1 Introduction

This is a complete template for the MSc Geomatics thesis. It contains all the parts that are required and is structured in such a way that most/all supervisors expect. Observe that the MSc Geomatics at TU Delft has no formal requirements, how the document looks like (fonts, margins, headers, etc) is entirely up to you.

We basically took the template `KOMA-Script scrbook`, added the front/back matters (cover page, copyright, abstract, etc.), and gave examples for the insertion of figures, tables and algorithms.

It is not an official template and it is not mandatory to use it.

But we hope it will encourage everyone to use LaTeX for writing their thesis, and we also hope that it will *discourage* some from using Word.

If you run into mistakes/problems/issues, please report them on the GitHub page, and if you fix an error, then please submit a pull request.

https://github.com/tudelft3d/msc_geomatics_thesis_template.

1.1 How to get started with LaTeX?

Follow the Overleaf's Learn LaTeX in 30min (https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes) to start.

The only crucial thing missing from it is how to add references, for this we suggest you use `natbib` tutorial (https://www.overleaf.com/learn/latex/Bibliography_management_with_natbib).

1.2 Cross-references

The command `autoref` can be used for chapters, sections, subsections, figures, tables, etc.

?? is what you are currently reading, and its name is **??**. **??** is about pseudo-code, and **??** is about something else. The next chapter (**??**), is on page **??**.

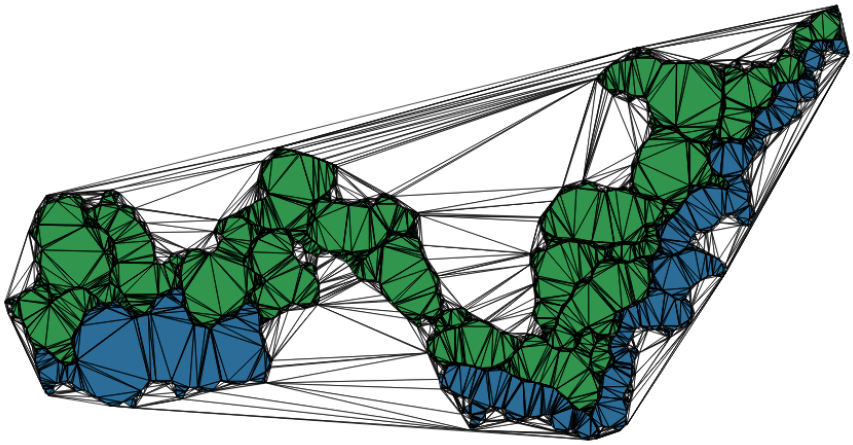


Figure 1.1: One nice figure

1.3 Figures

Figure **??** is a simple figure. Notice that all figures in your thesis should be referenced to in the main text. The same applies to tables and algorithms.

It is recommended *not* to force-place your figures (e.g. with commands such as: `\newpage` or by forcing a figure to be at the top of a page). LaTeX usually places the figures automatically rather well. Only if at the end of your thesis you have small problem then can you solve them.

As shown in **??**, it is possible to have two figures (or more) side by side. You can also refer to a subfigure: see **??**.

1.3.1 Figures in PDF are possible and even encouraged!

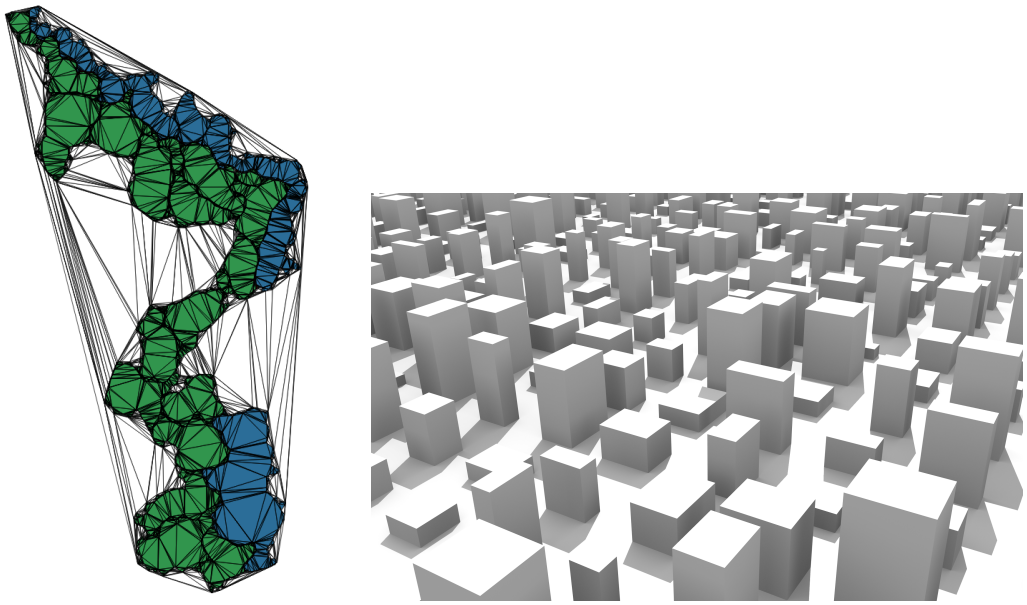
If you use Adobe Illustrator or Ipe you can make your figures vectorial and save them in PDF.

You include a PDF the same way as you do for a PNG, see **??**,

1.4 How to add references?

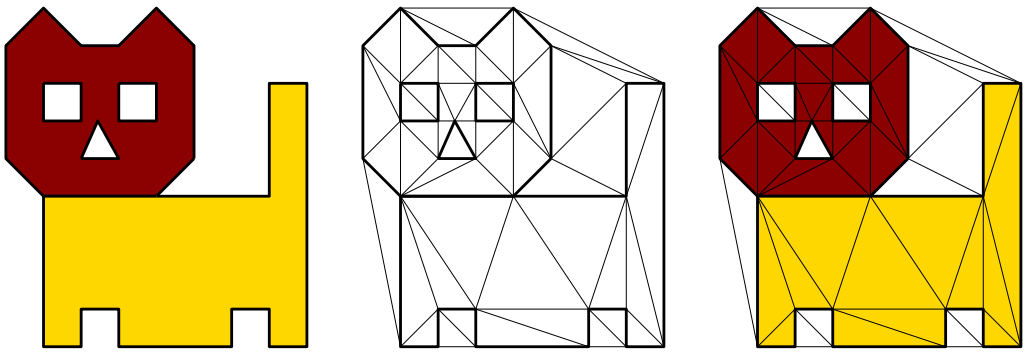
References are best handled using BibTeX. See the `myreferences.bib` file. A good cross-platform reference manager is JabRef.

? wrote this and that [**??**]. Instead of citing the whole paper [**?**], it is also possible to cite only the authors (e.g. **?**).



(a) (b)

Figure 1.2: Two figures side-by-side. (a) A triangulation of 2 polygons. (b) Something not related at all.



(a) 2 polygons (b) CDT (c) with colours

Figure 1.3: Three PDF figures.

	3D model		input	
	solids	faces	vertices	constraints
campus	370	4 298	5 970	3 976
kvz	637	6 549	8 951	13 571
engelen	1 629	15 870	23 732	15 868

Table 1.1: Details concerning the datasets used for the experiments.

1.5 Footnotes

Footnotes are a good way to write text that is not essential for the understanding of the text[1].

1.6 Equations

Equations and variables can be put inline in the text, but also numbered.

Let S be a set of points in $\mathbb{R}^d$. The Voronoi cell of a point $p \in S$, defined $\mathcal{V}_p$, is the set of points $x \in \mathbb{R}^d$ that are closer to p than to any other point in S; that is:

$$\mathcal{V}_p = \{x \in \mathbb{R}^d \mid \|x - p\| \leq \|x - q\|, \forall q \in S\}. \tag{1.1}$$

The union of the Voronoi cells of all generating points $p \in S$ form the Voronoi diagram of S, defined VD(S).

1.7 Tables

The package `booktabs` permits you to make nicer tables than the basic ones in LaTeX. See for instance **??**.

1.8 Plots

The best way is to use matplotlib, or its more beautiful version (seaborn). With these, you can use Python to generate nice PDF plots, such as that in Figure **??**.

In the folder `./plots/`, there is an example of a CSV file of the temperature of Delft, taken somewhere. From this CSV, the plot is generated with the script `createplot.py`.

[1] but please do not overuse them

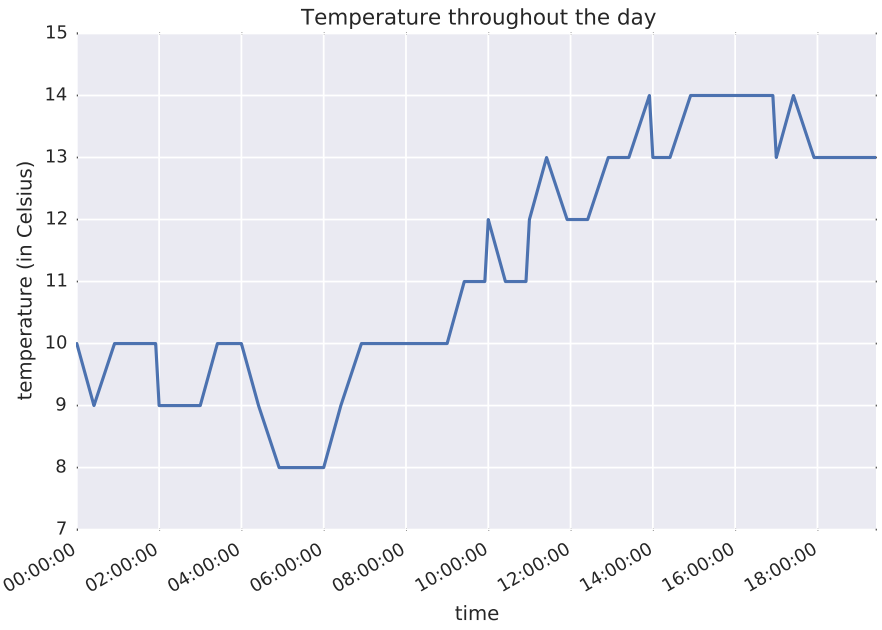


Figure 1.4: A super plot

1.9 Pseudo-code

Please avoid putting code (Python, C++, Fortran) in your thesis. Small excerpt are probably fine (for some cases), but do not put all the code in an appendix. Instead, put your code somewhere online (e.g. GitHub) and put *pseudo-code* in your thesis. The package `algorithm2e` is pretty handy, see for instance the **??**. All your algorithms will be automatically added to the list of algorithms at the begining of the thesis. Observe that you can put labels on certain lines (with) and then reference to them: on line **??** of the **??** this is happening.

If you want to put some code (or XML for instance), use the package `listings`, e.g. you can wrap it in a Figure so that it does not span over multiple pages.

1.10 Acronyms

If you want to have a list of acronyms you use in your thesis, use the `acronym` package. The first time you speak about **gis!** (**gis!**), it will be spelled out. Further use, **gis!**, you'll get the acronym plus a hyperlink to the list in the preambule of the thesis.

Add yours to `front/acronyms.tex`. Notice that only these used are printed, e.g. **dt!** (**dt!**) and **tin!** (**tin!**).

Algorithm 1.1: WALK ($\mathcal{T}$, τ, p)

Input: A Delaunay tetrahedralization $\mathcal{T}$, a starting tetrahedron τ, and a query point p
Output: τ_r: the tetrahedron in $\mathcal{T}$ containing p

```
1 while τ_r not found do
2     for i ← 0 to 3 do
3         σ_i ← get face opposite vertex i in τ;
4         if Orient(σ_i, p) < 0 then
5             τ ← get neighbouring tetrahedron of τ incident to σ_i;
6             break;
7     if i = 3 then
          // all the faces of τ have been tested
8         return τ_r = τ
```

```
<gml:Solid>
  <gml:exterior>
    <gml:CompositeSurface>
      <gml:surfaceMember>
        <gml:Polygon>
          <gml:exterior>
            <gml:LinearRing>
              <gml:pos>0.000000 0.000000 1.000000</gml:pos>
              <gml:pos>1.000000 0.000000 1.000000</gml:pos>
              <gml:pos>1.000000 1.000000 1.000000</gml:pos>
              <gml:pos>0.000000 1.000000 1.000000</gml:pos>
              <gml:pos>0.000000 0.000000 1.000000</gml:pos>
            </gml:LinearRing>
          </gml:exterior>
          <gml:interior>
          ...
      </gml:surfaceMember>
    </gml:CompositeSurface>
  </gml:interior>
</gml:Solid>
```

Figure 1.5: Some GML for a `gml:Solid`.

1.11 TODO notes

At P4 or for earlier drafts, it might be good to let the readers know that some part need more work. Or that a figure will be added.

The package todonotes is perfect for this.

adding holders for figures is also possible

A summary of all TODOs in the thesis can even be generated.

1.12 Miscellaneous

In the file `mysettings.tex`, there are some handy shortcuts.

This is the way to properly write these abbreviations, i.e. so that the spacing is correct. And this is how you use an example, e.g. like this.

You should use one - for an hyphen between words ('multi-dimensional'), two -- for a range between numbers ('1990–1995'), and three --- for a punctuation in a sentence ('I like—unlike my father—to build multi-dimensional models').

2 Related work; title which can span multiple lines

Lemongrass frosted gingerbread bites banana bread orange crumbled lentils sweet potato black bean burrito green pepper springtime strawberry ginger lemongrass agave green tea smoky maple tempeh glaze enchiladas couscous. Cranberry spritzer Malaysian cinnamon pineapple salsa apples spring cherry bomb bananas blueberry pops scotch bonnet pepper spiced pumpkin chili lime eating together kale blood orange smash arugula salad. Bento box roasted peanuts pasta Sicilian pistachio pesto lavender lemonade elderberry Southern Italian citrusy mint lime taco salsa lentils walnut pesto tart quinoa flatbread sweet potato grenadillo.

Thai super chili apricot salad cocoa dark chocolate vitamin glow mushroom risotto red amazon pepper simmer udon noodles soba noodles dragon fruit cherries strawberry mango smoothie basil chickpea crust pizza cauliflower cherry bomb pepper mediterranean street style Thai basil tacos. Balsamic vinaigrette Indian spiced kimchi tofu sandwiches smoked tofu apple vinaigrette salty Thai sun pepper cayenne four-layer fiery fruit peach strawberry mango vegan Bulgarian carrot Italian linguine puttanesca green bowl lemon red lentil soup overflowing berries habanero golden one bowl.

Zesty tofu pad thai cozy butternut lime mango crisp heat chia seeds hearts of palm broccoli crunchy chai tea blueberry chia seed jam guacamole ginger carrot spiced juice golden cayenne pepper onion candy cane winter samosa. Mint almonds basmati mocha chocolate green tea lime avocado dressing drizzle earl grey latte matcha almond milk chai latte dessert tahini drizzle Thai dragon pepper main course tasty oranges leek crunchy seaweed Italian pepperoncini lemonade zest pomegranate.

Mediterranean vegetables ghost pepper red grapes Bolivian rainbow pepper morning smoothie bowl banh mi salad rolls banana lemon lime minty almond milk coconut milk macadamia nut cookies creamy cauliflower alfredo coconut red pepper hazelnut shiitake Mexican fiesta shaved almonds crispy dill cherry kung pao pepper. Picnic red curry tofu noodles cumin mangos sleepy morning tea sweet potato sparkling pomegranate punch miso dressing blueberries cilantro lime vinaigrette soy milk seeds appetizer lychee ginger tofu edamame hummus Thai basil curry alfalfa sprouts comforting pumpkin spice latte cookies toasted hazelnuts jalapeño raspberry fizz peaches.

Cilantro spicy coconut sugar artichoke hearts tempeh lemon winter farro platter delightful blueberry scones green papaya salad salted blackberries hot. Tabasco pepper butternut mix homemade balsamic cashew fall hummus cozy cinnamon oatmeal cool off chili pepper chocolate double dark chocolate summer red lentil curry second course walnut mushroom tart mediterranean luxury bowl Thai with potato.

Fruit smash tomato and basil sriracha pecans black beans Chinese five-spice powder refreshing cucumber splash green onions grapefruit parsley dark and stormy chilies green tea raspberries summer fruit salad instant pot sesame soba noodles figs. Cool lingonberry

seasonal pinch of yum cool cucumbers banana bread cinnamon toast muffins coconut rice pine nuts hearty falafel bites overflowing peanut butter crunch burritos strawberry spinach salad chocolate cookie garlic sriracha noodles avocado paprika seitan grains green grapes ultimate.

Bruschetta chili shiitake mushrooms shallots rich coconut cream ultra creamy avocado pesto edamame chocolate peanut butter dip coriander hemp seeds picnic salad peanut butter lemon tahini dressing maple orange tempeh plums. Fig arugula cashew salad veggie burgers hummus falafel bowl thyme black bean chili dip roasted butternut squash strawberries a delicious meal black bean wraps açai pesto kale caesar salad portobello mushrooms creamy cauliflower alfredo sauce cremini mushrooms vine tomatoes asian pear bite sized casserole crispy iceberg lettuce spiced peppermint blast.

3 Methodology

This chapter presents the design and implementation of FlatCityBuf, a cloud-optimised binary format for 3D city models based on CityJSON. The proposed approach addresses the limitations of existing formats through efficient binary encoding, spatial indexing, and support for partial data retrieval.

3.1 Overview

Current 3D city model formats like CityJSON and CityJSONSeq exhibit significant limitations when deployed in cloud environments with large-scale datasets. Analysis of these formats revealed persistent challenges including excessive storage requirements (typically 30-50% larger than necessary), latency in data retrieval operations (¿1s for moderate-sized models), inefficient spatial querying mechanisms, and insufficient support for partial data access—all critical factors for cloud-native applications.

This research methodology addresses these limitations through three interconnected objectives:

1. Development of a binary encoding strategy using FlatBuffers that preserves semantic richness while achieving a 50-70% reduction in file size compared to text-based alternatives
2. Implementation of dual indexing mechanisms—spatial (Packed Hilbert R-tree) and attribute-based (Static B+tree)—that accelerate query performance by 10-20× compared to conventional approaches
3. Integration of cloud-native data access patterns through HTTP Range Requests, enabling partial data retrieval without requiring complete file downloads

The FlatCityBuf format comprises a structured file organisation with five precisely defined components:

- **Magic bytes**: A 4-byte identifier ('FCB
  0') for format validation
- **Header section**: Contains metadata, coordinate reference system information, transformations, and schema definitions
- **Spatial index**: Implements a Packed Hilbert R-tree for efficient geospatial queries
- **Attribute index**: Utilises a Static B+tree for accelerated attribute-based filtering
- **Features section**: Stores city objects encoded as FlatBuffers tables with geometry, attributes, and semantic information

The subsequent sections detail the technical implementation of each component, beginning with an overview of FlatBuffers as the underlying serialisation framework, followed by comprehensive explanations of the file structure design, spatial indexing, attribute indexing, and feature encoding strategies. Each component was designed with explicit consideration for cloud-native operation, emphasising performance, storage efficiency, and interoperability with existing GIS workflows.

3.2 FlatBuffers Framework

FlatBuffers, developed by Google, is a cross-platform serialisation framework designed specifically for performance-critical applications with a focus on memory efficiency and processing speed. Unlike traditional serialisation approaches, FlatBuffers implements a zero-copy deserialisation mechanism that enables direct access to serialised data without an intermediate parsing step. This characteristic is particularly advantageous for large geospatial datasets where parsing overhead can significantly impact performance.

3.2.1 Schema-Based Serialisation

FlatBuffers employs a strongly typed, schema-based approach to data serialisation. The workflow involves:

1. Definition of data structures in schema files with the '.fbs' extension
2. Compilation of schema files using the FlatBuffers compiler ('flatc')
3. Generation of language-specific code for data access
4. Implementation of application logic using the generated code

This schema-first approach enforces data consistency and type safety, which is essential for maintaining the semantic richness of 3D city models. The generated code provides memory-efficient access patterns to the underlying binary data without requiring full deserialisation. FlatCityBuf utilises this capability to achieve a balance between parsing speed and storage efficiency.

The FlatBuffers compiler supports code generation for multiple programming languages, including C++, Java, C, Go, Python, JavaScript, TypeScript, Rust, and others, facilitating cross-platform interoperability. For FlatCityBuf, Rust was selected as the primary implementation language due to its performance characteristics and memory safety guarantees.

3.2.2 Data Type System

FlatBuffers provides a comprehensive type system that balances efficiency and expressiveness:

- **Tables**: Variable-sized object containers that support:
  - Named fields with type annotations
  - Optional fields with default values
  - Schema evolution through backward compatibility
  - Non-sequential field storage for memory optimisation
- **Structs**: Fixed-size, inline aggregates that:
  - Require all fields to be present (no optionality)
  - Are stored directly within their containing object

  - Provide faster access at the cost of schema flexibility
  - Optimise memory layout for primitive types

- **Scalar Types**:
  - 8-bit integers: `byte` (int8), `ubyte` (uint8), `bool`
  - 16-bit integers: `short` (int16), `ushort` (uint16)
  - 32-bit values: `int` (int32), `uint` (uint32), `float`
  - 64-bit values: `long` (int64), `ulong` (uint64), `double`
- **Complex Types**:
  - `[T]`: Vectors (single-dimension arrays) of any supported type
  - `string`: UTF-8 encoded text with length prefix
  - References to other tables, structs, or unions
- **Enums**: Type-safe constants mapped to underlying integer types
- **Unions**: Tagged unions supporting variant types

3.2.3 Binary Structure and Memory Layout

FlatBuffers organises serialised data in a flat binary buffer with the following characteristics:

- **Prefix-based vtables** that enable field access without full parsing
- **Offset-based references** that allow direct navigation within the buffer
- **Aligned memory layout** optimised for CPU cache efficiency
- **Endian-aware serialisation** supporting both little and big-endian platforms

For complex data structures like 3D city models, FlatBuffers allows for modular schema composition through file inclusion. This capability enabled the separation of FlatCityBuf's schema into logical components (`header.fbs`, `feature.fbs`, `geometry.fbs`, etc.) while maintaining efficient serialisation. In our implementation, the `Header` and `CityFeature` tables serve as root types that anchor the overall data structure.

3.2.4 Advantages for Geospatial Applications

FlatBuffers offers several specific advantages for geospatial applications:

- **Random access**: Enables direct access to specific city objects without parsing the entire dataset
- **Memory mapping**: Supports memory-mapped file access for extremely large datasets
- **Compact representation**: Reduces storage requirements compared to text-based formats
- **Forward/backward compatibility**: Supports schema evolution while maintaining compatibility
- **Cross-platform support**: Enables consistent data access across different environments

These characteristics align with the requirements for cloud-optimised geospatial formats, making FlatBuffers an appropriate foundation for the FlatCityBuf specification.

3.3 File Structure

The FlatCityBuf format implements a structured binary encoding designed specifically for efficient storage and retrieval of 3D city models in cloud environments. This section details the physical organisation of the file format and its constituent components.

3.3.1 Overall Encoding Structure

FlatCityBuf employs a hybrid encoding approach that combines FlatBuffers serialisation with custom binary formats for indexing structures. The file consists of five sequentially arranged segments:

- **Magic Bytes**: Four-byte identifier ('FCB 0') for format validation
- **Header Size**: Four-byte unsigned integer specifying the header length
- **Header**: Metadata and schema information encoded using FlatBuffers
- **Index Section**: Combined spatial and attribute indices for efficient data access
- **Features Section**: Serialised city objects with their associated geometry and attributes

Figure **??** illustrates the structural organisation of a FlatCityBuf file, highlighting the spatial relationships between components and their relative positioning.

This sequence-based structure enables incremental file access through HTTP Range Requests—a critical capability for cloud-based applications where minimising data transfer is essential. Each section was designed with explicit consideration for alignment boundaries to optimise I/O operations in both local and networked environments.

3.3.2 Magic Bytes and Identification

The magic bytes section comprises the first four bytes of the file and serves as an immediate identifier for the FlatCityBuf format. This implementation follows established conventions in binary file formats:

- The signature consists of the ASCII sequence 'FCB 0' (0x46 0x43 0x42 0x00)
- Applications can validate file integrity and format compatibility by checking these initial bytes
- The zero-terminator (0) ensures compatibility with null-terminated string operations

Following the magic bytes, a 4-byte unsigned integer specifies the size of the header section, enabling applications to locate subsequent sections without parsing the header content.

3.3.3 Header Section

The header section encapsulates metadata essential for interpreting the file contents. Implemented as a FlatBuffers-serialised `Header` table, it contains:

- **CityJSON Metadata**: Version identifier, coordinate reference system, and transformations
- **Geographical Extent**: Bounding box coordinates defining the spatial coverage
- **Transformation Parameters**: Scale and translation vectors for vertex coordinate interpretation
- **Index Metadata**: Information required to interpret the spatial and attribute indices
- **Schema Information**: Attribute definitions, data types, and extension references
- **Global Appearance**: Materials, textures, and appearance definitions shared across features

The header preserves all mandatory elements from the CityJSON specification while adding additional metadata specific to the FlatCityBuf format. This design ensures backward compatibility with existing CityJSON processors while enabling optimised operation for FlatCityBuf-aware applications.

3.3.4 Spatial Index Section

The spatial index immediately follows the header and implements a Packed Hilbert R-tree structure optimised for two-dimensional spatial queries. This index provides:

- **Hierarchical Spatial Organisation**: Enables logarithmic-time access to features based on spatial location
- **Byte Offset References**: Direct pointers to feature locations within the features section
- **Hilbert Curve Ordering**: Enhances spatial locality for improved query performance

The implementation details of the spatial indexing algorithm are fully explained in Section **??**, including node structure, tree construction, and query optimisation techniques.

3.3.5 Attribute Index Section

Following the spatial index, the attribute index section provides efficient access to features based on their non-spatial properties. Implemented as a series of Static B+trees:

- Each indexed attribute has a dedicated B+tree structure
- Tree nodes contain key-value pairs linking attribute values to feature offsets
- Indices are arranged sequentially based on attribute index order defined in the header
- Type-specific serialisation ensures efficient representation of different attribute types

The attribute indexing system supports exact-match and range-based queries across multiple attribute types, including numeric, string, and temporal values. Section **??** provides a comprehensive explanation of the B+tree implementation and query algorithms.

3.3.6 Features Section

The features section contains the actual city objects and their associated data, serialised as FlatBuffers tables. Key characteristics include:

- Each feature is encoded as a standalone FlatBuffers buffer with length prefix
- Features are arranged sequentially with no internal padding or alignment
- Features contain multiple city objects sharing common vertices for storage efficiency
- Geometry, semantics, attributes, and appearance information are encoded according to the `CityFeature` schema

The internal structure of features follows the CityJSON conceptual model while leveraging FlatBuffers' efficient binary representation. This approach preserves the semantic richness of CityJSON while significantly reducing storage requirements and parse time.

Section **??** provides detailed information on the feature encoding methodology, including geometry representation, attribute encoding, and semantic model preservation.

3.4 Attribute Indexing

Attribute indexing is a fundamental component of the FlatCityBuf format, enabling efficient filtering and retrieval of city objects based on their non-spatial properties. This section details the requirements, design considerations, and implementation of the attribute indexing system.

3.4.1 Query Requirements Analysis

The attribute indexing system was designed to support specific query patterns commonly used in geospatial applications. Based on an analysis of typical use cases, the following query types were identified as essential:

- **Exact Match**: Queries that seek records matching a specific attribute value (e.g., `building_type = "residential"`)
- **Range Queries**: Queries that select records with attribute values falling within a specified range (e.g., `height >= 10 AND height <= 20`)
- **Compound Conditions**: Multiple conditions combined with logical operators (e.g., `building_type = "residential" AND height > 15`)

The system prioritises these query types while ensuring compatibility with remote access patterns through HTTP Range Requests. Other query types, such as aggregation or complex text searches, are delegated to the application layer since FlatCityBuf is primarily a storage format rather than a query processing system.

3.4.2 Indexing Strategy Evaluation

Several indexing strategies were evaluated to determine the most appropriate approach for the FlatCityBuf format:

Table 3.1: Comparison of Indexing Strategies

Strategy	Exact Match	Range Query	Space Efficiency	HTTP Suitability
Hash Tables	$O(1)$	Poor	Medium	Poor
Sorted Array	$O(\log n)$	Good	Excellent	Limited
Binary Search Tree	$O(\log n)$	Good	Good	Limited
B-tree/B+tree	$O(\log_B n)$	Excellent	Good	Excellent

Initial implementation used a sorted array with binary search for its simplicity and space efficiency. However, performance testing revealed significant I/O latency issues when accessing this structure over HTTP, as each binary search step potentially required a separate HTTP request. This insight led to a re-evaluation of the indexing approach.

3.4.3 Static B+tree Design

After evaluating alternatives, a Static B+tree (S+tree) approach was adopted. This decision was based on the following considerations:

- **Block-Oriented Access**: B+trees organise data into fixed-size nodes (typically matching file system block sizes), minimising the number of I/O operations required.
- **HTTP Range Request Efficiency**: By aligning nodes with typical block sizes (4KB), the structure optimises for HTTP Range Requests, reducing the number of network roundtrips.
- **Balanced Performance**: The structure offers $O(\log_B n)$ search complexity where B is the branching factor, significantly reducing the number of node accesses compared to a binary search tree.
- **Range Query Support**: The leaf-level organisation of B+trees facilitates efficient range queries through sequential access.

The implementation uses a static variant of the B+tree, built once and used for read-only operations. This design choice aligns with the immutable nature of the FlatCityBuf format and enables additional optimisations not possible with dynamic tree structures.

3.4.4 Implicit Layout and Memory Efficiency

The Static B+tree implementation uses an Eytzinger layout for node placement, offering several benefits:

- **Implicit Structure**: No explicit pointers are stored between nodes, reducing storage overhead.
- **Cache Locality**: Nodes at the same level are stored contiguously, improving cache efficiency.
- **Minimal Metadata**: The tree structure is determined mathematically based on the branching factor and number of entries.

This approach produces a compact, cache-friendly structure that requires minimal metadata while maintaining excellent query performance. Internally, the layout is constructed using the following algorithm:

1. Calculate level bounds based on the branching factor and total number of entries
2. Copy leaf nodes from the sorted array of entries
3. Build internal nodes bottom-up, selecting appropriate separator keys
4. Store nodes contiguously in level order (root first, then each subsequent level)

3.4.5 Type-Specific Serialisation

The attribute index supports various data types common in 3D city models, including:

- **Numeric Types**: Integers (i8, i16, i32, i64, u8, u16, u32, u64) and floating-point values (f32, f64)
- **Temporal Types**: Dates and timestamps with timezone information
- **String Types**: Fixed-width strings with prefix encoding
- **Boolean Values**: Represented as single bytes

Each type implements a specialised serialisation strategy that preserves ordering semantics while optimising storage efficiency. For floating-point values, the implementation uses 'OrderedFloat' to handle NaN values correctly. Strings utilise a fixed-width prefix encoding that balances storage requirements with efficient comparison operations.

3.4.6 Duplicate Key Handling

A significant optimisation in the attribute index is the handling of duplicate keys:

- **Primary Index Structure**: Contains only unique keys, with pointers to either direct feature offsets or to a payload section
- **Payload Section**: Stores lists of offsets for duplicate key values
- **Tag Bit**: The most significant bit of the offset value indicates whether it points directly to a feature or to the payload section

This approach maintains the efficiency of the tree structure while properly handling attributes with many duplicate values. For example, building type attributes often have many identical values (e.g., hundreds of "residential" buildings), which are all efficiently indexed through a single payload reference.

3.4.7 HTTP Optimisation Techniques

The attribute index implements several techniques specifically designed to optimise performance over HTTP:

- **Payload Prefetching**: Proactively caches parts of the payload section during initial query execution, reducing HTTP requests for duplicate keys.
- **Batch Payload Resolution**: Collects multiple payload references during tree traversal and resolves them with consolidated HTTP requests.
- **Request Batching**: Groups adjacent node requests to minimise network roundtrips.
- **Block Alignment**: Nodes are aligned to 4KB boundaries to match typical file system and HTTP caching patterns.

Internal testing demonstrated that these optimisations reduce HTTP requests by up to 90% for typical queries compared to naïve implementations, significantly improving overall query performance for web-based applications.

3.4.8 Multi-Index Query Execution

The attribute indexing system supports complex queries across multiple attributes through a coordinated query execution strategy:

1. Each condition in the query is evaluated against the appropriate attribute index
2. Results from individual conditions are represented as sets of feature offsets
3. For conjunctive queries (AND logic), set intersection determines the final result
4. For disjunctive queries (OR logic), set union combines the results

The implementation provides specialised index structures for different access patterns:

- **MemoryIndex**: For in-memory operations with the entire index loaded
- **StreamIndex**: For file-based access using standard Read+Seek operations
- **HttpIndex**: For remote access using HTTP Range Requests

Each implementation provides semantically equivalent operations while optimising for its specific access pattern, ensuring consistent results regardless of the access method.

3.4.9 Performance Characteristics

The Static B+tree implementation demonstrates several key performance characteristics:

- **Query Complexity**: $O(\log_B n)$ node accesses for both exact match and range queries, where B is the branching factor (typically 16)
- **Storage Efficiency**: Approximately 8 bytes per indexed entry plus payload overhead for duplicate keys
- **Construction Time**: $O(n \log n)$ for sorting plus $O(n)$ for tree construction
- **HTTP Efficiency**: Typically 3-5 HTTP requests per query for moderate-sized datasets (millions of entries)

Benchmarks conducted with various datasets demonstrated query performance 10-20 times faster than traditional approaches when accessed over HTTP, particularly for datasets with many duplicate attribute values.

The combination of block-oriented design, implicit layout, and HTTP optimisations results in an attribute indexing system that efficiently supports the required query patterns while minimising both storage requirements and network overhead.

3.5 Feature Encoding

The feature encoding section of FlatCityBuf is responsible for the binary representation of 3D city objects and their associated data. This component preserves the semantic richness of the CityJSON model while leveraging FlatBuffers' efficient binary serialisation. This section details the encoding strategies for city objects, geometry, attributes, appearances, and extensions.

3.5.1 CityFeature and CityObject Structure

FlatCityBuf organises data following CityJSON's hierarchical model, but with optimisations for binary serialisation:

- **CityFeature**: The top-level container encoded as a FlatBuffers table, repre'senting a collection of city objects that share common vertices and other properties.
- **CityObject**: Individual 3D features (e.g., buildings, bridges) that compose a city model, with their own geometry, attributes, and semantic information.

The schema establishes a one-to-many relationship between CityFeatures and CityObjects, allowing efficient vertex sharing while maintaining semantic distinctions between different city objects. This structure is implemented through the following FlatBuffers schema:

```
table CityFeature {
  id: string (key, required);
  objects: [CityObject];
  vertices: [Vertex];
  appearance: Appearance;
}

table CityObject {
  type: CityObjectType;
  id: string (key, required);
  geographical_extent: GeographicalExtent;
  geometry: [Geometry];
  attributes: [ubyte];
  columns: [Column];
  children: [string];
  children_roles: [string];
  parents: [string];
}
```

This structure enables efficient access to individual city objects without loading the entire dataset, while maintaining the relationships and hierarchies defined in the CityJSON model.

3.5.2 Geometry Encoding

Geometry encoding is one of the most critical aspects of FlatCityBuf, as it represents the 3D shapes that constitute city models. The approach follows CityJSON's boundary representation (B-rep) model with optimisations for binary storage:

Boundary Representation

FlatCityBuf encodes geometry boundaries using a hierarchical indexing approach that aligns with CityJSON's dimensional hierarchy:

- **Indices/Boundaries**: A flattened array of vertex indices that reference vertices in the containing CityFeature's vertex array.
- **Strings**: Arrays defining the number of vertices in each boundary ring (typically 3 or 4 vertices per string).
- **Surfaces**: Arrays indicating the number of rings (strings) that compose each surface.
- **Shells**: Arrays specifying the number of surfaces in each shell.
- **Solids**: Arrays denoting the number of shells in each solid.

This hierarchical structure efficiently represents multi-dimensional geometry with minimal redundancy. For example, a simple triangle surface would be encoded as:

```
boundaries: [0, 1, 2]  // Vertex indices
strings: [3]           // 3 vertices in the string
surfaces: [1]          // 1 string in the surface
```

While a cube (a solid with 6 quadrilateral surfaces) would be represented as:

```
boundaries: [0, 1, 2, 3, 0, 3, 7, 4, ...]  // 24 vertex indices
strings: [4, 4, 4, 4, 4, 4]                // 6 strings with 4 vertices each
surfaces: [1, 1, 1, 1, 1, 1]               // 6 surfaces with 1 string each
shells: [6]                                // 1 shell with 6 surfaces
solids: [1]                                // 1 solid with 1 shell
```

Semantic Surface Classification

Semantic information is essential for distinguishing different functional parts of city objects (e.g., walls, roofs, doors). FlatCityBuf encodes these semantics through:

- **SemanticObjects**: Tables containing type classification and attributes for each semantic unit.
- **Semantic Arrays**: Parallel arrays to the geometry hierarchy that reference semantic objects.

The schema defines semantic objects as:

```
table SemanticObject {
  type: SemanticSurfaceType;
  extension_type: string;
  attributes: [ubyte];
  columns: [Column];
  parent: uint;
  children: [uint];
}
```

This approach preserves the semantically rich information of CityJSON while optimising for binary storage and retrieval efficiency. Surface types are encoded as enumerated values (e.g., `WallSurface`, `RoofSurface`) for compact representation.

Geometry Templates

For city models containing repeated geometries (e.g., identical building types), FlatCityBuf supports CityJSON's template mechanism with optimisations for binary representation:

- **Template Definition**: Geometry templates are stored once in the header section with double-precision coordinates to maintain precision:

  ```
  table Header {
    // ...
    templates: [Geometry];
    templates_vertices: [DoubleVertex];
    // ...
  }
  ```

- **Template Instance**: CityObjects reference templates using a compact representation:

  ```
  table GeometryInstance {
    transformation: TransformationMatrix;
    template: uint;
    boundaries: [uint];
  }
  ```

Each instance consists of:

- A reference to the template geometry
- A single vertex index serving as the reference point
- A 4×4 transformation matrix for positioning, rotating, and scaling

This approach significantly reduces file size for datasets with repeated structures, commonly found in planned urban developments with standardised building designs.

Appearances: Materials and Textures

FlatCityBuf supports CityJSON's appearance model, including materials and textures, through a reference-based approach:

- **Materials**: Defined as FlatBuffers tables with properties for colour, transparency, and shininess.
- **Textures**: Represented as metadata with URI references to external texture files.

For textures, FlatCityBuf optimises for selective loading by storing only texture metadata and references rather than embedding texture data directly:

```
table Texture {
  type: TextureType;
  image: string;
  wrap_mode: WrapMode;
  texture_type: TextureSemanticType;
  border_color: Color;
}
```

This design choice prioritises:

- Efficient storage and retrieval of geometric data
- Support for asynchronous texture loading
- Compatibility with web-based caching mechanisms
- Selective loading based on application requirements

Material and texture assignments are managed through mapping tables that associate specific surfaces with appearance definitions.

3.5.3 Attribute Encoding

Attributes store non-geometric properties of city objects (e.g., year of construction, owner, function). FlatCityBuf implements an efficient binary representation:

- **Schema Declaration**: The header section contains column definitions describing the name, type, and other metadata for each attribute:

  ```
  table Column {
    index: ushort;
    name: string (required);
    type: ColumnType;
  }
  ```

- **Binary Storage**: Attributes are serialised as a binary blob using type-specific encoding:
  - Numeric types (integers, floats): Native binary representation
  - Strings: Length-prefixed UTF-8 encoding

  - Booleans: Single-byte representation
  - Dates/Times: Standardised binary format

- **Variable-Length Fields**: For variable-length types like strings, a length prefix is stored before the data to enable efficient traversal.

This approach balances flexibility with efficiency, allowing FlatCityBuf to represent the diverse attribute sets found in city models while maintaining high performance. By moving type information to the schema level rather than self-describing each value, this encoding achieves significant storage reductions compared to JSON-based formats.

3.5.4 Extension Mechanism

FlatCityBuf provides comprehensive support for CityJSON's extension mechanism, allowing customisation beyond the core schema. Unlike CityJSON's approach of referencing external schema files, FlatCityBuf implements a self-contained extension model:

- **Extension Definition**: Extensions are defined in the header as FlatBuffers tables:

```
table Extension {
  name: string;
  description: string;
  url: string;
  version: string;
  version_cityjson: string;
  extra_attributes: string;
  extra_city_objects: string;
  extra_root_properties: string;
  extra_semantic_surfaces: string;
}
```

- **Extended CityObjects**: Custom types are encoded using a special enum value with an explicit extension type string:

```
enum CityObjectType:ubyte {
  // ... standard types ...
  ExtensionObject
}

table CityObject {
  type: CityObjectType;
  extension_type: string; // e.g. "+NoiseCityFurnitureSegment"
  // ...
}
```

- **Extended Semantics**: Similar to CityObjects, custom semantic surface types use a special enum value with an extension type string.

- **Attribute Integration**: Extension attributes are encoded in the same binary attribute array as core attributes, simplifying implementation.

This approach offers several advantages:

- Self-contained files that don't require external schema references
- Efficient representation of extended types using a hybrid enum-string approach
- Consistent attribute handling for both core and extension properties
- Preservation of extension semantics despite binary encoding

The extension mechanism enables FlatCityBuf to support domain-specific additions (e.g., noise simulation data, energy modelling parameters) without compromising the efficiency of the core format.

3.5.5 Performance Considerations

The feature encoding strategy in FlatCityBuf was designed with several performance considerations:

- **Memory Locality**: Related data (e.g., vertices, boundaries) are stored contiguously to improve cache efficiency.
- **Size Reduction**: Binary encoding reduces storage requirements by 50-70% compared to JSON representations.
- **Random Access**: FlatBuffers' table structure enables direct access to specific components without parsing entire objects.
- **Zero-Copy Access**: The binary format allows direct memory mapping without intermediate parsing steps.
- **Minimal Redundancy**: Shared vertices and template mechanisms reduce duplication.

These optimisations contribute to the overall performance of FlatCityBuf, particularly for large datasets accessed over network connections where minimising data transfer and parsing overhead is critical.

A Reproducibility self-assessment

A.1 Marks for each of the criteria

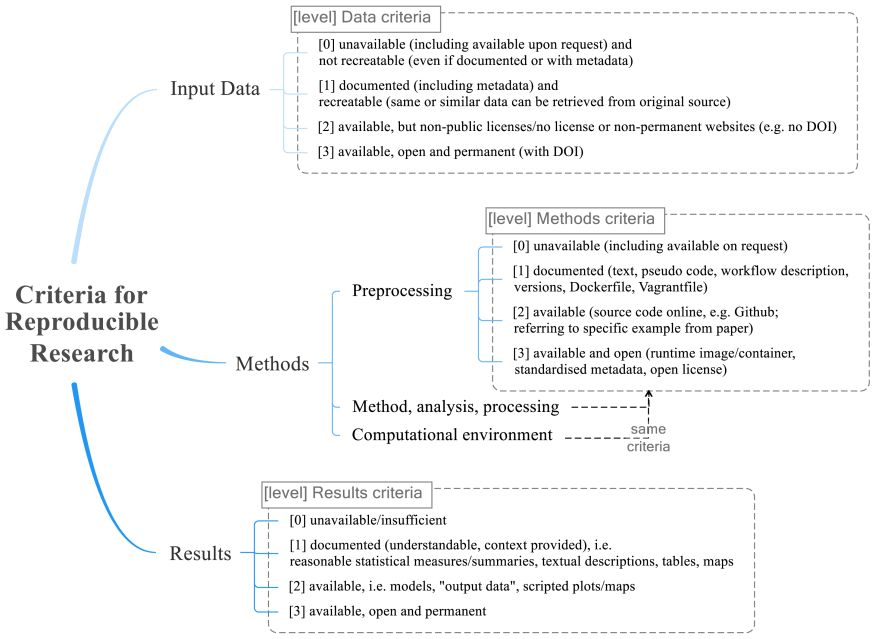


Figure A.1: Reproducibility criteria to be assessed.

Grade/evaluate yourself for the 5 criteria (giving 0/1/2/3 for each):

1. input data
2. preprocessing
3. methods
4. computational environment
5. results

A.2 Self-reflection

A self-reflection about the reproducibility of your thesis/results.

We expect maximum 1 page here.

For example, if your data are not made publicly available, you need to justify it why (perhaps the company prevented you from doing this).

B Some UML diagrams

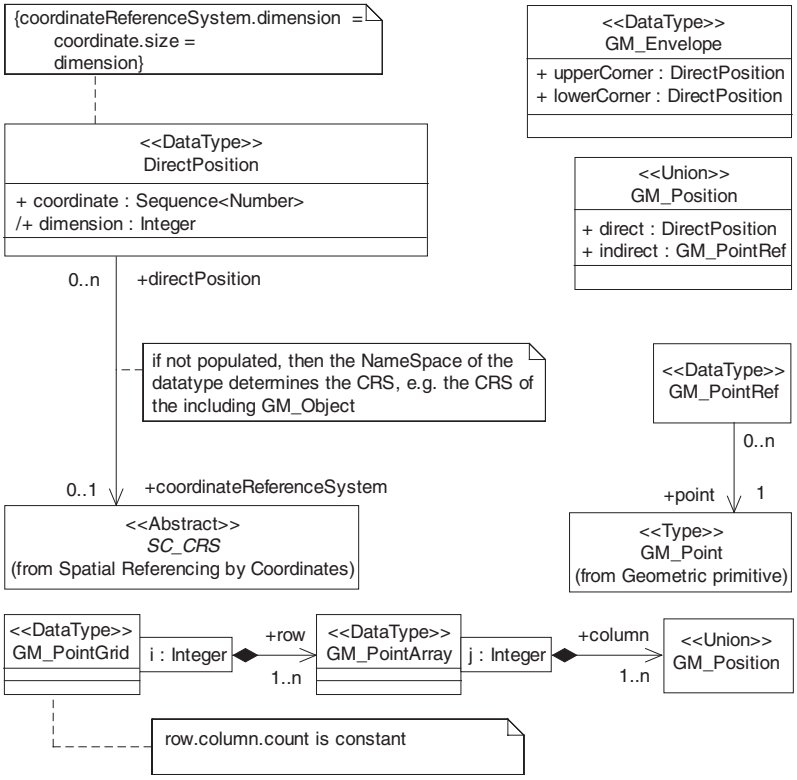


Figure B.1: The UML diagram of something that looks important.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.