CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Scene Graph Memory Management

Minimising Cache Misses in Dynamic Scene Graphs

Master's thesis in Computer science and engineering

Magnus Åkerstedt Bergsten

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Scene Graph Memory Management

Minimising Cache Misses in Dynamic Scene Graphs

Magnus Åkerstedt Bergsten

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Scene Graph Memory Management
Minimising Cache Misses in Dynamic Scene Graphs
Magnus Åkerstedt Bergsten

© Magnus Åkerstedt Bergsten, 2019.

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Advisor: Patrik Ellrén, Carmenta AB
Examiner: Ulf Assarsson, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Scene Graph Memory Management
Minimising Cache Misses in Dynamic Scene Graphs
Magnus Åkerstedt Bergsten
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Due to the growing disparity in performance between memory and processors, it is becoming increasingly important to consider the layout of applications' working set data in memory to make effective use of cache memories and avoid performance bottlenecks from memory latency.

This thesis studies the effects of data layout on scene graphs, a common data structure for organising scenes in graphics applications. Specifically, it studies (1) which way of packing nodes in memory yield the best performance for typical scene graph traversal patterns, and (2) proposes a novel technique for maintaining such a data layout in a scene graph in which nodes are added and deleted.

Three data layouts — orderings for nodes, in which they are packed in memory — are evaluated for static scene graphs: a depth-first order, a breadth-first order, and a *van Emde Boas* layout. These are compared against a "naïve" layout, wherein nodes are individually allocated on the heap.

In a set benchmarks representing typical operations on scene graphs, all data layouts yield similar performance, which is up to three times faster than the naïve layout for large scene graphs. They show very similar performance to the naïve layout for smaller scene graphs, however.

Further, the dynamic memory management system presented in this thesis yields better performance than the naïve layout, in an evaluation simulating a highly dynamic scene-graph application, by up to a factor two for large scene graphs. A limitation with the approach, though, is that memory usage increases on average by a factor of 2.2 in the evaluation.

Keywords: Computer, science, computer science, engineering, thesis, scene graph, memory management, cache friendly, cache efficient.

Acknowledgements

I would like to express my gratitude to Erik Sintorn, my supervisor, who supported and guided the project, recommending directions for my research. I would also like to thank Patrik Ellrén, my advisor at Carmenta, who initially suggested the subject for this project and who provided guidance, input, and support throughout my work.

Magnus Åkerstedt Bergsten, Gothenburg, June 2019

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Background 1

1.2 Purpose 2

1.2.1 Scope and Limitations 2

1.3 Outcome 3

2 Related work **5**

2.1 Cache-efficiency in tree data structures 5

2.1.1 Cache-efficient search trees 5

2.1.2 Cache-efficient layouts for bounding volume hierarchies 6

2.2 Scene-graph optimisation 7

2.3 Survey of open-source scene graphs 7

2.3.1 Self-contained and individually allocated node objects 8

2.3.2 *Ogre3D 2.1*: Node objects with separately managed data . . . 8

2.3.3 *NVidia Pro Pipeline*: Mixed representation 9

3 Theory **11**

3.1 Cache memory 11

3.1.1 Prefetching 12

3.1.2 Cache efficiency 13

3.1.3 Cache models for algorithm analysis 14

3.1.3.1 Cache-aware and cache-oblivious algorithms 14

3.2 Scene graphs 16

3.2.1 Updates 17

3.2.2 Transform propagation 17

3.2.2.1 Breadth-first transform propagation 18

3.2.2.2 Depth-first transform propagation 19

3.2.3 Rendering traversal 20

4 Implementation **21**

4.1 Scene-graph data layouts 21

4.2 Node types 22

4.3 Traversals . . . 24
4.3.1 Transform propagation . . . 24
4.3.2 Rendering traversal . . . 26
4.4 Dynamic scene graph data structure . . . 27
4.4.1 Goals and requirements . . . 27
4.4.2 Data structure design . . . 27
4.4.3 Node-storage memory management . . . 29
4.4.4 Management of meta-data array . . . 30
4.4.5 Maintaining references to relocated nodes . . . 30
4.4.6 The reconstitution operation . . . 31
4.4.7 Configurable parameters . . . 32

5 Evaluation methodology 33
5.1 Scene graph data layout evaluation . . . 34
5.1.1 Dynamic scene graph data structure evaluation . . . 35
5.1.1.1 Scene graph updates . . . 35
5.1.2 Parameters . . . 38

6 Results 39
6.1 Data-layout evaluation for static scene graphs . . . 39
6.2 Dynamic scene graphs . . . 43

7 Discussion 47
7.1 Limitations of the dynamic data structure . . . 47
7.2 Other approaches . . . 47
7.3 Ethical considerations . . . 48
7.4 Future work . . . 48
7.4.1 Traverse in breadth-first order . . . 48
7.4.2 Have a maximum size for slices . . . 48
7.5 Conclusion . . . 49

Bibliography 51

List of Figures

3.1 A binary tree in a *van Emde Boas* data layout. 15

4.1 An example of a tree in a *depth-first order* layout (DFS layout). . . . 21
4.2 An example of a tree in a *breadth-first order* layout (BFS layout). . . 22
4.3 An example of a node-storage data region with space lost to fragmentation. 29
4.4 An illustration of the reconstitution operation. 31

6.1 Results of the transform-propagation benchmark. 40
6.2 Results of the rendering-simulation benchmark. 41
6.3 Results of the combined-traversals benchmark (both transform propagation and rendering simulation). 42
6.4 Results of the dynamic transform-propagation benchmark. 44
6.5 Results of the dynamic rendering-simulation benchmark. 44
6.6 Results of the dynamic combined (frame-simulation) benchmark. . . . 45
6.7 Storage overhead of dynamic-DFS (slice-based) data structure compared with naïve solution. 45

List of Tables

4.1 Size of node types in bytes on the evaluated platforms. 24

5.1 The hardware and software platforms upon which evaluation was performed. 33
5.2 Distribution of node types in the benchmark scene graphs. 34
5.3 Scene graph sizes, number of benchmark iterations, and data set size for all benchmarks. 35
5.4 Parameters for the dynamic scene graph data structure and their chosen values. 38

Glossary

BFS layout A data layout for graphs where the nodes are stored in a sequence matching the order in which a breadth-first search would visit them.

BVH Bounding-volume hierarchy.

cache Cache (more specifically, cache memory) refers to small but fast memories residing between the CPU and main memory, which provide faster access to a subset of the current working set data.

cache line A contiguous block of memory, the unit in which memory is transferred between levels in a memory hierarchy.

cache miss An access to a memory location not currently in the cache.

data layout The way a (subset of) an application's working set data is laid out in memory; in this report, this term is used to discuss the layout of scene-graph nodes in memory.

DFS layout A data layout for graphs where the nodes are stored in a sequence matching the order in which a depth-first search would visit them.

directed acyclic graph Directed graphs in which there are no cycles; i.e. there is no path from a node to itself.

hierarchical memory A model of memory where there are multiple memory levels organised in a hierarchy, for example, multiple levels of cache and a main memory.

prefetching The process of moving cache lines into the cache before they are requested. Exists both as hardware prefetching and as software prefetching.

random-access memory A model of memory in which all memory accesses are assumed to be of equal cost.

real time A requirement on the timeliness of results in a system: timeliness is considered necessary for correctness.

scene graph A hierarchical data structure (tree or directed acyclic graph) representing objects to be rendered in a scene.

vEB layout A data layout for trees where nodes are laid out according to a recursive structure, with the characteristic that root-to-leaf traversals have provable bounds on number of cache misses.

1 Introduction

In computer graphics applications, data structures are used to organise virtual scenes. In the simplest case, a scene could be described by an array containing objects to be rendered. A more complex but common approach is the *scene graph*: a data structure describing the objects in a scene as a hierarchy, where properties are propagated from parent to child. Good performance is required for such data structures to prevent graphics applications from being limited by CPU and memory speeds.

To achieve good performance in a data structure, one should consider not only the algorithmic complexities of its operations, but also factors such as memory-access costs. Memory-access performance can vary wildly depending *data layout* — how data is organised in memory. This is because memory is organised in a hierarchy: there is a small memory called the *cache* in between the CPU and main memory, the former of which is much faster than the latter. Memory accesses that hit the cache allow the program to proceed much faster than otherwise. If the data is laid out in a cache-efficient manner, more memory accesses will hit the cache than they would otherwise.

This thesis project presents research on data layouts with the goal of reducing cache misses in scene graphs. A set of data layouts are evaluated for static scene graphs, and an experimental solution for maintaining such layouts in a dynamic scene graph is described and evaluated.

This project is done in collaboration with Carmenta AB.

1.1 Background

Much has been written in recent years on how data structures and algorithms may be designed and analysed for *hierarchical memory*, in which memory is organised in layers of increasing size and decreasing speed. In practice, these levels consist of cache memories residing between the CPU and main memory. This differs from the conventional *random-access memory* model for algorithm analysis in that memory accesses are not all assumed to be of equal performance cost [4]: accessing memory from a cache is significantly faster than accessing main memory (or even secondary storage). The motivation for this research is the ever-widening gap between processor and memory performance; memory latency has become a bottleneck for an increasing number of applications [8].

This applies especially to real time graphics applications. As graphics processing units have become faster, it has become increasingly important to ensure that applications make efficient use of CPU and main memory. Otherwise, scene complexity and fidelity may become limited by — for example — main memory latency, while much of the graphics processing capacity is left unused.

The *scene graph* is a central data structure in many graphics applications. A scene graph is a hierarchical representation of objects and properties thereof in a virtual scene [2, Chapter 19]. A conventional implementation technique for scene graphs is to store nodes at arbitrary locations in main memory. The graph is traversed by following pointers between node objects, which (especially for large and dynamic scene graphs) is an ineffective use of cache memories. This results in a large amount of *cache misses*, where the CPU has to wait for the requested data to be transferred from main memory to the cache memories [8].

While cache-efficient implementations of some data structures[1] have been studied in detail, there is, to our knowledge, no material in the literature on cache-efficient scene graphs in particular. The wide-spread use of scene graphs in graphics applications and the high performance requirements of those systems provide ample motivation for such research.

1.2 Purpose

The aim of this project is to research, implement, and evaluate memory-management techniques for dynamic scene graphs for modern CPUs, with the goal of minimising performance bottlenecks caused by memory access costs. The research questions are:

1. how to lay out nodes in memory to minimise cache-miss costs for typical scene-graph use cases, and
2. how to maintain such a data layout as the scene graph is modified.

1.2.1 Scope and Limitations

Scene graphs have varying characteristics and use cases for different applications. They are typically either trees or directed acyclic graphs [2, Chapter 19], but there is no agreed-upon topology in the definition of the term “scene graph”. To specify a distinct subject for this project, the following limitations are made:

- The implementation is restricted to a tree topology.

[1]See Chapter 2, *Related work*.

- The implementation is intended and evaluated for applications in which traversal follows a pattern typical of scene graph applications (as explained in Section 3.2).
- The implementation is designed as an in-memory data structure and is targeted at and evaluated using commodity hardware.
- Focus lies on *soft real-time* graphics applications: applications where timeliness of results (consistency of frame rate) is important, but occasional untimeliness results in degraded quality, not outright incorrectness [30].
- Empirical measurements of execution time is the main evaluation criteria, but theoretical asymptotic cache-efficiency analysis[2] is in some cases performed to aid analysis.

1.3 Outcome

For each investigated scene-graph usage pattern — scene graph update, transform propagation, and rendering traversal — a benchmark application was created along with a "frame emulation"-benchmark which combines all three, executed subsequently, to emulate the usage pattern of a scene graph during a single frame of a graphics application. The benchmarks were implemented to perform the traversals in different ways to evaluate the effects of various combinations of scene-graph memory layouts and traversal patterns.

For static scene graphs, depth-first order and breadth-first order data layouts were found to be most efficient with very comparable performance, up to three times faster in large scene graph than a "naïve" data layout. A "van Emde Boas" data layout provided performance very close to the depth-first and breadth-first layouts. Performance difference varied with graph size; for small graphs, memory layout had a smaller impact on performance.

The dynamic scene graph data structure was up to 60% faster than the naïve solution for large scene graphs, but provided little benefit for smaller ones. Additionally, it requires significantly more memory than the naïve solution. This suggests that there is value in the approach of the dynamic data structure, but further research is required to improve it.

[2]See Section 3.1.3 for an explanation of asymptotic cache-complexity analysis.

2

Related work

This chapter presents previous work on related subjects and discusses those results' applicability to the research questions considered in this report. It presents both published scientific work and an analysis of techniques used in practice via a survey on open-source scene-graph code bases.

2.1 Cache-efficiency in tree data structures

There are several results on cache-efficient tree data structures in the cache-aware and cache-oblivious models (described in Section 3.1.3) [27, 4]. However, these largely focus on search trees such as B-trees; as a result of that, they also focus on particular traversal patterns — usually root-to-leaf search paths — which do not correspond exactly to those of scene graphs (as is discussed in Section 3.2). A selection of such results are presented below.

2.1.1 Cache-efficient search trees

In 1999, Prokop introduced the notion of *cache-oblivious algorithms* [27]. One of the presented results is the *van Emde Boas* data layout for binary search trees[1]. This layout has provable worst-case bounds for the number of cache-lines traversed on any root-to-leaf path. For an explanation of cache obliviousness and the van Emde Boas layout, see Section 3.1.3. This method is, however, limited to static complete binary trees.

Bender et al. generalised the concept of cache-oblivious trees as introduced by Prokop, presenting a cache-oblivious *dynamic* B-tree [5]. This tree relies on strongly weight-balanced update algorithms to maintain the van Emde Boas data layout during node insertions and deletions. They present asymptotic bounds on the number of cache-block transfers for search traversals, insertions, and deletions.

Several further results on dynamic B-trees simplified the aforementioned approach while maintaining the same asymptotic bounds on cache-block transfers. One such

[1]The van Emde Boas data layout is named for its parallels to a data structure presented by P. van Emde Boas [9].

example is the *density-based dynamic B-tree* of Brodal et al. [7]. A very similar result, with the same complexity bounds, was independently discovered by Bender et al. [6]. Their tree data structure works by embedding the dynamic binary tree of height $\log N + O(1)$ (where N is number of nodes) within a static, complete binary tree of height H. The static tree is, in turn, laid out in an array using the van Emde Boas data layout. The dynamic tree is kept at a small height — so that it may fit inside the static tree — using update algorithms that re-balance the tree to minimise its height.

The aforementioned results diverge from the requirements of scene graphs in two ways: firstly, they are optimised for the particular traversal patterns of root-to-leaf searches rather than the traversal patterns of typical scene-graph use cases[2]; and secondly, they work on the assumption of balanced trees with a fixed branching factor, whereas scene graphs can have arbitrary tree topology and may be unbalanced.

2.1.2 Cache-efficient layouts for bounding volume hierarchies

A result somewhat close to the subject of this project was presented by Yoon and Manocha [34]. They present an algorithm that computes cache-efficient data layouts for bounding volume hierarchies (BVH, hierarchies of geometry-enclosing volumes used to accelerate collision queries, e.g. ray-to-volume collision for ray-tracing). The algorithm uses an analysis of BVH traversal patterns, which includes *spatial locality*, *parent-child locality*, and a geometrically-based probabilistic model, pre-computing the probabilities that a query visiting one node will also visit each of its children. This is used to compute a *static* data layout for the nodes in the tree.

This result has some applicability to scene graphs. First, scene graphs sometimes embed a BVH[3], and traversals (e.g. for rendering) may be guided by a collision query such as view frustum against scene elements, to only traverse visible nodes. Further, the considered BVH — unlike the trees in the results presented above — does not have a fixed branching factor. Finally, in a BVH, traversals also often visit many more nodes than only those on a root-to-leaf search path (much like in a scene graph). However, the algorithm produces a static data layout and requires significant processing to do so; as such, it is not directly applicable to the subject of this project.

Another result on cache-efficient BVH layouts — a cache-oblivious R-tree — is presented by Arge et al [3]. This, too, is an algorithm for determining a static data layout. Further, R-trees have particular topology requirements which do not apply to scene graphs.

[2]Such traversals are described in Section 3.2.

[3]E.g. by storing, in each node, a variable with the minimum radius that encloses all objects within the sub-tree rooted in that node.

2.2 Scene-graph optimisation

Roth et al. present an approach for allowing multiple threads to work on a shared scene graph using data replication and synchronisation techniques [28]. They consider the scene graph's data layout with the goal of avoiding destructive interference due to multiple threads writing to the same cache line, but focus lies on providing thread-safe concurrent mutable access to a shared scene graph.

Wörister et al. describe a scene-graph rendering system which aims to achieve high performance by maintaining rendering caches, which are efficient representations of the rendering instructions required to draw entire scene graph sub-trees [32]. They present a method for updating the rendering caches without traversing the scene graph through the use of graphs modelling the dependencies between the rendering caches and scene graph components.

The approach of Wörister et al. can be compared to that of the *NVidia Pro Pipeline* scene graph, which is described in the next section. Both of these have as a goal to eschew scene graph traversal in favour of maintaining separate scene representations which are kept in sync with the scene graph.

2.3 Survey of open-source scene graphs

Due to the scarcity of published research on memory management in scene graphs, a survey of open-source code bases containing scene graphs was undertaken. This provides some insights into how the problem has been handled in various projects before.

The following code bases were studied, each at their most recently published versions as of February 2019:

- *Ogre3D* (versions 1.11 and 2.1) [23]
- *OpenSceneGraph* [26]
- *VulkanSceneGraph*[4] [31]
- *Scenix* [21]
- *NVidia Pro Pipeline* [20]
- *Irrlicht* [15]

A noteworthy fact is that some of these code bases have, in later revisions, been reworked to be more cache-efficient. For example, the data layout differs vastly between versions 1.11 and 2.1 of *Ogre3D*. This was done to improve graph traversal performance with large scene graphs [24, 22]

[4] *VulkanSceneGraph* was in an early prototype phase at the time of surveying.

The code bases exemplified a variety of techniques for the implementation of their graph data structures and the associated memory management. The techniques are broadly classified in the following sections.

2.3.1 Self-contained and individually allocated node objects

This is the most conventional and straightforward object-oriented implementation technique. Node classes contain all per-node data and represent edge relationships using pointers to other nodes. Variations on this scheme include reference-counted lifetime management of nodes and slightly different memory allocation schemes.

Surveyed code bases that fall under this classification are:

- *Ogre3D* version 1.11 (but not version 2.1)
- *OpenSceneGraph*
- *VulkanSceneGraph*
- *Scenix*
- *Irrlicht*

OpenSceneGraph, *VulkanSceneGraph*, and *Irrlicht* are fairly straightforward examples of this implementation technique. *Scenix* works in much the same way, but uses a special "chunk memory allocator" that places small objects within pre-allocated blocks of memory, which may have some positive impact on the cache-efficiency of the scene graph. These four all use reference-counted lifetime management for nodes.

VulkanSceneGraph, despite being a separate project that is developed from scratch, maintains the same general graph implementation technique as *OpenSceneGraph*. It has, however, been optimised for cache efficiency by reducing the size of each node by moving non-essential data out of the node classes and into optional "auxiliary" classes, referred to by an optional pointer in the node. *VulkanSceneGraph* also includes an interface allowing the user to provide their own memory allocation implementation for its object types, but uses the standard C++ dynamic memory allocator by default.

Ogre3D version 1.11 also falls into this classification, but does not use reference counting. The lifetimes of node objects are bound to their owning *SceneManager*, and may be manually destroyed by the client code.

2.3.2 *Ogre3D 2.1*: Node objects with separately managed data

Ogre3D has had its scene graph data representation vastly redesigned in version 2.1. While the API still presents a similar structure to that of earlier versions, the actual data storage is entirely different and optimised for cache-efficient traversal.

The *Ogre3D 2.1* data layout can be described as "structure-of-arrays with a conventional node graph interface". In practice, all the data is stored in a collection of arrays of homogeneous types (e.g. translation vectors). The nodes in the scene graph store pointers to each of their associated data values in these arrays. As such, they can be used to read and modify the contents of the data arrays. There is one data array per depth level in the scene-graph hierarchy.

Hence, the actual graph data structure implementation consists of a set of arrays, one for each kind of attribute in the nodes (structure-of-arrays representation), including parent-child relationships. Some types of traversals can be done very cache-efficiently with this layout by iterating over these arrays instead of following pointers between nodes. The conventional node objects act as a more convenient (but less efficient) interface to query and manipulate the actual array-based data representation.

The data arrays are managed by an *array memory manager* each, which allocates slots within its array upon request. The array memory managers may also defragment their arrays. This is possible since the array memory managers keep track of to which node in the scene graph each array slot belongs. After defragmenting, it notifies the node to update its data pointers to the new location. This is done when enough slots have been de-allocated, leaving holes in the array.

One final aspect of the data layout is the fact that vector values in the data arrays are "packed" in units fitting the target platform's vector-register size. As an example, a 3D translation vector may be represented in groups of four as `XXXX YYYY ZZZZ`, instead of as `XYZ XYZ XYZ XYZ`. Traversal over the graph (for example, to calculate absolute transform values[5]) processes four nodes at a time and use single-instruction multiple-data operations to process all four at once. In a conventional object-oriented graph representation, this would scarcely have been possible, which illustrates the ability of careful data layout to enable optimisations beyond that of cache efficiency.

2.3.3 *NVidia Pro Pipeline*: Mixed representation

As with *Ogre 2.1*, the *NVidia Pro Pipeline* scene graph has been designed to reduce CPU and memory overhead. The project takes quite a different approach, however. Instead of optimising the traversal of the scene graph, it has been designed to minimise the amount of traversal required.

NVidia Pro Pipeline maintains multiple data representations of the scene: a conventional directed-acyclic graph, a strictly tree-shaped graph, and a linear sequence of renderable objects. The former of these stores most of the required data and is the application's interface for modifying the scene. The other two representations are automatically updated to reflect changes to the scene graph using notifications built using the observer pattern [12].

[5]This process is described in Section 3.2.2.

The tree-shaped graph is an “expansion” of the scene graph in the sense that sub-graphs with multiple parents are copied as separate nodes under each of the parents. The nodes in the tree are, however, much smaller than in the scene graph, referencing the data stored therein rather than copying it all. The goal of keeping both of these scene representations is to get the benefits of both approaches: sub-graphs can be re-used to reduce redundancy and memory overhead, and having a unique path from any node to the root (as in the tree) helps with caching intermediate results in nodes.

3 Theory

This chapter presents underlying theory. It explains how cache memories work, how software may be optimised and analysed for cache efficiency, and what scene graphs are and how they are used.

3.1 Cache memory

Memory speed has become an increasingly central bottleneck in computer performance during the last decade, as CPU speeds have increased at a higher rate than memory speeds [13]. As a result of this, the time spent by a CPU waiting for data has increased in proportion to the time spent performing useful operations, limiting performance.

This performance discrepancy — the *processor-memory performance gap* or *memory wall* [33, 18] — has been partially ameliorated through the use of *cache memory* in CPUs, which allows the CPU to work with a subset of the memory data in a faster but smaller memory [13, 8, 10]. There may be one or more layers of cache between the CPU and the main memory, constituting a *memory hierarchy*, where increasing levels are larger in capacity but slower to access. The smallest and fastest cache level is called the *L1* cache, the next level is called *L2*, and so on. The number of cache levels varies with processor, but two or three levels of cache are common.

The L1 cache is typically divided into an *instruction cache* for the CPU instruction stream and a *data cache* for application data. This project investigates a data structure from a cache-efficiency perspective, hence we focus on the data cache.

In multi-core systems, it is common for the cores to share lower-level (larger) caches but for each core to have its own higher-level (smaller) cache. As an example, on an *Intel i7-8700* CPU (as used in the evaluations for this thesis), there are three cache levels. Each core has its own L1 and L2 cache of sizes 64 KiB and 256 KiB respectively[1], and all cores share a 12 MiB L3 cache.

The memory address space is divided into a set of *cache lines* (or *cache blocks*) of equal size[2]. Data from main memory is copied into the cache (and written back to

[1]The L1 cache is divided into a 32 KiB L1i instruction cache and a 32 KiB L1d data cache

[2]A typical cache-line size is 64 bytes [8, 11].

memory) in units of cache lines. When a cache line is copied from main memory to the cache, another line may need to be *evicted* from the cache to make space.

When the CPU needs to access a certain memory location, the cache memory is first checked; if the required data is in the cache, there is a *cache hit*: the program can continue executing faster than if it was not — a *cache miss* — as that would require the cache line containing the data to be fetched from main memory, a much slower operation [13].

Each cache-line is mapped to a certain block in cache memory. The mapping between memory location and cache line is usually performed by considering only some of the lower-valued bits of a memory address value [13].

Since the main-memory address space is far larger than the cache memory, such a mapping scheme would mean that there will be many different addresses in memory that map to the same block in the cache. If an algorithm alternately accesses two memory locations a and b such that they both map to the same cache line, each access to a would have to evict the cache line holding b and vice versa, nullifying the benefit of having a cache. To resolve this, caches usually have some level of *associativity*. *N-way set associativity* means that for any cache line, there are N locations in the cache to which it may be mapped. For example, for a cache with two-way set associativity, two cache lines with the same mapping (as given by their addresses' lower-valued bits) may be in cache simultaneously. In contrast, a cache wherein each memory location may only correspond to one location in cache is called a *direct-mapped* cache.

When a cache line is loaded in a N-way set associative cache, a choice must be made of which of the N potential lines in cache to evict. This selection is typically done by evicting the *least-recently used* of the cache lines.

3.1.1 Prefetching

Processors use certain heuristics to predict which memory locations are likely to be used in the near future, and preemptively copy the relevant cache lines from main memory into the caches in a process called *hardware prefetching*.

One common type of hardware prefetching is called *stream prefetching*. Stream prefetching maintains a *stream buffer* of cache lines which is separate from the cache. When a cache line a is requested, causing a cache miss, the stream prefetcher may be triggered; it would then load lines $a+1, a+2, a+3, \ldots$ into the stream buffer. If cache line $a+1$ is requested next, it can be served from the stream buffer instead of from main memory, resulting in faster memory access. The head of the stream buffer would then advance to $a+2$, and so on [16, 8, 19].

Stream prefetching is typically triggered by a series of cache misses within some CPU-specific *hardware-cache prefetch trigger distance* [8]. Some hardware prefetchers detect the stride between requested cache lines to skip prefetching lines that will not be used. For example, if a program requests cache lines $a, a+4, a+8, a+12, \ldots$, then the prefetching logic may detect that the stride is 4 cache lines and prefetch accordingly; this is referred to as a *stride prefetcher*. CPUs are often able to track multiple streams (with differing stride) and can contain multiple stream buffers [16, 8, 14].

Another type of hardware prefetching is *adjacent-line prefetching* (also known as spatial prefetching) [14]. This simply means that each cache line is considered to belong to a pair together with an adjacent cache line; when a cache miss triggers the loading of one cache line in such a pair into the cache, the other is prefetched. For example, requesting a memory location in a cache line a might also prefetch $a + 1$ into the L2 cache.

The *Intel 64 and IA-32 Architectures Optimization Manual* describes the hardware prefetchers in recent Intel CPUs, which include stride prefetching and adjacent-line prefetching [14].

In addition to hardware prefetching, there is also *software prefetching*. This consists of CPU instructions which may be included in software to instruct the CPU to prefetch specific cache lines to one or more cache levels [19, 17, 14]. These instructions may be manually inserted by a programmer or automatically inserted by a compiler.

3.1.2 Cache efficiency

While cache memories are designed to be efficient for typical memory-access patterns in software, it is still important for programmers to be aware of the memory caches and to structure programs' working-set data and the associated memory accesses — the programs' *memory-access patterns* — in a way that minimises cache misses. Doing so entails programming to maximise *locality of reference*: data used together should be stored together, which helps as it increases the probability that a memory access refers to a memory location that is already in cache (or that the memory accesses follow predictable patterns, making effective use of hardware prefetching).

There are two types of locality relevant to cache-efficiency: *spatial locality* and *temporal locality*. Spatial locality means that data used together is stored close together in memory This type of locality benefits from caching partly because the data may share the same cache line, and partly because memory accesses become more predictable for the hardware-prefetching logic. Temporal locality means that recently used data is likely to be used soon again. A program exhibiting temporal locality benefits from the cache in that further accesses to the same memory location have greater probability of hitting the cache.

In a scene graph, nodes are sometimes larger than typical cache-line sizes (often 64 bytes). Hence, when laying out nodes in memory, there is limited benefit from considering spatial locality with the goal of fitting more nodes in the same cache line. However, the data layout can have a major impact on the effectiveness on the hardware prefetcher. If the nodes are stored in the same order that they are accessed during a traversal, the hardware-prefetching logic should trigger stream prefetching of the following nodes.

3.1.3 Cache models for algorithm analysis

The conventional big-O asymptotic time complexity analysis for algorithms relies on a *random-access* model of memory, wherein all memory accesses are assumed to be of equal performance cost. In a memory hierarchy, this is not the case: accessing memory locations corresponding to cache lines already in the cache is much faster than accessing other memory locations [4]. As such, a data structure or algorithm may have worse performance in practice than conventional time-complexity analysis suggests.

To resolve this, several models that do not assume random-access memory have been proposed. One such memory model is the two-level *I/O model* by Aggarwal and Vitter [1]. In this two-level model, there is a faster memory of size M and an external memory of infinite size. Data is transferred between these two levels in blocks of size B. The model is then used to analyse the number of such *memory transfers* of an algorithm, often asymptotically, yielding what we will refer to as *cache complexity*, analogously to conventional time complexity of algorithms.

In contrast to the typical case for cache memories, the I/O model assumes that memory transfers are completely controlled by the algorithm; as such, it is closer to a model of the relationship between secondary storage and internal memory, rather than between cache memory and internal memory.

Chatterjee et al. extended this model for further applicability to cache memories [29]. There, they add a parameter L for the relative cost of accessing the slow, large memory in relation to the cost of accessing the small, fast memory. A fixed mapping from memory address to cache block, as in a typical cache memory, is assumed. They provide a method for emulating the behaviour of algorithms in the I/O model using an extra buffer, allowing results for the I/O model to also apply to this cache model. Finally, they further extend the model to consider the effects of limited associativity and to include multiple levels of cache.

3.1.3.1 Cache-aware and cache-oblivious algorithms

There has been significant work on creating and tuning algorithms to reduce the cache complexity in various cache models. A *cache-aware* algorithm is defined as an algorithm with parameters that may be tuned to optimise the performance for some particular cache-line size (or other parameters of the cache) [27].

Cache-oblivious algorithms, introduced by Prokop in 1999, were proposed to avoid limitations with cache-awareness [27, 4]. In contrast to cache-aware algorithms, cache-oblivious ones make no use of parameters such as cache-line size. An algorithm that is optimal in terms of cache complexity in a cache-oblivious manner is optimal regardless of the actual cache parameters of the system on which it is run. This

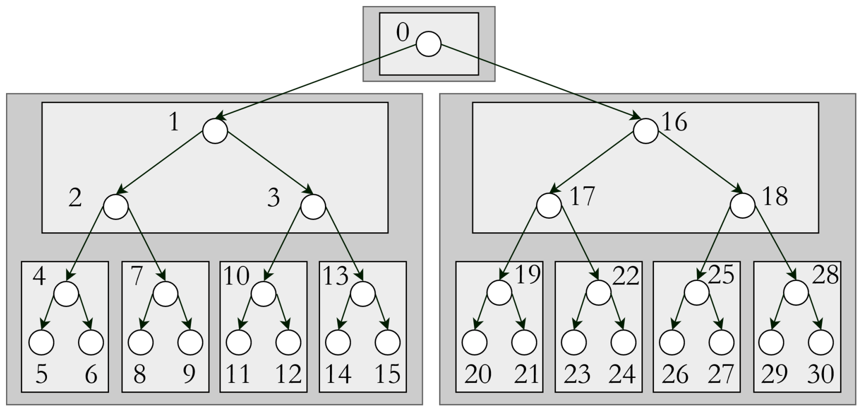


Figure 3.1: A binary tree in a *van Emde Boas* data layout.
The numbers indicate the nodes' location in the data layout.

makes algorithms more stable and portable to different systems and also allows them to be optimised for all levels of a memory hierarchy simultaneously.

Prokop presents a number of cache-optimal algorithms in the cache-oblivious model [27]. Since then, a large amount of further algorithms and data structures have been studied under this model. The cache-complexity analysis underlying these results builds on a two-level *ideal-cache model* wherein the first level — the cache — has size M and cache-line size B, and the second level (random-access memory) is arbitrarily large[3]. Cache lines can be placed at arbitrary locations in the cache, which is assumed to be fully associative and to be using an ideal replacement strategy: the cache line for which the next access is the furthest in the future is evicted first. Further, it is often assumed that the size of the cache is at least the square of the cache-line size: $M \geq B^2$. This is called the *tall-cache assumption*.

Some of the assumptions above may seem generous, but the number of memory transfers under this model have been shown to be within a constant factor from more realistic cache models (e.g. using least-recently-used cache-line eviction) [27]. Indeed, algorithms designed for the cache-oblivious model have been shown in empirical investigations to have comparable — albeit slightly worse — performance to cache-aware solutions [4].

A pertinent example of a cache-oblivious result, presented by Prokop, is the *van Emde Boas* data layout for binary search trees[4]. It is a recursive data layout that provides optimal cache-complexity for root-to-leaf traversals [27].

This data layout places the nodes in an array of size N (where N is the number of nodes) in the following manner. Divide the tree into a *top tree* and a set of *bottom trees* by splitting it in half by its height h[5]. The layout then consists of the van

[3]Prokop's thesis uses the notation Z for cache size and L for cache-line size.

[4]The van Emde Boas data layout is named for its parallels to a data structure presented by P. van Emde Boas [9].

[5]There are a few different ways to distribute the height between top and bottom trees if the height is not an even number. One such way gives the top tree the height $\lfloor h/2 \rfloor$ and the bottom trees height $\lceil h/2 \rceil$.

Emde Boas layout recursively applied to the top sub-tree followed by the van Emde Boas layouts of each of the bottom sub-trees (in left-to-right order). In the base case, the height $h = 1$, in which case the layout simply consists of that one node. An illustration of this layout can be seen in Figure 3.1.

The key to the cache-optimality of root-to-leaf searches in the van Emde Boas tree layout lies in its recursive nature. For any cache-line size B, pick the shallowest level of the recursion such that the size of the top and bottom sub-trees are less than or equal to B. The whole tree is divided into sub-trees of this size, and from the definition of the van Emde Boas layout, each such sub-tree is stored in a contiguous block of memory. Any path from root to leaf will traverse $\log N / \log B = \log_B N$ such sub-trees, each invoking $O(1)$ memory transfers (since the sub-trees are smaller than or equal to B in size). The resulting asymptotic cache-complexity is thus $\log_B N$, which is equivalent to that of a cache-aware B-tree [4].

3.2 Scene graphs

Scene graphs are hierarchical graphs (usually trees or directed acyclic graphs) representing a scene for 3D visualisation [2, Chapter 19]. Scene graphs are organised by spatial relations and semantic relations. For example, a node representing a car may have five child nodes: one for each wheel and one for the car's body. Node properties such as location in the scene are propagated to the node's children recursively down toward the leaf nodes. Scene graphs may also organise objects by other properties, such as surface material.

A typical flow for rendering a scene is that the scene graph is traversed from root toward the leaves, rendering the objects described by the nodes. Properties such as location are propagated from parent node to child node when the scene is rendered (or in a pass before). Hence, it suffices to change the location of the car node (from the previous example) to relocate both the car's body and its wheels.

Many results on cache-efficiency in tree data structures focus on various kinds of search trees, for example binary trees, B-trees, and so on. These differ from scene graphs in a crucial way: in search trees, the typical and the most performance-critical traversal is along a path from the root node to a leaf. This avoids (by design) visiting as many nodes as possible. In a scene graph, however, all nodes (or a large subset thereof) are visited each frame. The tree topology is not used to guide a traversal in order to find a node; instead, it represents the relation between nodes.

Due to the real-time nature of many scene graph use cases, it is vital that these data structures performs well; they must have an acceptable worst-case performance. In real-time visualisations, the scene needs to be rendered at a certain rate, for example 60 times per second [2, Chapter 1], which requires all per-frame computations to finish within about 16 ms. Average performance may also be important, but even occasional slow-down such that the system does not meet its timeliness requirements results in reduced utility (in a *soft real-time* requirement) or even outright failure of the system (in a *hard real-time* requirement) [30].

Scene graphs have a wide variety of use cases within applications, where some are more performance-critical than others. The order in which nodes are visited (which may vary depending on implementation) is of fundamental importance to the design of cache-efficient memory layouts. For this project, a particular usage pattern for scene graphs, intended to be representative of a single frame in a graphics application, is considered. The following sections describes each step of this usage pattern.

3.2.1 Updates

The application will often make changes to the scene graph during its running time, in both content and topology. This often consists of changing data within existing nodes (e.g. changing the translation value of a node's transform to move the corresponding object) but may also include destroying and creating nodes. The latter may occur when input parameters change, so that cached data must be re-computed; when scene elements are streamed in from secondary storage, as the viewer traverses the virtual world; when replacing a sub-tree with a simpler one, to represent a simplified version of an object when it is distant, or vice versa (level-of-detail rendering) [2, Chapter 19]; and simply when entities are created or disappear in the virtual world.

Updates are typically implemented by allocating and destroying node objects using regular dynamic memory allocators. In this thesis, a memory-management technique which can maintain the desired data layouts is required. Due to the apparent lack of published research on the topic, a novel technique is proposed in Section 4.4.

3.2.2 Transform propagation

One of the fundamental uses of a scene graph is to organise the spatial relations between objects in a scene in a hierarchical manner. Typically, this is achieved through the use of a *transform hierarchy*, wherein nodes contain a transform value. A transform is a collection of three attributes: translation, orientation (rotation), and scale. A translation is an offset in each of the two or three spatial dimensions of the scene. Transforms are commonly expressed as collections of vectors (for translation and scale) and unit quaternions (for orientation) or as a transform matrix [2, Chapter 4] (or both, the latter being calculated from the former).

The transform of a node is specified relative to the transform of its parent, and so on all the way up to the root. This way, changing the transform of a node that is root of a sub-tree (e.g. by adjusting the translation to move it up into the air) will also recursively affect all of its children.

To borrow the car-example from before (where a car is represented as a node with four child nodes for wheels and one child node for the body), moving the car's root node also moves all of the car's components. Each wheel node could specify a rotation around the wheel's axis as they roll; this rotation would then be applied after the rotation of the car's root node.

In some scene graphs, every node has a transform value (*Ogre* is an example of this [23]); in others, only special node types specify a transform, which applies recursively to that node's children (this is the case in *OpenSceneGraph* [26]).

This is may implemented by storing both *relative* (to parent node) and *absolute* (in world-space) transform values in each transform node [2, Chapter 19]. The absolute transform values are calculated by combining the relative-transform value of each node with the absolute-transform value of its parent, for example via matrix multiplication. There is an ordering dependency here, since parents' absolute-transform value must be calculated before that of any of its children. This is done by traversing the scene graph, propagating the effects of parent nodes down toward the leaf nodes, starting from the root.

Transform propagation is one example (perhaps the most important) of propagation of node properties in a scene graph. The approach can be generalised to the propagation of other properties; hence, we shall study how this type of propagation may be done in more detail. Two ways of performing propagation traversals are described below.

3.2.2.1 Breadth-first transform propagation

The breadth-first approach to transform propagation works as follows. For a tree-topology (as this project is restricted to) this type of traversal is equivalent to the following: divide the tree into a set of "depth levels" consisting of nodes at the same depth (i.e. distance from root); then traverse all the nodes in each depth level, starting with the root depth level and continuing downwards.

Breadth-first traversal can be implemented very cache-efficiently: traversing over the nodes in a depth level represented as an array results in $\Theta\left(m/B\right)$ memory transfers[6], where m is number of nodes in the depth level and B is the cache-line size[7].

For transform propagation, however, it is also required to read the transform values of each node's parent. This entails a look-up into the depth-level above the currently traversed one. If there is no relation between the ordering of nodes in the two depth levels, then each access of a parent node's transform is effectively random; this means that in the worst case, there may be a cache miss for each visited node: a cache-complexity of $O(m)$.

This changes if there is a distinct relation between the ordering of the nodes in the depth levels. If the nodes' data layout is a BFS layout, the cache-complexity has a

[6]For these cache-complexity analyses, we consider the maximum difference in size between node types to be a constant factor.

[7]This is simply the cache-complexity of scanning any array.

Algorithm 1 Depth-first transform propagation traversal

```
1: procedure TRANSFORM_DFS(parent_transform, cur_node)
2:     cur_node.absolute_transform ←
           COMBINE(parent_transform, cur_node.relative_transform)
3:     let cur_transform ← cur_node.absolute_transform
4:     for child in cur_node.children do
5:         TRANSFORM_DFS(cur_transform, child)
```

better upper bound:

Theorem 1. *The cache-complexity of sorted BFS-traversal for transform propagation is bounded by* $O\left(\frac{m_1+m_2}{B}\right)$*, where* m_1, m_2 *are the number of nodes in the depth-level currently traversed and the one above.*

Proof. When visiting a node, it either has the same parent as the preceding node, or a different parent; in that case the parent either immediately follows the preceding node's parent in the DFS-ordering, or there are one or more leaf nodes at the parents' depth-level in between. Hence, the index of the parent of each visited node must be non-decreasing as the index of the currently visited node grows.

As a result, the cache-complexity is bounded by the sum of the complexity of scanning the two depth-level arrays separately: Consider two caches of size $M/2$ and cache-line size B. Dedicate one of the caches to scanning the parent-depth array and the other to scanning the lower one. This results in a cache complexity of $O\left(m_1/B\right)$ and $O\left(m_2/B\right)$, respectively. By the ideal-cache assumption, the number of cache misses when using one cache to scan the two arrays (with any interleaving) can be no worse than this. Hence, the total cache-complexity is in $O\left(\frac{m_1+m_2}{B}\right)$. □

3.2.2.2 Depth-first transform propagation

This works very similarly to the aforementioned breadth-first approach. For this type, however, the traversal follows a depth-first ordering.

Keep the latest (parent node's) transform value on the function stack. Start with an identity transform, combine it with the root node's transform, and write the result to the root. Then, recursively call the function on each of the child nodes, and repeat the aforementioned procedure until reaching a node with no children (a leaf node). This is expressed in pseudo-code in Algorithm 1.

Theoretically, the ideal data layout for such a traversal is the DFS layout, which stores the nodes in the same order as they are visited. This effectively reduces the traversal to iterating over an array for the purposes of cache complexity, which for arrays of length n is $\Theta\left(n/B\right)$.

However, if the parent transform which is accessed in the procedure (*cur_transform* in the pseudo-code in Algorithm 1) is a *reference* to the transform value stored within the parent node object, the traversal is no longer equivalent to iterating over an array. To see why, consider a tree with a root node with two children, each the root of a sub-tree consisting of m nodes: When the traversal visits the first child of the root node, then *parent_transform* refers to location 0 in the tree's data layout and *cur_transform* refers to location 1. Then, the algorithm continues through the sub-tree rooted in the first child, visiting all m nodes within. Afterwards, it visits the second child of the root node, which would have location $m + 2$; *parent_transform*, however, is still stored at location 0, resulting in a memory access far backward, how far depending on the size of the sub-tree.

This can be avoided by keeping the whole parent-transform value on the function stack instead of as a reference to the value in its node. This will, however, result in a larger function stack, and an additional transform copy operation per recursive iteration.

3.2.3 Rendering traversal

The arguably most important operation done using a scene graph is rendering. This is typically done by traversing the scene graph from the root node and towards the leaf nodes; when reaching nodes representing an object to be rendered, a draw call is either dispatched directly or a draw command is written to some buffer to be handled by a separate rendering system[8]. Hierarchical properties in particular node types, such as surface material, are picked up during the traversal and are applied to the objects in the sub-tree below.

If some or all nodes in the scene graph contain bounding-volume information — information on the spatial extents of all objects within the sub-tree rooted in a node, such as a bounding sphere — this can be used in the rendering traversal to skip rendering objects outside of the current view, a technique called *frustum culling* [2, Chapter 19]. Using such techniques changes the rendering operation's traversal patterns from visiting all nodes to skipping whole sub-trees. Since this is effectively including a bounding-volume hierarchy into the scene graph, previous research on traversal patterns for bounding-volume hierarchies [34] may be applicable to such traversals.

[8]There are several reasons why the latter approach may be preferred. One such reason is that the render-command buffer can be sorted to minimise costly changes to graphics-API state (for more details on this, refer to the book "Real-time rendering" [2, Section 18.4]). Another is that scene graph traversal might be distributed over multiple threads; in some graphics APIs, draw calls can only be dispatched from one particular thread.

4

Implementation

This chapter describes the ways in which scene-graph data structures, their memory-management systems, and their associated traversals were implemented. To evaluate different data layouts and dynamic scene-graph memory-management techniques, the following components were implemented:

- A set of scene-graph nodes, representing a some commonly used node types in scene-graph applications.
- A set of traversals performing typical performance-sensitive operations on the scene graph.
- A data structure for maintaining a cache-efficient layout in a dynamic scene graph (where nodes are created and destroyed during an application's running time).

The implementation is done in standard C++17.

4.1 Scene-graph data layouts

This section presents the data layouts considered in this project. The set of child-pointers in a node is considered to be ordered and the data layouts (and traversals) follow this ordering. Four layouts were considered: depth-first layout, breadth-first layout, van Emde Boas layout, and a "naïve" layout. In these data layouts, with the exception of the naïve layout, nodes are tightly packed in a contiguous range of memory in the order described below for each.

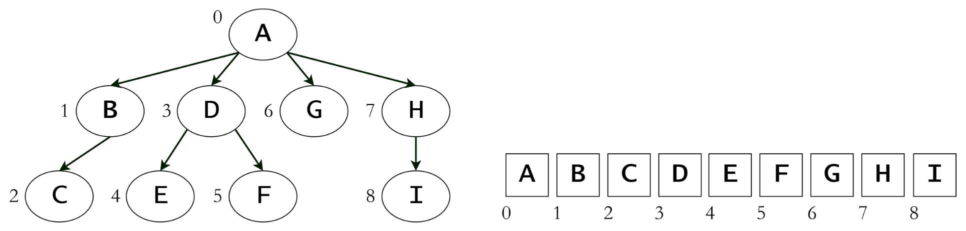


Figure 4.1: An example of a tree in a *depth-first order* layout (DFS layout). The tree topology is presented on the left, and its layout in memory on the right.

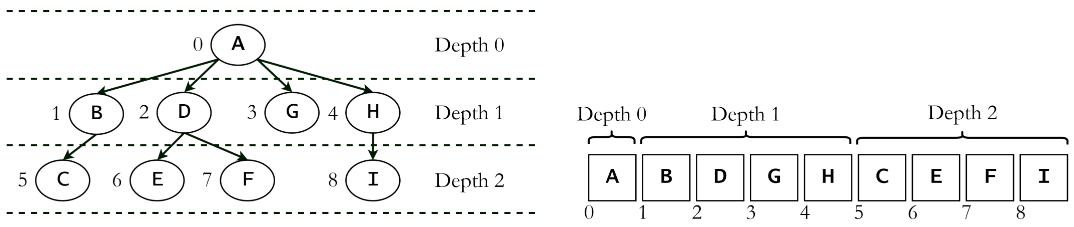


Figure 4.2: An example of a tree in a *breadth-first order* layout (BFS layout). The tree topology is presented on the left, and its layout in memory on the right.

The depth-first layout, or *DFS layout*, has the nodes laid out in the order that they would be visited by a depth-first search. An example can be seen in Figure 4.1.

The breadth-first layout, or *BFS layout*, is defined similarly, the nodes are laid out in the order that they would be visited by a breadth-first search. An example can be seen in Figure 4.2.

The van Emde Boas layout, or *vEB layout*, which is cache-optimal for search trees, is described above in Section 3.1.3.1. An example can be seen in Figure 3.1.

Finally, the "naïve" layout consists of allocating each node individually on the heap so that nodes are not stored contiguously and there is no pre-defined ordering between the nodes' addresses in memory. This applies to both the static scene graph evaluation and the dynamic data structure.

4.2 Node types

The scene-graph implementation uses a particular set of node types, representing a minimal subset of the node types which may be used in a scene-graph application. These nodes types are required to position, rotate, and scale elements in the scene; to describe the shapes to be rendered; and to describe the surface properties of those shapes. The library *OpenGL Mathematics* (GLM) [25] was used for vector, quaternion, and matrix types and for operations on those.

Three node types were implemented:

- *TransformNode*: a node containing two transform values: one transform relative to the parent and one absolute transform in world space[1]. Each of these consists of a translation, an orientation, and a per-axis scaling factor. The translation and per-axis scaling are represented by three-dimensional vectors of 32-bit floating-point values. The orientation is represented by a unit-quaternion value consisting four 32-bit floating-point values.
- *MaterialNode*: a node that contains a dummy pointer which in a real scene graph would point to the surface material that applies (recursively) to all child nodes.

[1]See Section 3.2.2 for an explanation of why both are needed.

```cpp
constexpr size_t local_storage_size = 8;
struct Mesh;     // Empty stand-in, never defined.
struct Material; // Empty stand-in, never defined.

// Used to refer to nodes in the dynamic data structure.
struct NodeHandle {
    uint32_t slice_index;
    uint32_t internal_index;
};
struct NodeBase {
    small_vector<NodeBase*, local_storage_size> children;
    NodeBase*  parent;
    NodeHandle self_handle;
    uint32_t   depth;
};
struct Transform {
    glm::quaternion orientation;
    glm::vec3       translation;
    glm::vec3       scale;
};

struct TransformNode : NodeBase {
    Transform transform;
    Transform absolute_transform;
};
struct MaterialNode : NodeBase {
    Material* material;
};
struct ShapeNode : NodeBase {
    Mesh*     mesh;
    glm::mat4 matrix;
};
```

Listing 1: Simplified definition of node types used in the experiments. *local_storage_size* is the number of elements for which storage is to be reserved inline in *children*.

- *ShapeNode*: a node containing a dummy pointer, which in a real scene graph would point to a geometrical shape (e.g. a 3D mesh) to render. It also contains a 4×4 transform matrix consisting of 32-bit floating-point values, representing the absolute transform for the node (renderers often require transform matrices of this form).

All of these inherit from a base class *NodeBase*, which contains common data like parent and child pointers. It also stores the node's depth (distance from root) and its self-handle (which is used to locate the node's storage in the dynamic data structure,

Node type	Size in bytes
TransformNode	180
MaterialNode	108
ShapeNode	172

Table 4.1: Size of node types in bytes on the evaluated platforms.
These values may depend on platform, compiler, and compiler configuration.

see Section 4.4.5).

The child-pointers are stored in a *small_vector* container, which is a dynamic array similar to the C++ standard library's *std::vector*, except that it reserves — inline within the *small_vector*-object — a fixed amount of storage; it falls back to dynamic memory allocation only when the number of elements grows past the fixed storage size — here eight elements. This is good for cache-efficiency, since accessing the child pointers of a node has high spatial locality in the common case of a node having eight or fewer children.

For exposition, simplified C++ definitions of the node types are presented in Listing 1. The sizes of each node type in bytes (on the tested platforms) can be seen in Table 4.1.

4.3 Traversals

This section presents the ways in which the scene-graph traversals described in Section 3.2 were implemented. All of the traversals were implemented using the *visitor pattern*, which allows different function overloads to be applied depending on an object's derived type without requiring operations to be implemented as virtual functions in the class hierarchy [12]. Each traversal is implemented as a visitor class which has a member function *visit*, overloaded for each node type. These traversals were used in benchmarks to evaluate scene-graph memory-management techniques, as is described in Chapter 5.

4.3.1 Transform propagation

Since the transform values in the scene graph are hierarchical such that each node's transform value is provided *relative* to that of its parent, they must at some point be propagated downwards, from the root towards the leaves, such that the *absolute* transform value for each node is produced. It may either be done in the same traversal as rendering or before it, in a separate traversal pass; the latter was chosen for this thesis.

Two ways to perform this type of traversal are described in detail in Section 3.2.2. The implementation follows the DFS-traversal order described in that section. The

```
void TransformVisitor::visit(TransformNode& node) {
    // Combine relative transform with parent's absolute transform
    // using overloaded operators from the glm library.
    node.absolute_transform.orientation = node.transform.orientation
                                        * parent_abs_transform->orientation;
    node.absolute_transform.translation = node.transform.translation
                                        + parent_abs_transform->translation;
    node.absolute_transform.scale       = node.transform.scale
                                        * parent_abs_transform->scale;
    parent_abs_transform = &node.absolute_transform;
    visit_children(node);
    parent_abs_transform = current_transform;
}
void TransformVisitor::visit(ShapeNode& node) {
    node.matrix = glm::mat4_cast(t.orientation) * glm::translate(t.translation)
                                                * glm::scale(t.scale);
    visit_children(node);
}
void TransformVisitor::visit(MaterialNode& node) {
    visit_children(node);
}
```

Listing 2: Simplified C++ implementation of *TransformVisitor*.

BFS-traversal order is difficult to apply efficiently to this scene graph, since any given *TransformNode* might not have another *TransformNode* as its parent; thus, finding the parent transform to combine with the transform of any visited node might entail traversing up all the way back to the root. The DFS-traversal is hence more general and works for more kinds of scene graphs.

Transform propagation is implemented as a visitor called *TransformVisitor*. It contains a pointer to *TransformNode* called *parent_abs_transform*. When the visitor is first constructed, *parent_abs_transform* points to an identity-transform value. The implementation is shown in Listing 2. The visitor traverses tree in DFS-order, with the different node types handled as follows.

When visiting a *TransformNode* a, the relative transform of a is combined with the transform pointed to by *parent_abs_transform* and the resulting absolute transform is stored in a's absolute-transform value. *parent_abs_transform* is then set to point to a's new absolute-transform value, and the visitor then visits the node's children. After the sub-tree has been traversed, *parent_abs_transform* is reset to the value it had before.

When visiting a *ShapeNode*, a transform matrix is calculated from the current value of *parent_abs_transform*, and is stored in the node.

TransformVisitor performs no operation when visiting a *MaterialNode*, and only continues to the child nodes.

```cpp
struct DrawCmd {
    Mesh const*     mesh;
    Material const* material;
};

std::vector<DrawCmd>   draw_command;
std::vector<glm::mat4> draw_matrices;

void visit(TransformNode& node) final { visit_children(node); }

void visit(ShapeNode& node) final
{
    draw_command.push_back({ node.mesh, current_material });
    draw_matrices.push_back(node.matrix);
    visit_children(node);
}

void visit(MaterialNode& node) final
{
    Material const* const prev_material = current_material;
    current_material                    = node.material;
    visit_children(node);
    current_material = prev_material;
}
```

Listing 3: Simplified C++ implementation of *RenderSimVisitor*.

4.3.2 Rendering traversal

Since this project is only concerned with the cache-efficiency aspect of scene graphs, no actual scene rendering was implemented. A traversal simulating how a scene graph might be accessed for rendering was, however, implemented to evaluate the effect of node data layout on rendering traversals.

The implemented traversal constructs two arrays holding data required for rendering, which in a real graphics application would be passed on to a rendering system. This type of separation between scene-graph traversal and rendering is one way in which a scene-graph renderer may be implemented; as was mentioned in Section 3.2.3, an alternative approach would be to dispatch draw calls to the GPU directly during the scene-graph traversal instead of creating intermediate arrays. For this project, though, what we aim to measure for is the performance cost of traversing the scene graph and reading the data. As such, it is of less importance which type of renderer is being simulated.

The rendering-simulation traversal works by traversing the scene graph in DFS-order in the following manner. When visiting a *ShapeNode*, the contained *Mesh* and *Ma-*

terial pointers are copied out into an array stored in the visitor[2]. The *ShapeNode*'s transform matrix is also copied out into a separate array within the visitor. These represent a minimalist data set required for rendering. The arrays are cleared between each iteration of the benchmark loop. A simplified C++ implementation is shown in Listing 3.

4.4 Dynamic scene graph data structure

The first research question of this project is to find which data layout is best (with regards to cache efficiency) for scene graphs. The second research question is how to maintain such layouts when the scene graph undergoes changes: how to maintain a *dynamic* cache-efficient scene graph. This section describes the implementation of a novel dynamic scene-graph data structure.

4.4.1 Goals and requirements

The main goal for the data structure is to minimise cache-miss performance costs during traversal by approximating the data layouts found most efficient for static scene graphs. Secondly, the space and time overhead for maintaining the data structure itself should be minimised.

Ideally, the dynamic scene graph data structure should be configurable to maintain any of the data layouts considered for this project. In practice, however, restricting the design to a particular data layout allows more flexibility in the design of the data structure. The present implementation supports only the DFS layout, but could be extended to allow other data layouts. This restriction allows a simple method of deciding in which contiguous block of memory to store any given node, and allows an efficient algorithm for determining desired node order. Further research would be required to find efficient algorithms for maintaining other data layouts in dynamic scene graphs.

4.4.2 Data structure design

An overview of the design is as follows. The nodes are allocated in memory-managing units which we will refer to as *slices*. Each slice consists of a contiguous region in memory, which is divided into a node-storage region and a meta-data region. All nodes stored in a slice belong to the same sub-tree and are stored as close to DFS-layout order as possible.

A slice may hold up to a certain amount of nodes, limited by its storage space and meta-data capacity. The meta-data in a slice consists of an array of structures of

[2]The *std::vector* dynamic arrays have memory reserved before the benchmark loop, to avoid the re-allocation and copying operations that would otherwise be required to grow the arrays.

```
struct NodeMeta {
    uint32_t offset;
    uint32_t size;
    uint32_t next_free;
};

class Slice {
    std::unique_ptr<uint8_t[]> data;
    span<uint8_t>              storage;       // view into data
    span<NodeMeta>             node_metadata; // view into data
    uint32_t                   num_nodes;
    uint32_t                   next_storage_index;
    uint32_t                   next_insert_offset;
    uint32_t                   occupied_space;
    uint32_t                   first_free;

public:
    Slice(size_t node_storage_size, size_t max_num_nodes);

    template<typename NodeT, typename... Args>
    uint32_t insert(Args&&... args); // Returns node's internal_index

    void erase(uint32_t node_internal_index);

    void reconstitute(float expansion_factor);
};
```

Listing 4: Simplified definition of the *Slice* data management type. The *span* values are views into the byte-array owned by *data*, and consist of a pointer and a size.

the form {*node_offset*, *node_size*, *next_free*}. These values represent, respectively, at which offset in the node-storage region a node is located and what the size of the storage allocated to the node is (the purpose of *next_free* is discussed in Section 4.4.4). The index of the meta-data-array element that corresponds to a certain node is that node's *internal index* in the slice. The internal index remains valid for the lifetime of the referred-to node, even when the slice is re-built in the reconstitution operation (which is described below).

A simplified definition of the *Slice* implementation is shown in Listing 4. The following operations are provided:

- Constructor: create a new slice. Allocates, in a single allocation, space for both node storage and node meta-data. Parameters: *node_storage_size*, the size (in bytes) to allocate for storing nodes; *max_num_nodes*, the size of the array for storing node meta-data.

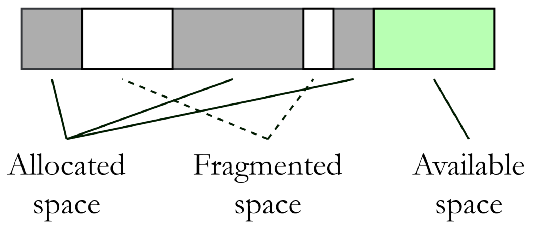


Figure 4.3: An example of a node-storage data region with space lost to fragmentation.

- *insert*: create a new node of a requested type, forwarding the provided arguments to the node's constructor. Returns the new node's internal-index value (or -1, if the node cannot fit).
- *erase*: destroy the node with the given internal-index.
- *reconstitute*: allocate new, larger storage, and move nodes and associated meta-data to the new storage. The nodes are tightly packed in the new storage in the DFS-layout order.

The whole data structure consists of a set of such slices. Exactly how the scene graph should be divided into slices can be adjusted using application-specific knowledge, but for this thesis, a particular system was chosen, as follows. The root node is given its own slice, which is just large enough to store it. When a node created with the root as its direct parent, a new slice is created for it. In all other cases, the node is created in the same slice as its parent.

The rationale for this approach is based on the following observation. In a scene graph, the only relation between two nodes a, b which are immediate children of the root node (and the sub-trees rooted in a and b) is that they are present in the same scene. We shall use the term *independent sub-tree* to refer to sub-trees rooted in such nodes. Separating the storage management for each independent sub-tree is a natural and practical choice: it is likely that whole independent sub-trees are created in a single frame, and similarly for destruction. If, in contrast, all nodes were stored in the same memory region, it may be necessary to move an unbounded number of nodes to fit new nodes. When the storage is separated per independent sub-tree, the number of nodes which would have to be moved in such a case is limited by the number of nodes within that sub-tree.

4.4.3 Node-storage memory management

The node-storage region in a slice is managed by a memory-allocation policy. One such policy is implemented: a *linear allocation* policy. The linear allocation policy allows nodes of varying size to be tightly packed in memory with very quick (constant-time) allocation performance, at the cost of high fragmentation. The fragmentation and data-layout-ordering problems are both handled by the reconstitution operation as described in Section 4.4.6.

A linear allocator simply tracks where the end of the previous allocation is and allocates further nodes at the immediately-following address. The benefit of this is that nodes will be tightly packed in memory irrespective of their size. However, the only condition by which it is possible to re-use memory after de-allocating a node is when the most recently allocated node is de-allocated (a first-in, last-out order). Other de-allocations will leave unusable gaps in the node-storage memory; the space is lost to *fragmentation*. Such losses will, however, be reclaimed during the next reconstitution operation. An example of a storage region, managed by a linear allocator, in fragmented state can be seen in Figure 4.3.

4.4.4 Management of meta-data array

Whenever a new node is inserted into the slice, an element of the meta-data array must be allocated for it. To track which elements in the meta-data array are unoccupied, a free-list of unoccupied elements is embedded in the array.

The slice contains a variable *first_free*, which holds the index of the first unoccupied meta-data array element (or -1, in case all elements in the array are occupied). Each unoccupied meta-data element contains a variable *next_free*, the index of the next unoccupied element. When the slice is first initialised, the *first_free* variable is assigned 0; then, the meta-data array is iterated and the *next_free* in each element is set to refer to the next element. The *next_free* in the very last element of the array instead gets -1.

Whenever a new node is inserted, the first free element (i.e. the one at index *first_free*) is allocated for the new node's meta-data. The element is removed from the free-list by overwriting the slice's *first_free* value with the element's *next_free*.

Similarly, when a node is removed, its meta-data element is appended to the free-list by assigning its *next_free* the value currently in *first_free*, and then assigning *first_free* the index of the newly-unoccupied meta-data element.

4.4.5 Maintaining references to relocated nodes

As the scene graph changes with nodes being added and removed, maintaining a cache-efficient layout will require moving existing nodes to different locations in memory. One of the challenges with this is to ensure that external references to individual nodes are not invalidated. Regular C++ pointers and references would be invalidated, as the memory address to which they refer no longer holds the target node. This requires that the implementation either provides a handle-to-node data type which remains valid even as the nodes are relocated or that all references to a node are updated when the node changes.

The implementation here does both: *NodeHandle* objects, consisting of a *slice-index* and a *internal-index* (each an unsigned 32-bit integer) may be used to refer to nodes in a manner robust to relocations. Further, the nodes' parent and child pointers —

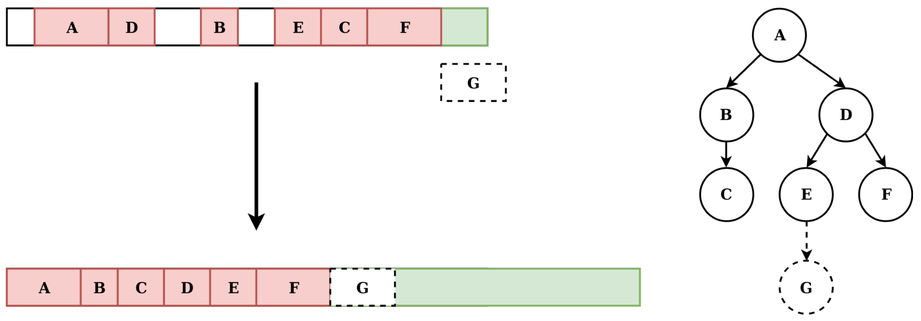


Figure 4.4: An illustration of the reconstitution operation.
The slice's original node-data storage is shown on top and the corresponding (sub-) tree is shown on the right. The application attempts to insert node G, but there is not enough space at the end of the node-storage area. Hence, reconstitution is invoked. The result is the larger node-storage area on the bottom, where the nodes are now stored in DFS-layout order and are tightly packed in memory.

which are still regular pointers — are updated to refer to the new location when a node is relocated.

4.4.6 The reconstitution operation

When a slice cannot fit any more nodes — either because the node cannot fit into the remaining node-data storage, or because the meta-data array is fully occupied — it may be *reconstituted*. This means that a new, larger region of memory is allocated, and all nodes are moved over into the new region. In addition, the nodes are rearranged in memory as they are copied, such that they will be in the desired DFS data layout order in the new storage, irrespective of whether they were before.

Reconstitution is implemented as follows. First, the new storage size is calculated. This is done by multiplying the storage size by some *expansion factor*. Exactly what this factor is can be exposed as a configurable parameter; for this project, an expansion factor of 1.5 was experimentally chosen. The new storage is allocated by creating a new slice with the requested size, and all node meta-data is copied over into the new slice.

An array of all nodes' internal-index values, stored in the desired DFS-layout order, is created as follows. The node which is root of the sub-tree within the slice is visited by a recursive function *make_sorted_node_indices*. This function first checks if the node belongs to the same slice (by extracting the slice index from the node's self-referencing *NodeHandle*, see Section 4.2). Otherwise, it writes the node's internal-index (also from the self-referencing *NodeHandle*) to the internal-index array. It then recurses over all the node's children. Hence, the result is an array of all nodes within the slice in DFS-layout order.

The array is then traversed and the contained node-indices are used to find each node in the now-sorted order; these nodes are moved over into the new slice by copying their data (using the C++ “move-construction” mechanism[3]). The child-pointer in the node’s parent and the parent-pointers of all its children are updated to refer to the new address. Updating the child-pointer in the node’s parent requires scanning through all the parent’s child pointers to find the right one to update. Finally, the storage buffer for the original slice is released and is replaced with the new slice data.

The time complexity of the reconstitution operation is bounded by $O(nm)$, where n is the number of nodes in the slice and m is the largest number of children of any node. The operation (as described above) visits each node in the slice only once; it constructs an array of internal-indices of size n and iterates over this list to move each node. Moving a node has time complexity m (the largest number of children of a node). This is due to the updating of the node’s parent’s child-pointer. If m is bounded by a constant, the reconstitution operation can thus be said to run in linear time[4].

4.4.7 Configurable parameters

The data structure has some (compile-time) configurable parameters. These are listed below.

- *expansion_factor*: the factor by which a slice’s node-storage space and meta-data array should grow during reconstitution.
- *initial_slice_storage_size*: the initial node-storage size (in bytes) for new slices.
- *initial_slice_max_num_nodes*: the initial meta-data array size (in number of elements) for new slices.

[3]An implementation problem manifests here. The slice operations work with pointers to *NodeBase* objects, but the actual nodes are of some derived type. Copying or moving an object in C++ requires that the actual type is known. To resolve this, *NodeBase* was given a virtual member function *NodeBase::move_to(void* target_address)*, which is implemented in the derived types to perform the actual move-operation.

[4]Another approach for linear-time reconstitution exists: within each node, store the index (in the parent’s child-pointer array) of the child-pointer corresponding to itself (aka. the node’s *sibling index*). This allows direct access to the parent’s corresponding child-pointer without scanning the array. However, that approach had worse performance in practice: when a node is deleted, the sibling index of the children following the deleted one must be updated, as they shift to cover the gap. This updating increased the number of cache misses, decreasing overall performance.

5

Evaluation methodology

This chapter describes the methods by which different data layouts were evaluated for static scene graphs and how the dynamic scene-graph data structure was evaluated.

A set of benchmarks were created, measuring the time taken to perform the traversals described in the previous chapter. The benchmarks make use of pseudo-random number generation. This was done using deterministic number generators and fixed seed-values, such that each invocation of a benchmark performs the exact same operations (regardless of e.g. which data layout is studied).

The benchmarks were run on the hardware and software platforms described in Table 5.1. For benchmarks on *Windows*, the code was compiled with *Visual Studio 2019*. For benchmarks on *Linux*, the code was compiled with *GCC 9.0*. The maximum optimisation level and latest available instruction set were used (`-O3 -march=native` for *GCC* and `/O2 /arch:AVX2` for *Visual Studio*).

Each benchmark was made in two varieties: one with and one without *flush*. In the flush benchmarks, the x86-assembly instruction *clflush* is used to evict the entire scene graph from all levels of cache between each iteration. Flushing the scene graph from cache in-between each iteration is intended to simulate an application with a large working set outside of the scene graph. Results in a real application would likely fall somewhere in between the flush and no-flush benchmarks. Flushing also highlights the effect of hardware prefetching.

CPU	L1†	L2†	L3	RAM	OS
Intel i7-8700	64KiB	256KiB	12MiB	32GiB DDR4-2666	Windows 10
Intel i5-6300HQ	64KiB	256KiB	6MiB	8GiB DDR4-2133	Linux Mint 18

Table 5.1: The hardware and software platforms upon which evaluation was performed.
†Each CPU core has its own L1 and L2 caches.

Type	Probability
TransformNode	0.4
ShapeNode	0.4
MaterialNode	0.2

Table 5.2: Distribution of node types in the benchmark scene graphs.

5.1 Scene graph data layout evaluation

To answer the first research question — how to layout node data in memory to minimise memory-transfer costs for typical scene graph use cases — a set of benchmarks implementing the traversals described in the theory section (see Section 3.2) were created for various data layouts and scene graph sizes. The benchmarks measure the wall-clock time taken to perform the selected operations on the scene graphs.

The following data layouts, as described in Section 4.1, were considered:

- Depth-first (DFS layout)
- Breadth-first (BFS layout)
- van Emde Boas (vEB layout)
- Each node allocated individually on the heap (naïve layout)

Algorithm 2 Pseudo-code outline of all static scene graph benchmarks.

```
1: procedure BENCHMARK(num_nodes, benchmark_operation, iterations)
2:     let scene_graph ← MAKE_SCENE_GRAPH(num_nodes)
3:     let timings ← []
4:     for i from 0 to iterations do
5:         START_TIME_COUNT()
6:         BENCHMARK_OPERATION(scene_graph)
7:         APPEND(timings, FINISH_TIME_COUNT())
```

All benchmarks follow the same general outline. The outline is described in pseudocode in Algorithm 2. A scene graph is constructed by adding nodes one-by-one as a child to a pseudo-randomly selected existing node. For each node to be created, the type is chosen pseudo-randomly with the following probability distribution: 40% probability of *TransformNode*, 40% probability of *ShapeNode*, 20% probability of *MaterialNode*.

After the scene graph is constructed, the main benchmark code is run and measured over 30 iterations. The whole process is repeated enough times to traverse $2^{22} = 4194304$ nodes per benchmark (irrespective of scene-graph size, i.e. there are more repetitions for the benchmarks with small scene graphs). Results are collected for each repetition, for each benchmark and scene-graph size.

Num. nodes	512	4096	32768	262144	1048576
Iterations	8192	1024	128	16	4
Data set size	81.2KiB	649KiB	5.07MiB	40.6MiB	162MiB

Table 5.3: Scene graph sizes, number of benchmark iterations, and data set size for all benchmarks.
The data set size is calculated as the sum of *node-type-size* × *node-type-proportion* × *num-nodes* for all node types.

Each benchmark is run for a variety of graph sizes (in terms of number of nodes). The chosen graph sizes are described in Table 5.3. While the larger of these sizes may seem quite large for many scene graph applications, they are of interest for the following reasons:

- Measuring up to very large graph sizes indicates the scalability of the solution.
- A real application will likely have a significant working set aside from the scene graph. Hence, the cache memory (and memory throughput) would not be entirely dedicated to the scene graph. The performance of benchmarking a large scene graph might hence be more representative for real applications.
- Results for large graphs are an indication of the performance on more constrained hardware environments (with more limited memory throughput or smaller cache memory, for example).
- The larger graph sizes are not inconceivable for some types of applications.

Benchmark applications were constructed for both of the traversal types described in Section 4.3: transform propagation and rendering simulation, along with a "frame simulation" benchmark that performs both traversals to simulate the behaviour of a graphics application during a single frame. For each iteration of the benchmark, the latter first performs a transform-propagation traversal and then a rendering simulation traversal.

5.1.1 Dynamic scene graph data structure evaluation

To evaluate the dynamic scene graph data structure, the same set of benchmarks as described above were run: transform propagation, rendering simulation, and frame simulation. However, the benchmarks were extended with updates: the scene graph is modified between each iteration, simulating an application updating the scene graph between frames. The new outline can be seen in pseudo-code in Algorithm 3.

5.1.1.1 Scene graph updates

To simulate updates to the scene graph in a graphics application, the scene graph is modified in the following manner. 20% of the nodes are marked for modification:

Algorithm 3 Pseudo-code outline of dynamic scene graph benchmarks; similar to that for static scene graph benchmarks but with updates to the scene graph in each iteration. *update_proportion* was set to 0.1 (10% of nodes deleted per iteration and equally many added).

```
1:  procedure BENCHMARK(num_nodes, benchmark_operation, iterations)
2:      let scene_graph ← MAKE_SCENE_GRAPH(num_nodes)
3:      let num_to_update ← num_nodes × update_proportion
4:      let update_timings ← []
5:      let traversal_timings ← []

6:      for i from 0 to iterations do
                                        ▷ Select nodes for updating (delete/extend).
7:          let leaves ← GET_ALL_LEAF_NODES(scene_graph)
8:          RANDOM_SHUFFLE(leaves)
9:          let to_delete ← SUBARRAY(leaves, begin: 0, num: num_to_update)
10:         let to_extend ← SUBARRAY(leaves, begin: num_to_update,
                                              num: num_to_update)

                                        ▷ Perform scene-graph updates.
11:         START_TIME_COUNT()
12:         for node in to_delete do
13:             DELETE_NODE(scene_graph, node)
14:         for node in to_extend do
15:             CREATE_NODE_WITH_PARENT(scene_graph, node)
16:         APPEND(update_timings, FINISH_TIME_COUNT())

                                        ▷ Perform scene-graph traversal.
17:         START_TIME_COUNT()
18:         BENCHMARK_OPERATION(scene_graph)
19:         APPEND(traversal_timings, FINISH_TIME_COUNT())
```

10% to be deleted, 10% to be extended with a new child node. This is done in two phases.

First, handles to all leaf nodes in the scene graph are created and a number of them, corresponding to 10% of the number of nodes in the scene graph, are pseudo-randomly selected for deletion. The selected nodes are then deleted. Restricting deletion to leaf nodes ensures that only one node is deleted. As such, the size of the scene graph remains the same before and after the update. If other nodes were selected, then the whole sub-tree would have to be deleted, changing the number of nodes in the graph between iterations.

After the deletion phase, equally many handles to remaining leaf nodes are selected to be extended: to receive a new child node. The type of node to insert is pseudo-randomly selected following the same probability distribution as the distribution of node types in the scene graph as a whole (40% probability of *TransformNode*, 40% of *ShapeNode*, and 20% of *MaterialNode*). The new nodes are then inserted into the scene graph as child nodes to the nodes selected for extension.

The benchmark measurements do not include the time taken to create the lists containing the nodes to delete and extend.

Parameter	Value
expansion_factor	1.5
initial_node_storage_size	1024
initial_max_num_nodes	12

Table 5.4: Parameters for the dynamic scene graph data structure and their chosen values.

5.1.2 Parameters

A few configurable parameters for the dynamic data structure were mentioned in Section 4.4.7. The values chosen for them in the evaluation are shown in Table 5.4. All of the values were chosen experimentally.

For convenience, we summarise the function of each parameter here:

- *expansion_factor* is the factor by which a slice's node-storage space and meta-data array should grow during reconstitution.
- *initial_slice_storage_size* is the initial node-storage size (in bytes) for new slices.
- *initial_slice_max_num_nodes* is the initial meta-data array size (in number of elements) for new slices.

6

Results

This chapter presents the results of the benchmarks described in Chapter 5 along with analysis of causes and implications of the results.

An interesting, overarching result is that performance is very similar across the two test platforms, despite one having significantly greater processing capacity than the other. This illustrates how memory may bottleneck applications, preventing computation capacity from reaching maximal utilisation. Some of the effect may also be caused by differences in code generation between the compilers.

The results in the benchmarks with cache flush may be compared with the upper limit on performance when considering only memory bandwidth. Assuming memory speed is the only limiting factor, that performance would be achieved with perfect prefetching: then, bandwidth would be the only limiting factor, not memory latency. The *i7-8700* and *i5-6300HQ* test platforms had DDR4-2666 and DDR4-2133 memory types, respectively, with bandwidths of 21333 MiB/s and 17066 MiB/s. Comparing with the average node size — weighted by node-type distribution — of 162 bytes[1], this results in

$$\frac{162\,\mathrm{B}}{21333\,\mathrm{MiB/s}} = 7.24\,\mathrm{ns}$$

per node for the *i7-8700* platform, and

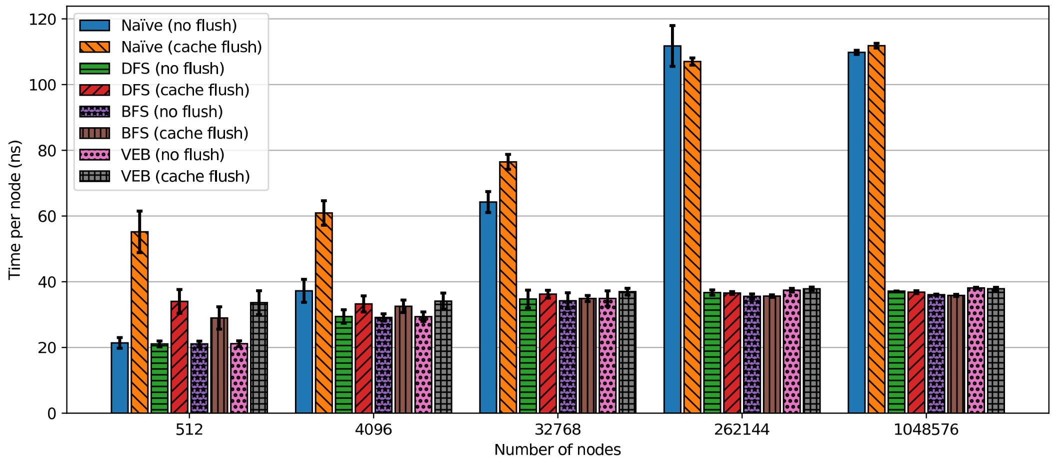
$$\frac{162\,\mathrm{B}}{17066\,\mathrm{MiB/s}} = 9.05\,\mathrm{ns}$$

per node for the *i5-6300HQ* platform.

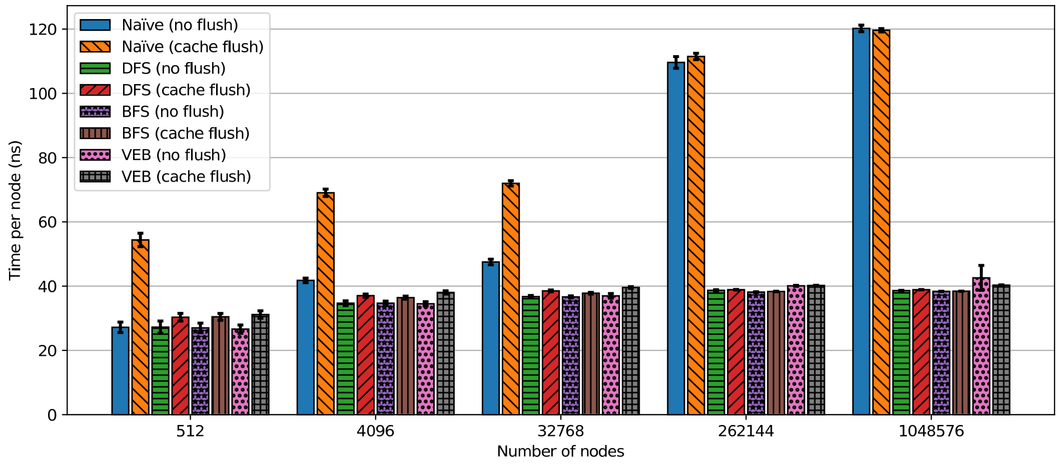
6.1 Data-layout evaluation for static scene graphs

The results of the transform propagation and rendering-simulation benchmarks are shown in Figures 6.1 and 6.2, respectively. The error bars show the standard deviation. These results suggest that depth-first order, breadth-first order, and van Emde Boas data layouts are similarly ideal for the investigated scene-graph usage patterns. For both benchmarks, breadth-first, depth-first, and van Emde boas memory layout

[1]See Tables 4.1 and 5.2.

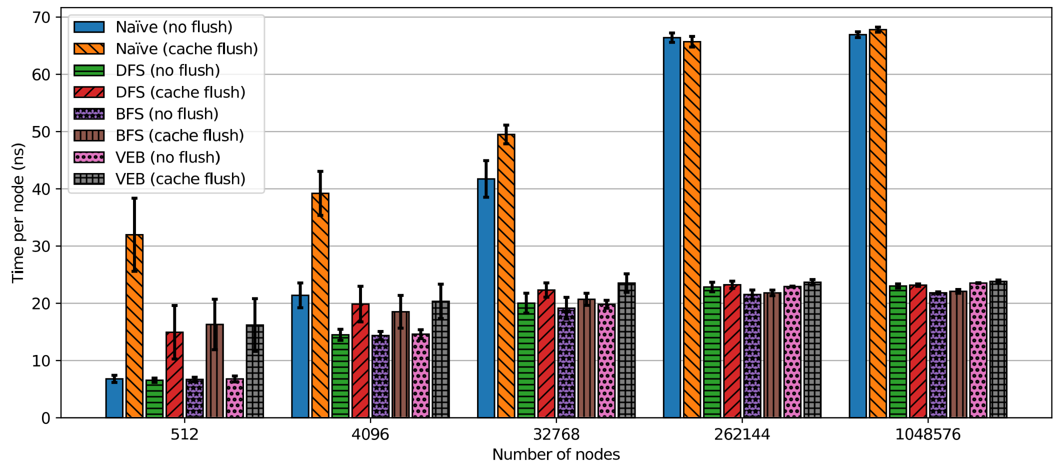


(a) *Intel i5-6300HQ, Linux, GCC*

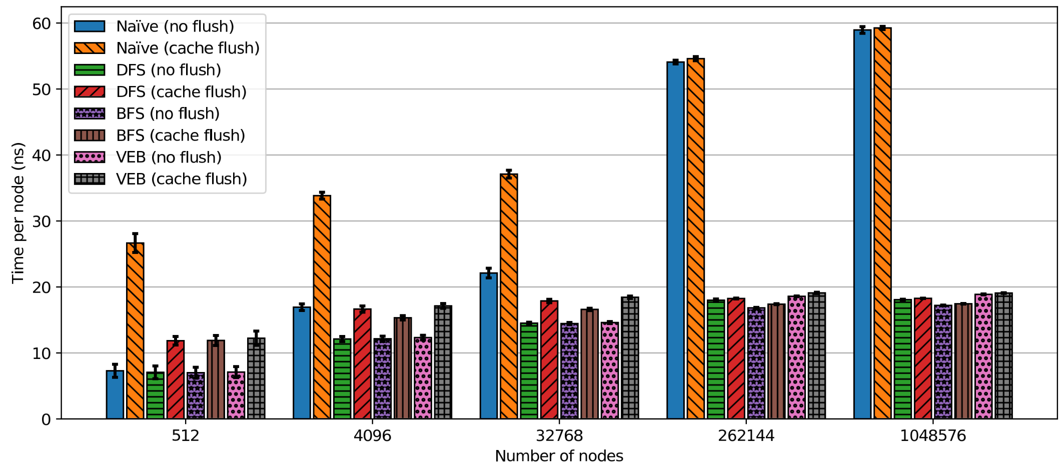


(b) *Intel i7-8700, Windows, Visual Studio*

Figure 6.1: Results of the transform-propagation benchmark.

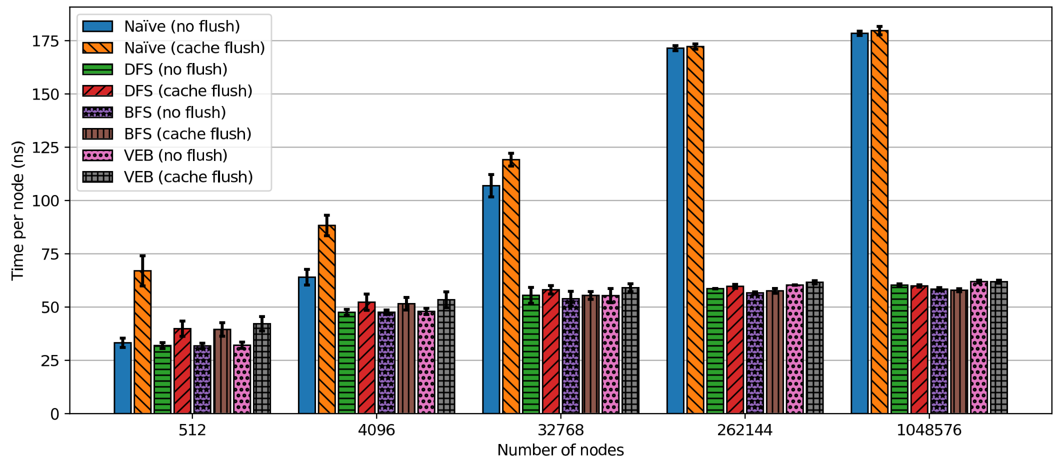


(a) *Intel i5-6300HQ, Linux, GCC*

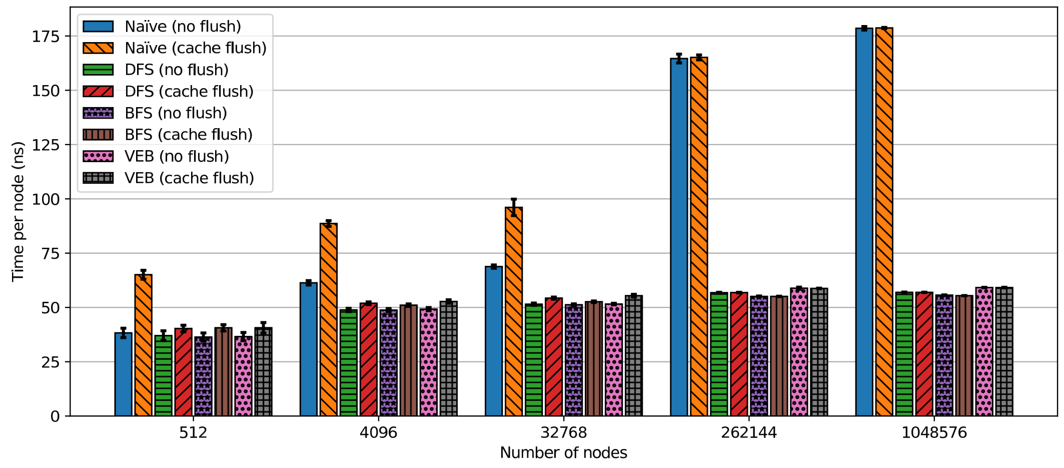


(b) *Intel i7-8700, Windows, Visual Studio*

Figure 6.2: Results of the rendering-simulation benchmark.



(a) *Intel i5-6300HQ, Linux, GCC*



(b) *Intel i7-8700, Windows, Visual Studio*

Figure 6.3: Results of the combined-traversals benchmark (both transform propagation and rendering simulation).

provided close-to-linear scaling with graph size, whereas the naïve layout resulted in super-linear scaling (as the probability of cache miss per node access increased).

The performance impact was greatly dependent on problem size; for small scene graphs, memory layout had little effect on results. This may be explained by the fact that the entire graph fits in cache memory, and will hence invoke no cache misses after the first benchmark iteration. This is confirmed by the values obtained for the benchmarks with “flush” — ones in which all scene-graph cache lines are flushed between each iteration of the benchmark, since the results for the naïve layout is significantly slower than the others even at small scene-graph sizes.

The flush benchmarks highlight the effect of hardware prefetching. Performance is almost unaffected by problem size in the transform-propagation benchmarks with flush. At the start of each iteration, none of the scene graph data are in cache, and yet the performance is significantly better for the non-naïve data layouts, which can only be explained by prefetching.

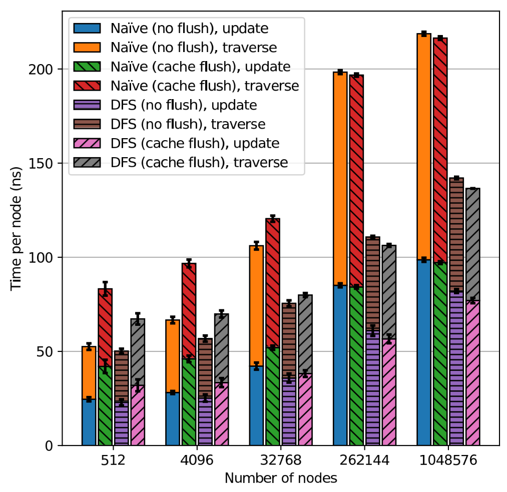
Interestingly, there is not much difference between the results of the various non-naïve layouts, only between those and the naïve layout. One might have expected DFS layout to provide superior performance, since it stores the nodes in the same order as they are accessed, which should be ideal for hardware prefetching. The fact that the others fair equally well demonstrates the capability of the hardware prefetching logic to detect multiple memory-access streams.

In the results for the *Intel i5-6300HQ*, for the no-flush benchmarks, the performance difference between scene-graph sizes of 4096 nodes and 32768 nodes are much more pronounced than those in the results of the *Intel i7-8700K*. This is because the data size of a scene graph of 32768 nodes fits well inside the larger L3 cache of the latter CPU.

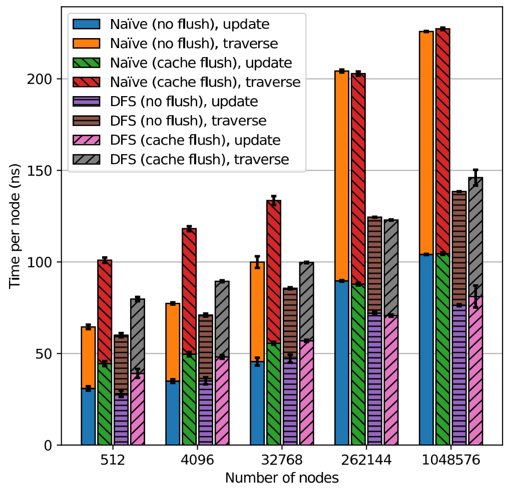
Execution times in the rendering-simulation benchmarks appear to be slightly more sensitive to the scene graph’s size than in the transform-propagation benchmark. This may be explained by the different types of work performed. The rendering-simulation benchmark performs no calculations, only traversing the graph and copying out data, whereas the transform-propagation benchmark is more computationally expensive due to the transform-combination calculations. The latter may “hide” some of the latency of memory accesses.

6.2 Dynamic scene graphs

Evaluating the dynamic scene-graph data structure benchmarks displayed significant performance improvements compared with the naïve implementation — wherein each node is allocated individually using regular dynamic memory allocation — for large benchmarks. The execution time does scale super-linearly as a function of scene-graph size, but the effect of the data layout is large enough to more than compensate for the overhead of maintaining the data layout. The general approach of the dynamic data structure is thus promising. Note also that updates themselves

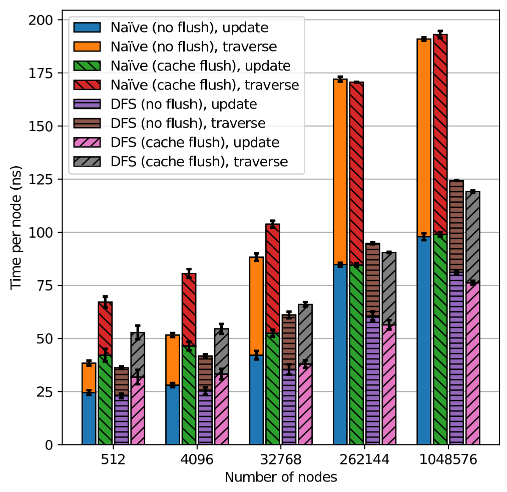


(a) *Intel i5-6300HQ, Linux, GCC*

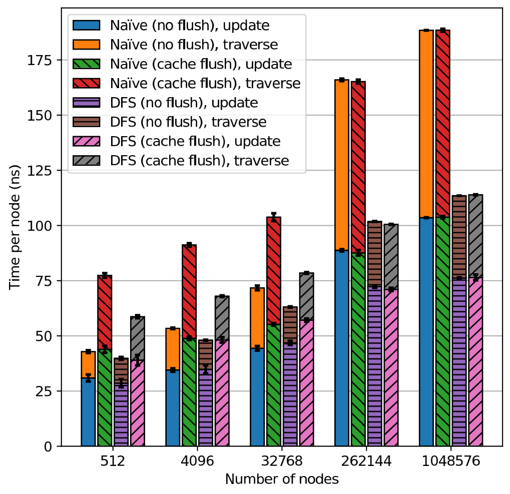


(b) *Intel i7-8700, Windows, Visual Studio*

Figure 6.4: Results of the dynamic transform-propagation benchmark.

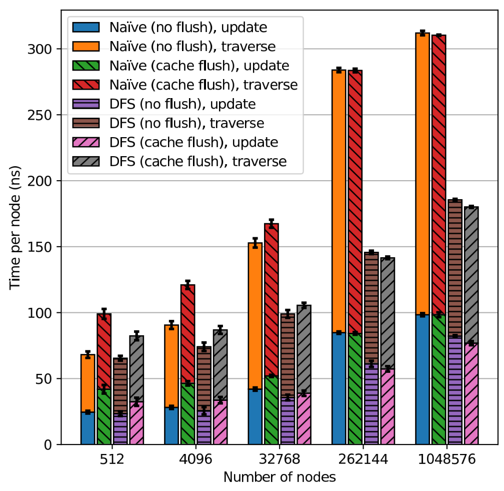


(a) *Intel i5-6300HQ, Linux, GCC*

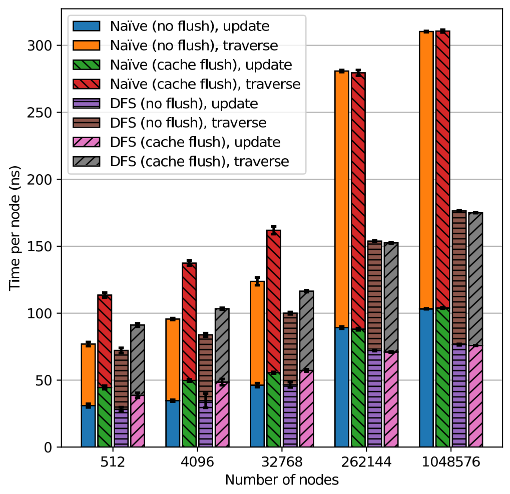


(b) *Intel i7-8700, Windows, Visual Studio*

Figure 6.5: Results of the dynamic rendering-simulation benchmark.



(a) *Intel i5-6300HQ, Linux, GCC*



(b) *Intel i7-8700, Windows, Visual Studio*

Figure 6.6: Results of the dynamic combined (frame-simulation) benchmark.

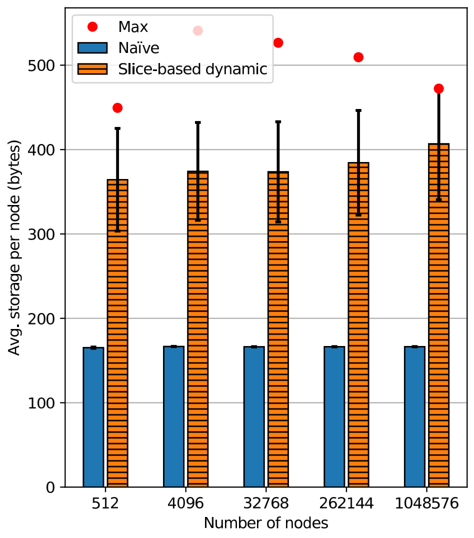


Figure 6.7: Storage overhead of dynamic-DFS (slice-based) data structure compared with naïve solution.

are generally faster in the dynamic data structure than in the naïve implementation. Despite the overhead of reconstution, avoiding dynamic memory allocations for each node is thus still a performance gain.

Memory usage, however, rose significantly. The increased memory usage is caused by the unused memory allocated to the data structure's slices. On average, as can be seen in Figure 6.7, the storage overhead increased by a factor of circa 2.2, with a maximum of about 3.3.

The difference between flush and no-flush benchmarks is much smaller here than in the static scene-graph benchmarks. This is likely due to the increased data throughput of the benchmark code itself. To perform the updates in each iteration, a list of all node-handles has to be created and shuffled. This increases contention for the cache and will cause many scene-graph cache lines to be evicted between traversals even in the no-flush case.

7 Discussion

Data layout had — on its own — a major impact on performance, by about a factor of three in the rendering simulation benchmark for large, static scene graphs. This is a clear illustration of the importance of data layout and memory access patterns.

Maintaining a good data layout when the scene graph undergoes unpredictable changes is challenging. Better results could likely be gained by using application-specific knowledge; using knowledge about how large a sub-tree is, whether a given sub-tree will change, and what sub-trees represent logically distinct objects can be immensely useful when maintaining the data layout.

7.1 Limitations of the dynamic data structure

The dynamic data structure proved useful in the benchmarks but has some limitations. A pathological case for its design is a scene graph where the root node has only one child node which in turn is parent to a large sub-tree. This would cause all nodes but the root to be stored in the same slice. A different slice-distribution mechanism would be needed for such cases.

7.2 Other approaches

This project was only concerned with node data layout. This allows existing scene graphs to be adapted, since the only thing that needs to be modified is the data placement. However, it may be possible to achieve better results by studying the possibilities of more fundamental changes to the scene graph representation. For example, one might want to separate the storage of node's data from the topology information; that is, do not store the parent and child-pointers together with the rest of the data. This could allow more radically different scene graph representations such as structure-of-arrays (SoA) approaches.

7.3 Ethical considerations

The project consists entirely of the development and analysis of a low-level software component. As such, there are no particular ethical issues involved. A more efficient scene-graph data structure can not, in itself, cause any harm. A higher-level software system built using the results could cause harm, but that applies no more to this project than to any other low-level software component or optimisation technique. The project made no use of human or animal subjects and no sensitive data was handled.

7.4 Future work

The results were promising but had some limitations, as discussed above. Some suggested directions for future work to resolve these limitations and to explore different approaches are described here.

7.4.1 Traverse in breadth-first order

In this report, all traversals over the scene graph move across it in depth-first order. An alternative traversal order was described for transform propagation in Section 3.2. Investigating such traversals over different data layouts would also be interesting; however, it may place some requirements on the graph's structure. Specifically, BFS-traversal for transform propagation does not naturally extend to scene graphs were not all nodes have transforms — as in the scene graph used for evaluations in this report — since there is no guarantee that any transform node has another transform node as parent.

7.4.2 Have a maximum size for slices

The benefit of storing nodes together in a slice does not scale linearly with the amount of nodes in the slice. The memory and performance overhead, however, does scale linearly. For this reason, it seems good to limit how large a slice can become. Beyond some maximum size, the slice would have to split into two or more. Doing so would bound both the cost of reconstituting a slice and the memory overhead of unused space in slices. Exactly how the splitting should be done requires some research. The resulting splits should still store a contiguous sub-tree each, they should ideally be of similar size, and the splitting algorithm should ideally be of linear time complexity. An exploration of such algorithms meeting those criteria is thus required.

7.5 Conclusion

The results illustrate the importance of data layout on performance. Interestingly, of the studied data layouts, all aside from the naïve layout resulted in very similar performance. The performance difference between different data layouts was of little significance for small scene graphs, but as they grew larger, the difference grew as well — with good data layout, the time spent per node did not grow as much with scene graph size as it did with the naïve layout.

For the largest scene graphs, the performance difference between naïve and non-naïve data layouts was a factor of three. The performance effect of different data layouts was, however, very small for smaller scene graphs. This is because the whole scene graph fits in cache memory and will remain in cache in between benchmark iterations. The difference is larger when the scene graph is flushed from cache in between iterations, confirming this. A real application would likely have a larger working set aside from the scene graph; as such, less of the scene graph can be expected to remain in cache between frames than between iterations in the (no-flush) benchmarks.

The dynamic data structure for maintaining depth-first layout showed promising results. It outperformed a naïve layout by up to a factor of two for large scene graphs and was not slower in any of the benchmarked cases. This shows that the overhead of maintaining the layout is more than compensated by the performance improvements gained by resulting layout. It also resulted in significant memory overhead, however, perhaps limiting its usefulness without further work. It may be possible to achieve more performance benefit with less memory overhead by using application-specific knowledge of the scene graph and what changes it may undergo.

Bibliography

[1] Alok Aggarwal, Jeffrey Vitter, et al. “The input/output complexity of sorting and related problems”. In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[2] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering.* AK Peters/CRC Press, 2018.

[3] Lars Arge, Mark de Berg, and Herman Haverkort. “Cache-oblivious R-trees”. In: *Proceedings of the twenty-first annual symposium on Computational geometry.* ACM. 2005, pp. 170–179.

[4] Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. “Cache-Oblivious Data Structures.” In: *Handbook of Data Structures and Applications.* Ed. by Sartaj Sahni Mehta Dinesh P. 2nd ed. CRC Press, 2018. Chap. 35.

[5] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. “Cache-oblivious B-trees”. In: *Proceedings 41st Annual Symposium on Foundations of Computer Science.* IEEE. 2000, pp. 399–409.

[6] Michael A Bender et al. “A locality-preserving cache-oblivious dynamic dictionary”. In: *Journal of Algorithms* 53.2 (2004), pp. 115–136.

[7] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. “Cache oblivious search trees via binary trees of small height”. In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics. 2002, pp. 39–48.

[8] Ulrich Drepper. *What every programmer should know about memory.* 2007. URL: `https://akkadia.org/drepper/cpumemory.pdf` (visited on 05/20/2019).

[9] Peter van Emde Boas. “Preserving order in a forest in less than logarithmic time”. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975).* IEEE. 1975, pp. 75–84.

[10] Agner Fog. *Optimizing software in C++.* 2018. URL: `https://www.agner.org/optimize/optimizing_cpp.pdf` (visited on 05/20/2019).

[11] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs.* 2018. URL: `https://www.agner.org/optimize/microarchitecture.pdf` (visited on 05/20/2019).

[12] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[13] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[14] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. No. 248966-041. Intel Corporation. 2019.

[15] *Irrlicht Engine*. URL: `http://www.irrlicht.sourceforge.net/` (visited on 05/20/2019).

[16] Norman P Jouppi. “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 18. 2SI. ACM. 1990, pp. 364–373.

[17] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. “When prefetching works, when it doesn’t, and why”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.1 (2012), p. 2.

[18] Sally A McKee. “Reflections on the memory wall.” In: *Conf. Computing Frontiers*. 2004, p. 162.

[19] Sparsh Mittal. “A survey of recent prefetching techniques for processor caches”. In: *ACM Computing Surveys (CSUR)* 49.2 (2016), p. 35.

[20] NVidia. *NVidia Pro Pipeline*. URL: `https://developer.nvidia.com/nvidia-pro-pipeline` (visited on 05/20/2019).

[21] NVidia. *SceniX*. URL: `https://www.nvidia.com/object/scenix.html` (visited on 05/20/2019).

[22] Ogre3D. *Ogre 2.1 FAQ: What happened to the SceneManagers?* URL: `https://wiki.ogre3d.org/tiki-index.php?page_ref_id=2191#What_happened_to_the_SceneManagers_e.g._OctreeSceneManager_BSP_etc_` (visited on 05/28/2019).

[23] *Ogre3D*. URL: `https://www.ogre3d.org/` (visited on 05/20/2019).

[24] Ogre3D. *What version to choose*. URL: `https://www.ogre3d.org/about/what-version-to-choose` (visited on 05/28/2019).

[25] *OpenGL Mathematics (GLM)*. URL: `https://glm.g-truc.net/` (visited on 05/20/2019).

[26] *OpenSceneGraph*. URL: `http://www.openscenegraph.org/` (visited on 05/20/2019).

[27] Harald Prokop. “Cache-oblivious algorithms”. Master’s thesis. Massachusetts Institute of Technology, 1999.

[28] Marcus Roth, Gerrit Voss, and Dirk Reiners. “Multi-threading and clustering for scene graph systems”. In: *Computers & Graphics* 28.1 (2004), pp. 63–66. ISSN: 0097-8493. DOI: `https://doi.org/10.1016/j.cag.2003.`

10.004. URL: http://www.sciencedirect.com/science/article/pii/S0097849303002310.

[29] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. "Towards a theory of cache-efficient algorithms". In: *Journal of the ACM (JACM)* 49.6 (2002), pp. 828–858.

[30] Kang G Shin and Parameswaran Ramanathan. "Real-time computing: A new discipline of computer science and engineering". In: *Proceedings of the IEEE* 82.1 (1994), pp. 6–24.

[31] *VulkanSceneGraph*. URL: http://www.vulkanscenegraph.org/ (visited on 05/20/2019).

[32] Michael Wörister et al. "Lazy incremental computation for efficient scene graph rendering". In: *Proceedings of the 5th high-performance graphics conference*. ACM. 2013, pp. 53–62.

[33] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[34] Sung-Eui Yoon and Dinesh Manocha. "Cache-efficient layouts of bounding volume hierarchies". In: *Computer Graphics Forum*. Vol. 25. 3. Wiley Online Library. 2006, pp. 507–516.