145/1247480.1247486`.

[5] In future work, we will seek to eliminate the need for native data structures alltogether.

[6] In addition to the raw generation process, the current implementation takes about 10 seconds to translate the generated data into a native OpenRA data structure. This additional times will disappear once we move more of the game engine logic to the database system.

[7] Tested on a MacBookAir with a 2,2 GHz Intel Core i7, 8 GB 1600MHz DDR3 RAM and a SM0512G SSD.