Data-Driven Map Generation

Daniel O'Grady

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

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

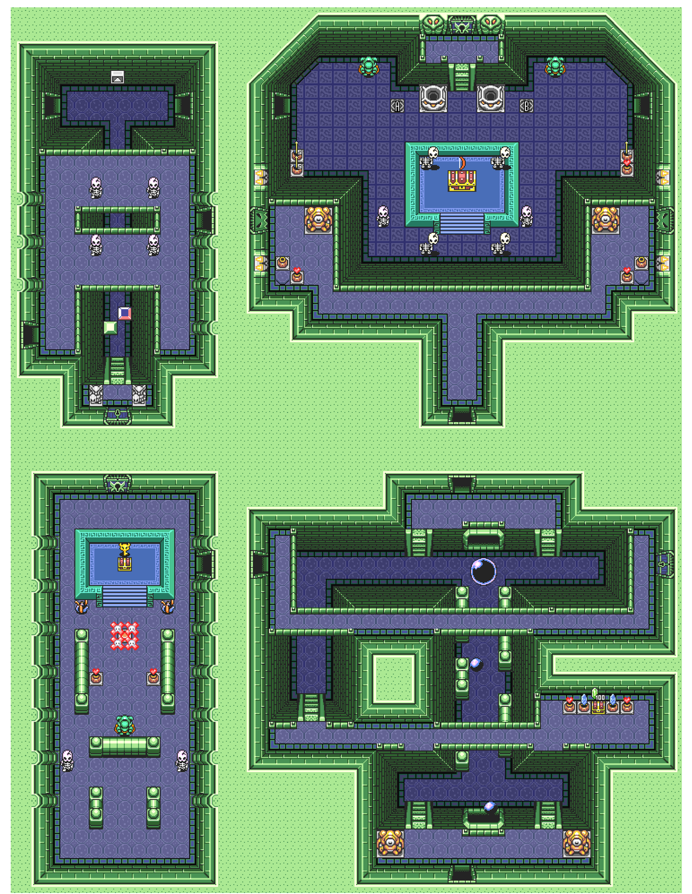
Maps in Video Games

Marrying Database Systems and Video Games Engines

We demonstrate the similarities between problems handled in **video game engines** and **database systems**. Video games deal with rapid computation over vast amounts of data in many areas. We claim that tying in database systems can improve the performance of those areas. To exemplify this, we demonstrate **data-driven map generation**.

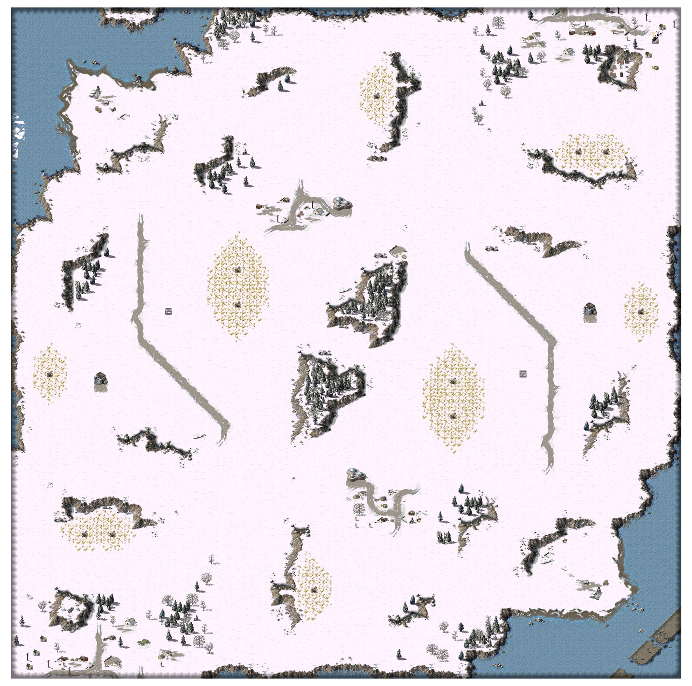
Maps Can Not Be Random Spaces

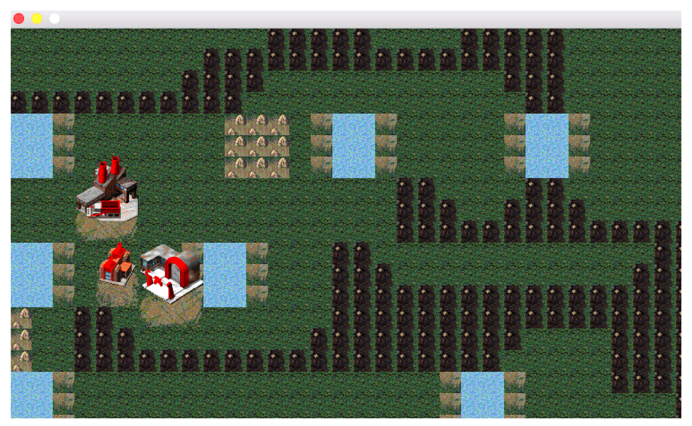
Maps have to cater to the type of game and can not be fully random to be enjoyable. But for each game, **building blocks** can be conceived of which a map can be comprised.



Part of a dungeon from Nintendo's game *The Legend of Zelda: A Link to the Past*. The design is **room based**, where rooms are connected to each other through doors. Rooms are enclosed spaces with functional and decorative elements. Either whole rooms could form building blocks to create random dungeons, or smaller elements, like the pathways on the lower right room form building blocks, to generate random rooms.

Manually crafted map from the game engine *OpenRA*, courtesy of developer *SoScared*. White space is walkable, blue areas are bodies of water, yellow patches are gold, and grey areas are walls or pathways. Note how this design is much more open than the above map of a dungeon and offers plenty of space to maneuver, as it is meant for *Real Time Strategy* games.



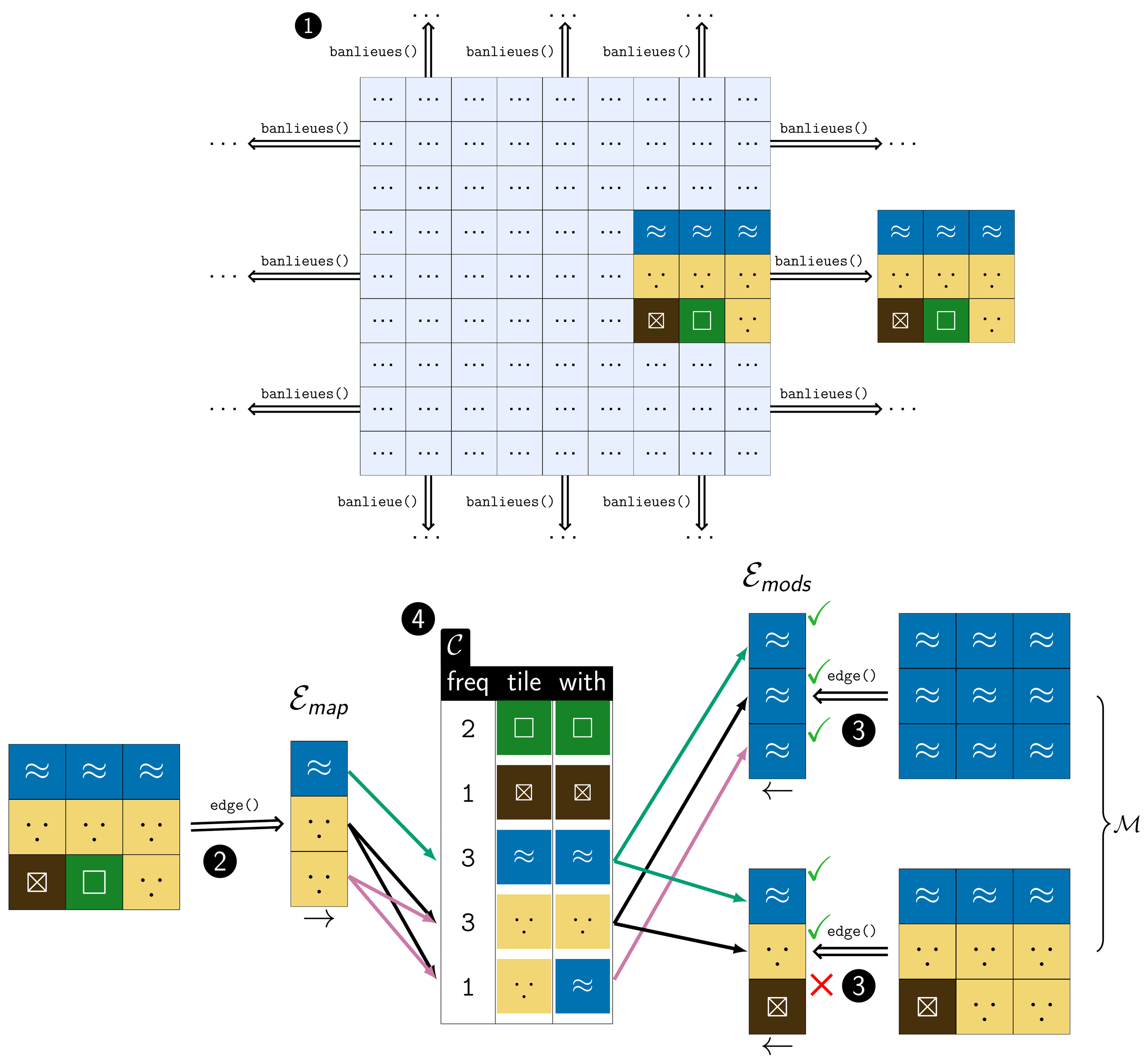


Screenshot of a generated map using our tool. The map generation is directly embedded into the game engine through a lightweight interface and produces playable maps. Note how the map, like the manually crafted one above, features open spaces with walls around and patches of water and resources.

Shaping Maps Through Data – Not Code

Growing the Map

The modules in the following example are composed of the tiles water (≈), coast (∵), walkable (□), and walls (⊠).



Core Algorithm for an Outward Expanding Map

⓿ From a set of available modules $\mathcal{M}$ take one module as seed $\mathcal{S}$. The seed can either be randomly selected or passed as parameter. (This step is not pictured above.)

❶ Select the outermost blocks of the map generated up to this point, using the user defined function `banlieues()`.

❷ Select the outer edges $\mathcal{E}_{map}$ from those blocks, together with the direction they are facing. Again, we are abstracting this process into a UDF `edge()`.

❸ Select the outer edges $\mathcal{E}_{mods}$ from all modules in $\mathcal{M}$, together with the direction they are facing. We can use `edge()` for this, too.

❹ Find edges in $\mathcal{E}_{map}$ and $\mathcal{E}_{mods}$ such that they face opposite directions and can be joined on the table of compatible tiles $\mathcal{C}$. If one edge in $\mathcal{E}_{map}$ can be paired with multiple edges in $\mathcal{E}_{mods}$, pick one at random. The column `freq` controls the probability of a matching row in $\mathcal{C}$ to be used as glue in situations where we can choose between multiple join partners.

❺ Repeat steps ❶ through ❹ until a termination condition has been reached, the default being having reached a certain map size.

Implementing the Algorithm in PostgreSQL Using `WITH RECURSIVE`

```
WITH RECURSIVE map(module,x,y) AS (
  (SELECT 𝒮, 0, 0) ⓿
  UNION ALL
  (WITH
     b AS (TABLE banlieues(map)),  ❶
     ℰ_map AS (TABLE edge(b)),     ❷
     ℰ_mods AS (TABLE edge(ℳ))     ❸
     SELECT DISTINCT ON (coords(ℰ_mods))
        module(ℰ_mods), -- assume that module() is a function
                         -- which restores a module from an edge.
        coords(ℰ_mods)  -- assume that coords() is a function
                         -- which assigns coordinates to
                         -- the module within the map.
     FROM
      ℰ_map
      JOIN 𝒞 ❹
        ON 𝒞.tile = ℰ_map
      JOIN ℰ_mods
        ON 𝒞.with = ℰ_mods
     WHERE
      NOT(⟨termination condition⟩) ❺
     ORDER BY
      freq_sort()  -- assume that freq_sort() is a function that
                   -- randomly sorts elements, but factors in
                   -- the column freq.
))
SELECT * FROM map;
```

Why Stop Here?

More (All!?) Parts of Game Engines Yearning for SQL

Not only map generation can benefit from borrowing from SQL! In future work we will **move more components of the game engine over to the database system**. More candidates that are typical components of a game engine are:

(1) Incremental simulation of physics, i.e. applying trajectory vectors to objects in bulk[a],
(2) collision detection,
(3) pathfinding,
(4) control of non-player characters (*NPCs* or "*AI*"),
(5) determining visual objects during rendering (*culling*),
(6) …

[a] 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.

18. BTW 2019
Universität Rostock · 600 Jahre

Visit us at `http://db.inf.uni-tuebingen.de`