Dynamical.JS: A composable framework for online exploratory visualization of arbitrarily-complex multivariate networks

Dynamical.JS: A composable framework for online exploratory visualization of arbitrarily-complex multivariate networks

Robert Lee Dotson

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

March 2023

Abstract

Multivariate networks (henceforth, *graphs*) represent entities (*vertices* or nodes), their relationships to each other (*edges*), and manifest or derived data about both (*attributes*). Graphs easily map onto real-world entities and relationships. Therefore the visual analysis of such provide a valuable proxy for the analysis of concrete entities. Because graphs are so effective in modeling, they are ubiquitous: graphs are used in the design and construction of complex circuitry, GIS (Geographic Information Systems), database modeling, social networking, and neuroscience, among other fields. Graph visualization is challenging, and domain-specific requirements further complicate the production of intelligible visual representations. The datasets underlying modern graphs have grown exponentially, complicating visualization tasks, whether *static*, such as in a printed scientific paper or poster, or *dynamic*, where the graph, the visualization, or both evolve. Exploratory visualization of *dynamical* graphs requires the visualizations to evolve due to externally-generated events in *realtime* while preserving the contents of the analyst's "mental map."

This thesis presents a novel, modular, composable Javascript API (Application Programming Interface) for organizing and applying the algorithmic operations and data structures required at each step of visualizing large, arbitrarily complex graphs while allowing the extensions and constraints required by specific application domains. To do so, we abstract quintessential graph layout tasks into EGOs (Exploratory Graph Operations), design an API exposing these operations, and demonstrate the synthesis of well-documented layout algorithms. Finally, we introduce a reference implementation in Javascript and WebGPU (GPU Computing for the Web) to visualize complex graphs in realtime and facilitate future research and development of novel approaches to these problems.

Acknowledgements

I want to thank Dr. Stratos Idreos for his guidance and encouragement as my professor in Data Systems and Big Data Systems and as my thesis director. Without his encouragement to carefully consider the meaning of success, I would not have been able to finish this process. I also want to thank my professor and mentor, Dr. Zona Kostic. This is not only for helping me formulate and explore the concepts underlying this thesis but also for allowing me to teach alongside her in The Art and Design of Information and Building Interactive Web Applications for Data Analysis. This experience will shape my professional and academic life for years to come. I want to thank Dr. Hongming Wang, my research advisor, for her guidance and encouragement while shepherding me through the thesis process. I would also like to thank my husband, Paul Lordan, for his endless patience while listening to me scream at my computer screen every day and night for a year. And finally, I would like to thank Pancakes, my brave rescue pup, without whom I wouldn't have survived the COVID-19 pandemic.

Table of Contents

List of Figures x
List of Tables xii
List of Equations xiii
List of Algorithms xiv
List of Code xv
Chapter 1. Introduction 1
1.1 Background 2
1.2 The Problem: Interactivity 4
1.3 The Solution: Dynamical.JS 6
1.4 Prior Work 7
1.5 Terminology 8
Chapter 2. Design & Methodology 11
2.1 Key Qualities 11
2.2 System Agnosticism 14
2.3 Modularity 14
2.3.1 Data Module 15
2.3.2 Layout Module 16
2.3.3 Drawing Module 17
2.3.4 Testing & Performance Module 17
2.4 Graph & Layout Types 18
2.4.1 Static Graph Layouts 19
2.4.2 Dynamic (Offline) Graph Layouts 21
2.4.3 Dynamical (Online) Graph Layouts 23
2.4.4 Context Aware Layouts 24

2.5 Aesthetics and the Mental Map 25
2.5.1 Preserving the Mental Map 26
2.5.2 Multiple Representation 26
2.5.3 Small Multiples 27
2.6 Cache Consciousness 28
2.6.1 GPU Memory 29
2.7 Exploratory Graph Operations (EGOs) 31
2.8 Graph Data Operations (GDOs) 33
2.8.1 Load Graph 33
2.8.2 Store Graph 34
2.8.3 Clone Graph 34
2.8.4 Subgraph 35
2.8.5 Merge Graphs 35
2.8.6 Retrieve Element(s) 35
2.8.7 Filter Elements 35
2.8.8 Add Element(s) 36
2.8.9 Remove Element(s) 36
2.8.10 Update Element(s) 36
2.8.11 Merge Element(s) 37
2.8.12 Materialize 37
2.8.13 Key Frame 37
2.8.14 Coalesce 37
2.8.15 Calculate Graph Statistics 38
2.8.16 Find Path 38
2.8.17 Neighbors 38
2.9 Graph Layout Operations (GLOs) 38

2.9.1 Layout . . . 38
2.9.2 Initial Layout . . . 39
2.9.3 Merge Layouts . . . 39
2.9.4 Calculate Layout Statistics . . . 39
2.9.5 Simplify . . . 40
2.9.6 Elaborate . . . 40
2.9.7 Partition . . . 40
2.9.8 Merge Partitions . . . 40
2.9.9 Calculate Forces . . . 41
2.9.10 Calculate Repulsive Forces . . . 41
2.9.11 Calculate Attractive Forces . . . 41
2.9.12 Calculate Offsets . . . 41
2.9.13 Centroid . . . 41
2.9.14 Place Node(s) . . . 42
2.9.15 Quality . . . 42
2.9.16 Finesse . . . 42
2.10 Graph Rendering Operations (GROs) . . . 42
2.10.1 Update Dimensions . . . 43
2.10.2 Apply Transform . . . 43
2.10.3 Render Layout . . . 43
2.10.4 Render Region . . . 43
2.10.5 Overlay . . . 44
2.10.6 Show Element(s) . . . 44
2.10.7 Hide Element(s) . . . 44
2.10.8 Apply Visual Mark to Element(s) . . . 44
2.10.9 Apply Visual Channel to Element(s) . . . 44

2.10.10 Interpolate 45
2.11 Graph Analysis Operations (GAOs) 46
2.11.1 Find Element(s) by Position 46
2.11.2 Find Bounded Element(s) 46
2.11.3 Select Element(s) 46
2.11.4 Deselect Element(s) 47
2.11.5 Filter Element(s) 47
2.11.6 Highlight Element(s) of Interest 47
2.11.7 Export Image 47
2.11.8 Step Backward and Forward through GRO History 47
2.12 Tunability 48
2.12.1 Environmental Constraints 48
2.12.2 Layout Constraints 49
2.12.3 Quality Metrics 50
Chapter 3. Layout Algorithm Decomposition 51
3.1 Algorithm Stages 52
3.1.1 Initial Placement 54
3.1.2 Statistics Calculation 54
3.1.3 Partition 55
3.1.4 Placement Loop 56
3.1.5 Merge Partitions 56
3.1.6 Finesse 56
3.2 Basic Algorithms 57
3.2.1 Random Placement 58
3.2.2 Geometric 58
3.2.3 Fixed & Free 60

3.3 Force-Directed Layouts 60
3.3.1 The Barycentric Method 63
3.4 Spring Systems 66
3.4.1 Fruchterman-Reingold 69
3.4.2 Edge-Edge Repulsion 71
3.5 Other Algorithms 75
Chapter 4. Computational Development 80
4.1 External Requirements 80
4.2 Publishing 80
4.3 Hardware Configuration 81
4.4 Tools 82
4.5 Javascript 83
4.5.1 Runtime & Concurrency 84
4.5.2 Asynchronous Execution 86
4.6 Computer Graphics 87
4.6.1 History 87
4.6.2 OpenGL 88
4.7 Graphical Object Models 89
4.8 Rendering Pipeline(s) 92
4.8.1 The Geometry Pipeline 93
4.8.2 The Pixel Pipeline 96
4.9 Programmable Pipelines 97
4.9.1 Shaders 98
4.9.2 Shading Languages 101
4.10 General-Purpose GPU Computing (GPGPU) 103
4.10.1 GPGPU Memory & Concurrency 104

4.11 GPGPU Frameworks . . . 106
4.12 Web Graphics . . . 107
4.12.1 WebGL & GLSL . . . 107
4.12.2 GPU for the Web (WebGPU) . . . 108
Chapter 5. Implementation & Results . . . 110
5.1 Code Syntax & Documentation . . . 110
5.2 Environment . . . 111
5.3 Code organization . . . 112
5.4 The `DGBase` Object . . . 113
5.4.1 Extensibility . . . 113
5.5 Asynchronous Interfaces . . . 115
5.5.1 Asynchronous Initialization . . . 115
5.5.2 Asynchronous Loops . . . 116
5.6 Mathematical Operations . . . 117
5.6.1 Vectors & Matrices . . . 117
5.6.2 (Pseudo) Random Number Generators (PRNGs) . . . 119
5.7 Data Structures . . . 122
5.7.1 Graph ADT . . . 123
5.7.2 Index Structures . . . 125
5.7.2.1 Bit Vectors & Matrices . . . 126
5.7.2.2 Bloom Filters . . . 126
5.8 Data Module . . . 126
5.8.1 Materialization . . . 127
5.8.2 GPU Materialization . . . 129
5.8.3 Materialization Data Structures . . . 130
5.8.3.1 Adjacency Lists . . . 132

5.8.3.2 Adjacency Matrices . . . 133
5.8.4 Graph Traversal . . . 134
5.8.5 Graph Updates . . . 135
5.8.5.1 Key Graphs . . . 135
5.8.5.2 Change Sets . . . 135
5.8.6 File Formats . . . 136
5.9 Layout Module . . . 138
5.9.1 Layout Execution . . . 139
5.10 Drawing Module . . . 142
5.10.1 Rendering & Animation . . . 142
Chapter 6. Conclusion . . . 145
6.1 Summary . . . 145
6.2 Challenges . . . 146
6.2.1 Multithreading . . . 146
6.2.2 Memory Limits . . . 147
6.2.3 Limited Optimization . . . 147
6.2.4 Timeslicing . . . 148
6.3 Future Work . . . 148
6.3.1 Architectural Changes . . . 148
6.3.2 Optimization Changes . . . 149
Appendix A. Glossary . . . 151
Appendix B. Acronyms . . . 159
Appendix C. Code . . . 164
C.1 Javascript Code . . . 164
C.1.1 DGBase . . . 164
C.1.2 DGTestRig . . . 174

C.1.3 DGGPUEngineTestRig 178
C.1.4 DGAsyncLoop 180
C.1.5 DGBitVector & DGBitMatrix 182
C.1.6 DGGraphBase 191
C.1.7 DGGraph 194
C.1.8 DGQuickBloom 214
C.1.9 DGLayoutEngineBase 220
C.2 WebGPU Code 229
C.2.1 DGWebGPUBase 229
C.2.2 DGComputeEngine 236
C.3 WGSL Code 263
C.3.1 XOR128 263
References 266

List of Figures

1.1 A node-link diagram with both on-node/on-edge encoding. 1
1.2 A Unified Markup Language (UML) network diagram, utilizing complex nodes and edges. 2
1.3 A network diagram of Barcelona's subway system encoding the distance between stations in 5-minute intervals, utilizing on-node/on-edge encodings. 7

2.1 A UML (Unified Modeling Language) block diagram of Dynamical.JS modules. 15
2.2 A block diagram of the Dynamical.JS Data Module. 16
2.3 A block diagram of the Dynamical.JS Layout Module. 17
2.4 A block diagram of the Dynamical.JS Drawing Module. 18
2.5 Graph edge classification. 20
2.6 A comparison of CPU (Central Processing Unit) and GPU (Graphics Processing Unit) memory layouts. 30

4.1 Graphics rendering pipeline stages. 92
4.2 The "classic" rendering pipelines. 93

5.1 A high-level UML package diagram of the Dynamical.JS framework. . 112
5.2 A UML package diagram for the Dynamical.JS `common` package. . . . 113
5.3 A UML class diagram of the Dynamical.JS `DGAsyncLoop` interface. . . 117
5.4 A UML class diagram of the Dynamical.JS matrix sub-package. . . . 118
5.5 A UML class diagram of the Dynamical.JS `random` sub-package. . . . 121
5.6 A UML class diagram of the Dynamical.JS ADT sub-package. 122

5.7 A UML class diagram of the Dynamical.JS graph ADT (Abstract Data Type) sub-package. 124
5.8 A UML class diagram of the Dynamical.JS Data Module. 127
5.9 A UML class diagram of the Dynamical.JS Layout Module. 138
5.10 A UML class diagram of the Dynamical.JS Drawing Module. 143

List of Tables

2.1 Common symbol notation for GDOs (Graph Data Operations). . . . 34
2.2 Common symbol notation for Layout GLOs (Graph Layout Operations). 39
2.3 Common symbol notation for GROs (Graph Rendering Operations). . 42
2.4 Environmental constraint parameters. 48
2.5 Layout constraint parameters. 49
2.6 Layout quality metrics. 50

3.1 Common symbol notation for force-directed algorithms. 63
3.2 Common symbol notation for spring algorithms. 66

5.1 Time complexity cost of operations on graph ADTs implemented as adjacency lists 132
5.2 Time complexity cost of lookup operations on adjacency lists implemented with `Array` or `Map`. 133

List of Equations

2.1 Undirected edge 19

2.2 Directed edge 20

2.3 Mixed edge 20

2.4 Mixed, multiple edges 20

2.5 Simple linear interpolation 45

2.6 Linear interpolation of homogeneous coordinates 45

2.7 Perspective interpolation of homogeneous coordinates 46

3.1 Barycenter force 64

3.4 Log-spring force 67

3.4 Inverse-square log-spring force 67

3.4 Alternate inverse-square log-spring force 67

3.5 Spring length 67

3.6 The force-directed energy function 67

3.7 Ideal distance between two nodes 68

3.8 Spring strength 68

3.11 Fruchterman-Reingold Repulsive Force 71

3.11 Fruchterman-Reingold attractive force 71

3.11 Fruchterman-Reingold optimal distance 71

List of Algorithms

3.1 The LAYOUT algorithm 53
3.2 The BASIC layout algorithm 57
3.3 The RANDOM layout algorithm 58
3.4 The POLYGON layout algorithm 59
3.5 The FIXED-FREE layout algorithm 59
3.6 The FORCE-DIRECTED algorithm 62
3.7 The barycentric layout algorithm by Tutte (1963). Pseudocode from Di Battista et al. (1999) 64
3.8 The BARYCENTER layout algorithm with EGOs. 65
3.9 The spring embedder layout algorithm by Eades (1984). Pseudocode from Kobourov (2013) 69
3.10 The SPRING layout algorithm with EGOs. 70
3.11 Pseudocode for the force-directed layout algorithm by Fruchterman & Reingold (1991) 72
3.12 The FRUCHTERMAN-REINGOLD layout algorithm with EGOs. 73
3.13 The edge-edge repulsion algorithm by Lin & Yen (2005) 77
3.14 The EDGE-EDGE-REPULSION algorithm with EGOs 78
4.1 Javascript event loop 85

List of Code

5.1 The HTML development "rig" for Dynamical.JS 111

5.2 The `DGExtensibleBase` interface. 114

5.3 Concrete subclass registering with a class cluster. 114

5.4 Example code demonstrating asynchronous initialization. 115

5.5 The basic Javascript interface for `DGGraph` 125

5.6 An example graph serialized into the JSON (JavaScript Object Notation) file format 137

5.7 An example graph serialized in the JSON (indexed) file format 138

5.8 The `DGLayoutEngineBase` implementation of the LAYOUT GLO. . . . 140

C.1 Javascript code for `DGBase`. 164

C.2 Javascript code for `DGTestRig`. 174

C.3 Javascript code for `DGGPUEngineTestRig`. 178

C.4 Javascript code for the `DGAsyncLoop` Interface. 180

C.5 Javascript code for `DGBitVector` & `DGBitMatrix`. 182

C.6 Javascript code for `DGGraphBase`. 191

C.7 Javascript code for `DGGraph`. 194

C.8 Javascript code for `DGBloomFilterBase` & `DGQuickBloom`. 214

C.9 Javascript code for `DGLayoutEngineBase`. 220

C.10 Javascript code for `DGWebGPUBase`. 229

C.11 Javascript code for `DGComputeEngine`. 236

C.12 WGSL code for the `XOR128` pseudo-random number generator. 263

Chapter I.

Introduction

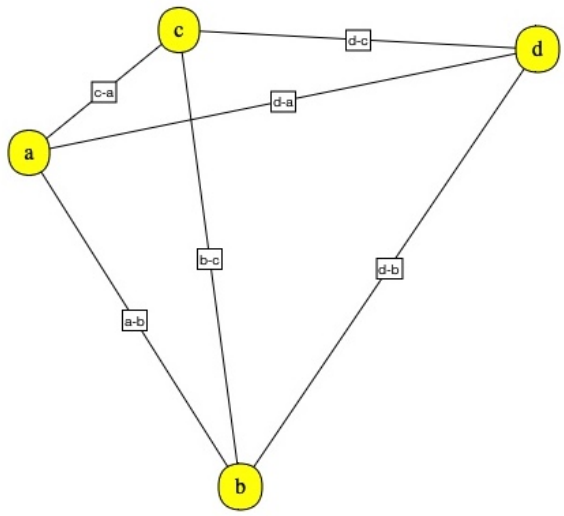


Figure 1.1. A node-link network diagram with both on-node and on-edge encoding[1].

Arbitrarily-complex multivariate network (henceforth, *graph*) drawings are visual representations of relational datasets (Tutte, 1963). Graph drawing is a standard means of visualizing relational data in many fields, including social networking, software engineering (Figure 1.2), bioinformatics, telecommunications, logistics, transportation (Figure 1.3), and hierarchical taxonomy (Nobre et al., 2019). Most data presentation problems posed by these analysis domains involve drawing n-dimensional graphs on 2D (two-dimensional) surfaces such as a page or screen (Di Battista et al., 1994; Beck et al., 2013). Graph drawings' ultimate utility is conveying salient domain-specific information to the analyst (Nobre et al., 2019, 2020); therefore, these drawings must emphasize pertinent features and minimize extraneous details. Aesthetic optimization of graph drawings for visual emphasis and analytical insight imposes domain-specific constraints on algorithm designers (Tutte, 1963); exploratory opti-

[1] The author created this graph with a prototype of Dynamical.JS, but it was drawn using Canvas2D as a final project for CSCI E14A at the Harvard Extension School.

mization imposes additional constraints, further complicating implementation (Du et al., 2017). Proposing an exploratory graph visualization framework design that successfully navigates these constraints is the primary goal of this thesis.

1.1. Background

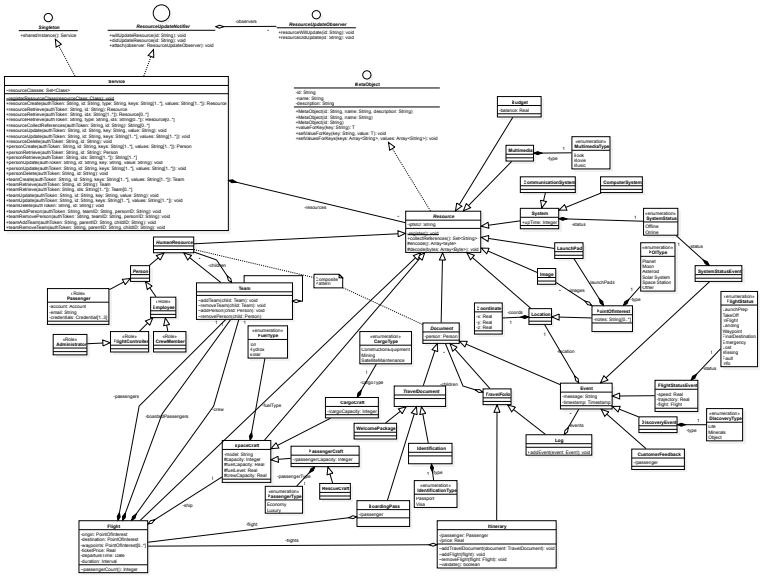


Figure 1.2. A UML (Unified Modeling Language) network diagram, utilizing complex nodes and edges[2].

Understanding large and growing relational datasets is a defining feature of the current computational landscape. Users, analysts, and artificial intelligentsia (AI) must increasingly understand the relationships within these datasets as their data evolve in realtime (Lin & Yen, 2005). As data volume and growth accelerate, the urgency and difficulty of exploring these relationships accelerate concomitantly (Idreos et al., 2018a,b). Algorithmic and aesthetic methods for displaying large and growing networks have diverged, lagging far behind current technological capabilities (Nobre et al., 2019). New algorithms, metrics, and implementation strategies must be

[2] This graph was created by the author as part of an assignment for CSCI E97 at the Harvard Extension School.

developed to meet this challenge (Namata et al., 2007; Idreos et al., 2018b). While AI has achieved prominence in the mind-space for data analysis, visual exploration using human intelligence remains a clear requirement (Stolper et al., 2014). There are many domain-specific tools for exploring datasets with *specific* properties. As a class, these tools are expensive, poorly maintained, infrequently updated, and platform-bound (Mockus, 2019). Generalized, freely-available tools and techniques for exploratory graph exploration present an open opportunity (Nobre et al., 2019).

Graph-theoretical analysis of relational datasets is a robust field and is well-documented in the literature (Tutte, 1963; Di Battista et al., 1994; Nobre et al., 2019). The field of graph visualization is likewise very mature (Tutte, 1963; Di Battista et al., 1994); thousands of visualization techniques for *small* graphs ($|G| \leqslant 10^3$) are well understood. Several graph drawing algorithms capable of drawing large ($|G| \geqslant 10^4$) and very large ($|G| \geqslant 10^5$) domain-specific graphs, such as VLSI (Very Large Silicon Integrated circuit) design, taxonomy, or bioinformatics research (Nobre et al., 2019) have also been published. Nevertheless, few are appropriate for rendering to computer screens or printed pages.

As datasets and their resultant graph drawings grow arbitrarily large and complex, most graph layout algorithms falter due to computational constraints. Even low-complexity algorithms have a time complexity of $\gtrapprox \mathrm{O}(|V|^3)$, or more precisely, $\mathrm{O}(|V|^3 \times \ell)$, where ℓ is the number of inner loops that must be performed on each node for optimal placement (Kamada & Kawai, 1989). To facilitate the creation of effective drawings within a bounded time frame, algorithms must employ specialized data structures tuned to available resources (Idreos et al., 2018b), exploit GPUs (Graphics Processing Units) or other specialized hardware (Bleiweiss, 2008; Jeowicz et al., 2013), or simplify the underlying graph to maintain "readability" (Dunne & Shneiderman, 2013). The final output medium also imposes a resolution constraint,

imposing an upper bound on any potential insights gathered from densely-connected graphs (Dunne & Shneiderman, 2013).

To circumvent these constraints, the structure and temporality of any dataset must be considered, and many dynamic algorithms have been invented to do so (Nobre et al., 2019). Successful visualization of evolving datasets is wholly dependent on maintaining readability in the face of (un)expected changes. During the presentation of any dynamical graph visualization (Section 1.5), maintaining an analyst's ability to gain insight from exploratory tasks is paramount (Di Battista et al., 1994), second only to increasing the quantity of salient data presented (Tutte, 1963). The selection of graph drawing algorithms that maintain insights within an analyst's "mental map" and maximize the *information density* remains an open problem (Misue et al., 1995; Archambault et al., 2011; Nobre et al., 2020).

1.2. The Problem: Interactivity

As indicated above, devising optimal visualization techniques for arbitrarily complex graphs is an NP-hard problem, even when limited to static datasets (Nobre et al., 2019). However, there are well-defined *exact* solutions for specific classes of small graphs using eigenvectors or spectral analysis (Koren, 2005). Several methods for animating interpolations between *static* graph drawings are also extant in the literature (Cohen et al., 1992; Du et al., 2017). The theoretical usefulness of such animations has been explored in depth (Misue et al., 1995), but these examinations fail to consider realtime performance. Thus, their interactive utility remains equivocal (Archambault et al., 2011). The analyses that take performance into account optimize over precomputed, serialized representations of *past* graph morphology (Section 2.4.2) and are, therefore, inappropriate for interactive exploration due to expensive rematerialization of the underlying data structures (Smith, 1984; Beck et al., 2012).

New, GPU-accelerated algorithms have been developed to generate layouts of huge ($|G| \geqslant 10^6$) graphs but are limited to *sparse* graphs with many nodes and few edges (Mi et al., 2016).

Interactivity with dynamical (progressively evolving) relational datasets is a poorly defined, nascent field of study (Hadlak et al., 2011; Sheng et al., 2019; Nobre et al., 2019). However, in recent years, new techniques for generating interactive graph visualization from dynamic, generalized graphs have appeared (Frishman & Tal, 2008; Wei et al., 2018; Simonetto et al., 2020). Unfortunately, these techniques are both highly domain-specific and unwieldy; dataset evolution is limited to *retrogressive* changes, and animated meta-visualizations incur enormous computational costs due to precomputed transitions and *data amplification* (Koren, 2005; Lin & Yen, 2005; Sheng et al., 2019). Progressively evolving datasets grow and shrink by arbitrary numbers of entities as time moves forward, necessitating novel structural and operational approaches (Idreos et al., 2018b) to (re)visualize a graph in response (Nobre et al., 2019). When dynamic visualization techniques are employed, rapid and unexpected changes to entity relationships may introduce unnoticed errors and obscure ephemeral insights (Simonetto et al., 2020). Online visualizations entail processing an event stream and incorporating arbitrary modifications in realtime (Udupa et al., 2009). Such changes necessitate adaptive dynamic graph layout algorithms that capture changes in morphology due to "normal" temporal evolution simultaneously with changes due to unpredictable asynchronous events (Hadlak et al., 2011; Simonetto et al., 2020). Extant methods use interpolation to animate between two *completed* graph layouts smoothly; unfortunately, expensive *data amplification* costs (Smith, 1984) may lead to UI (User Interface) lock-up, animation stutter, or lost frames if the number of graph elements differs between layouts (Archambault & Purchase, 2013; Simonetto et al., 2020).

1.3. The Solution: Dynamical.JS

This thesis seeks to determine whether an extensible, multi-platform, real-time exploratory graph visualization is feasible on commodity hardware. In addition to validating this feasibility hypothesis, this thesis aims to produce a useful and robust framework for application development and facilitate new expansion in the field of exploratory data visualization. To address the need for online-interactive graph exploration tools, we present a novel design for a standalone, embedded graph visualization framework runnable on any system hosting a modern web browser: Dynamical.JS. Our initial framework design enables the exploration of dynamical-relational datasets, visually represented as animated 2D graph drawings. Both the design and reference code presented in this thesis was developed using three substrate-agnostic, standards-based technologies: the Javascript platform (European Association for Standardizing Information and Communication Systems, 2020), WebGPU (GPU Computing for the Web) framework (World Wide Web Consortium, 2022b) and WGSL (WebGPU Shading Language) (World Wide Web Consortium, 2022c). As this development approach relies on yet-to-be-finalized technologies and standards, the reference code implementation included with this work is necessarily incomplete; further, any performance metrics presented below are purely analytical. The relevant standards bodies (World Wide Web Consortium, 2022d; European Association for Standardizing Information and Communication Systems, 2020; Apple Inc., 2021) have indicated these standards are nearing completion, and we intend to publish the completed framework publicly once the standards' implementation is generally available.

This thesis is structured as follows: Chapter 1 explores previous work in the field of exploratory graph visualization and exposes potential pitfalls and opportunities presented in the literature (Nobre et al., 2019; Di Battista et al., 1994; Ar-

chambault et al., 2011). Chapter 2 describes key qualities a successful dynamical graph visualization system must possess, defines a set of required EGOs (Exploratory Graph Operations) manifesting these qualities, and outlines the modular structure of Dynamical.JS, which incorporates these EGOs. In Chapter 3, we decompose several graph drawing algorithms, both classical and modern, and recompose them using the EGOs defined in Chapter 2. Chapter 4 describes our development methodology and design choices. Chapter 5 demonstrates the exploitation of modern software programming techniques and technologies to achieve our design goals using Javascript and WebGPU. Finally, Chapter 6 concludes and discusses future work.

1.4. Prior Work

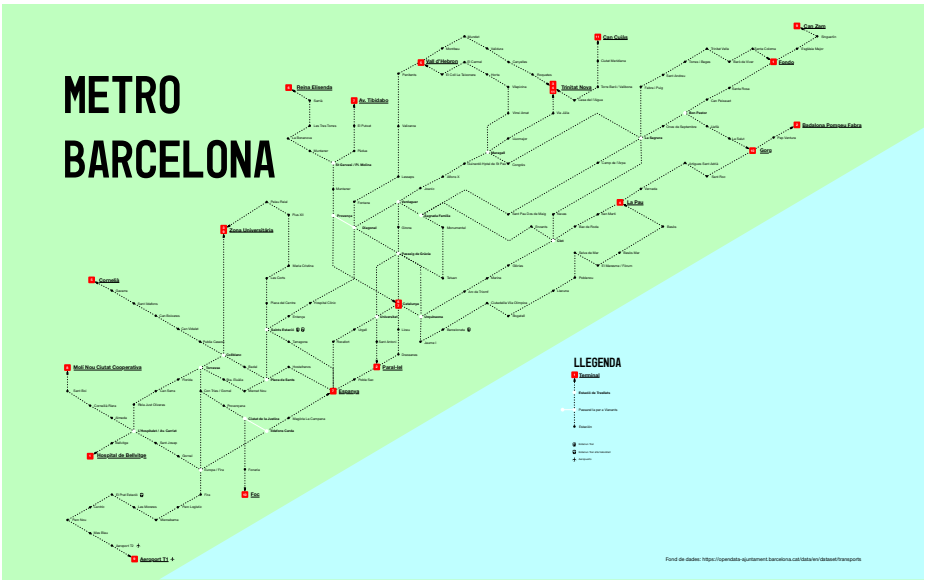


Figure 1.3. A network diagram of Barcelona's subway system encoding distance between stations in 5-minute intervals, utilizing various on-node and on-edge encodings[3].

,

[3] This graph was created by the author as part of an assignment for CSCI E79 at the Harvard Extension School.

Software packages capable of realtime updates to graph visualizations exist, yet few are well-documented open-source packages (Nobre et al., 2019, 2020). While D3 (Bostok, 2021) and NetworkX (NetworkX Developers, 2020) are powerful and capable of presenting small dynamic graphs, both have severe limitations, rendering them inappropriate for large, interactive dynamical graph visualizations. D3 is limited to relatively small graphs due to its reliance on the DOM (Document Object Model), SVG (Scalable Vector Graphics), and the need to function as a generalized data visualization package. NetworkX is a powerful tool specifically designed for network *analysis* but not drawing (Hagberg et al., 2022); it relies on many external packages to perform the layout and rendering. NetworkX also requires a relatively complex server-side installation which may be out of reach for many client-side developers, limiting its availability and usefulness for interactive dynamical graph visualizations. Other packages which run in a web browser, such as Sigma.js (Jacomy & Plique, 2022) and Graphology (Plique, 2022), are fully synchronous. Without extensive modifications, both can lock up the browser for extended lengths of time when processing large graphs or require running node.js (OpenJS Foundation, 2022) as a background service.

1.5. Terminology

This thesis uses several non-standard terms to disambiguate certain concepts and terms with multiple, interchangeable definitions within the literature or where the same term is used for different concepts in the same or different fields. The most important of these are the following:

Substrate This non-standard term always refers to the collection of CPU (Central Processing Unit), GPU, and ASIC (Application-Specific Integrated Circuit) re-

sources provided by a given hardware configuration. This includes instruction sets, *compute cores*, co-processors, memory configurations, or other control hardware to facilitate computation. Substrates may be *monolithic* (confined to a single host), *distributed* (shared among multiple hosts), or *virtual* (logical partitions of the above resources which may operate concurrently on the same host).

System Unless qualified, this refers specifically to the OS (Operating System) running on a given substrate. Windows, macOS, and Linux are examples of systems.

Framework A collection of software functions, APIs (Application Programming Interfaces), code libraries, or other technologies a given *system* provides to perform related computational tasks. OpenGL (Open Graphics Library), DirectX, WebGPU, and Dynamical.JS are examples of frameworks.

Platform A specific collection of *frameworks*, execution environments, or programming languages one or more *applications* require for execution. The Javascript and Python runtimes are examples of platforms.

Application An interactive *program* or *process* that runs on a specific *platform* to produce output useful to the *analyst*. Adobe Photoshop, Microsoft Word, and OmniGraffle are examples of applications.

Analyst A human who interacts with an *application* to solve a problem or to gain insight by analyzing its output.

Dynamic Any object, structure, or concept that changes in a predefined or predictable manner or has a known *retrogressive* evolution.

Dynamical Any object, structure, or concept that evolves *progressively* or changes in an unstructured or unpredictable manner in response to external events. A cloud computing service that reallocates resources due to fluctuations in

current client demand is an example of a dynamical *substrate*.

Node The fundamental graph-theoretical unit from which all *graphs* are formed.

Vertex A data structure that describes a point in two or 3D (three-dimensional) space and is the fundamental unit from which all graphical primitives are derived in a given computer graphics *framework*. This term also refers to a *graph node* that has been materialized for layout and rendering.

Additional terms will be defined when they are used or listed in the glossary.

Chapter II.

Design & Methodology

Designing software is complex; designing *reusable* software is even more so. Software engineers must *not* attempt to solve every problem from first principles; instead, they should reuse solutions that have worked in the past (Gamma et al., 1995). A framework acts as an isolation layer to shield the application developer from the implementation details; the framework's design must collect prior solutions into a standard format the developer can apply immediately and, in turn, provide these solutions to the analyst. The framework should *abstract* and *encapsulate* many recurring tasks so the developer need not rediscover or reimplement them (Jargstorff, 2004). To do so, the framework architect must isolate pertinent concepts, factor them into classes at the right granularity, define interfaces and inheritance hierarchies, and establish key relationships among them (Gamma et al., 1995). The framework must be specific to the existing problem of exploratory graph visualization but remain general enough to address future issues and requirements.

In the following sections, we describe the basic design of an exploratory graph visualization framework: Dynamical.JS. This framework demonstrates exploratory graph visualization's most important design and implementation aspects. However, it does not address many of the more complicated schemes necessary for an industrial-strength library, which is reserved for future work.

2.1. Key Qualities

A practical exploratory graph visualization framework must possess several key qualities and express these qualities through its architectural design. The framework must:

1. *Run on as many systems as possible.* System agnosticism (Section 2.2) allows a developer to write the framework once and an analyst to run framework-dependent applications on various systems *without modification*.

2. *Incorporate asynchronous operation throughout.* Dynamical graph visualization is by nature extemporaneous; changes to graph morphology are unpredictable (Udupa et al., 2009). Therefore, any graph exploration framework must respond in realtime to unexpected events without blocking the UI (User Interface) or otherwise capturing resources unnecessarily (Section 4.5.2).

3. *Organize functionality into logical modules.* Modules separate related computational tasks into self-contained units organized by functional responsibility (Section 2.3). Modules are further divided into submodules and provide standardized interfaces using common *design patterns* (Gamma et al., 1995). Encapsulation ensures application developers are exposed only to functionality appropriate to execute an analytical task; framework developers remain free to define and install new modules without modifying the framework structure.

4. *Support the drawing of a large number of different graph types.* At a minimum, the framework must lay out and draw directed, undirected, and weighted graphs (Di Battista et al., 1994). The framework must also be extensible enough to support special layouts for (a)cyclic, hierarchical, and domain-specific graph types (Section 2.4).

5. *Retain the aesthetic qualities and preserve the mental map whenever possible.* During exploratory tasks, the graph structure may be unknown until after the exploration task has commenced (Kreylos & Hamann, 2001); *post hoc* (re)interpretation of attributes may change the target graph's connectivity or

classification (Simonetto et al., 2020). (Re)applying a particular graph layout algorithm to an externally modified graph may destroy the analyst's mental map (Section 2.5.1). It is necessary to provide alternatives that adapt appropriately to a graph's current state (Misue et al., 1995; Archambault et al., 2011).

6. *Leverage cache-conscious memory layouts, avoid re-materializing inputs, and minimize memory copies between GPU (Graphics Processing Unit) and CPU (Central Processing Unit).* This minimizes stalls due to cache misses (Section 2.6) and reuses intermediate values wherever possible (Beck et al., 2012; Idreos et al., 2018b; Tan et al., 2020).

7. *Define an extensible library of composable EGOs (Exploratory Graph Operations) and mathematical functions.* A standard EGO library (Section 2.7) enables the replication of extant layout strategies and the synthesis of novel techniques from pre-defined components (Stolper et al., 2014). Extensibility empowers the programmer to provide novel or proprietary EGOs without modifying the underlying framework (Section 5.4.1).

8. *Record and expose tunable parameters as a transparent interface.* Analysis domains, presentation requirements, and performance goals impose functional *constraints* on algorithm execution and control graph evolution (Du et al., 2017). If exposed through a consistent interface, these constraint parameters can be shared among a suite of EGOs or saved as profiles to automatically apply appropriate resources to a particular EGO regardless of the underlying computational substrate (Section 2.12) (Beck et al., 2013).

The following sections describe how Dynamical.JS manifests these qualities in its design. Individual optimization of these key qualities could quickly expand into

independent projects outside this document's scope, as is our intent. However, due to time and platform constraints, we have prioritized a naive implementation of these key qualities for demonstration purposes, reserving optimizations for future work.

2.2. System Agnosticism

The primary design goal of Dynamical.JS is a self-contained, dependency-free, standards-based framework. This may seem like "reinventing the wheel," however, it is essential to realize that the open-source supply chain is fragile and is littered with abandoned or poorly maintained projects (Mockus, 2019). By taking a standards-only approach, Dynamical.JS is built on a foundation that is relatively immune to unpredictable dependency changes and unburdened by onerous licensing restrictions.

2.3. Modularity

A successful framework for exploratory graph visualization must be modular, divide responsibility among various components, and provide programmers with logical groupings of functionality when using or extending the framework. Modularity extends from the framework level down to base functionality, using object-oriented *design patterns* and standardized programming interfaces (Gamma et al., 1995). Object orientation encapsulates both data and the procedures which operate on that data; entities encapsulated in this manner cannot be modified without interacting with the appropriate interface. Dynamical.JS is organized into the following three discrete modules and one distributed virtual module: (1) Data, (2) Layout, (3) Drawing, and (4) Testing/Performance.

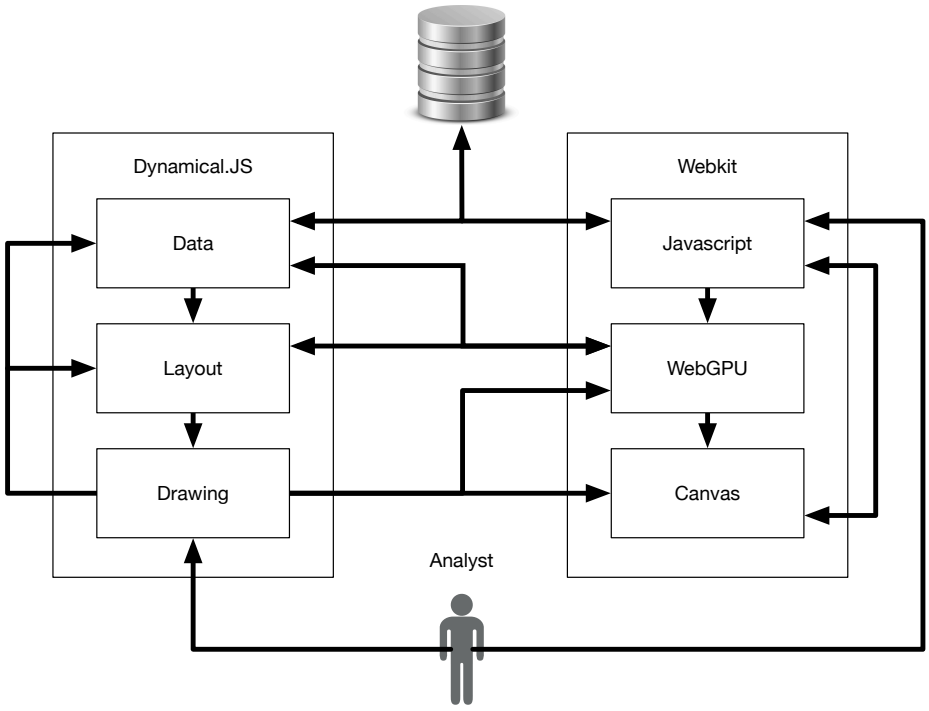


Figure 2.1. A UML (Unified Modeling Language) block diagram of Dynamical.JS modules.

2.3.1 Data Module

The Data Module is responsible for retrieving input data and assembling graphs sourced via standard methods: file importation, remote API (Application Programming Interface) calls, and database connections. This module parses the input data and performs the graph-theoretical and statistical operations required for analysis. Once the data are parsed, they reside in appropriate data structures for indexing, retrieval, and modification. The Data Module then transforms, or *materializes*, the current graph state into a format suitable for layout on either CPU or GPU via hints provided by the *Layout Module*. The Data Module also predefines hints to determine the best memory layout strategy to optimize the *locality of reference* (Ward & Halstead Jr., 1999) and directs the *Layout Module* to (re)apply layout algorithms as the underlying graph data change. Future plans for this project include recording

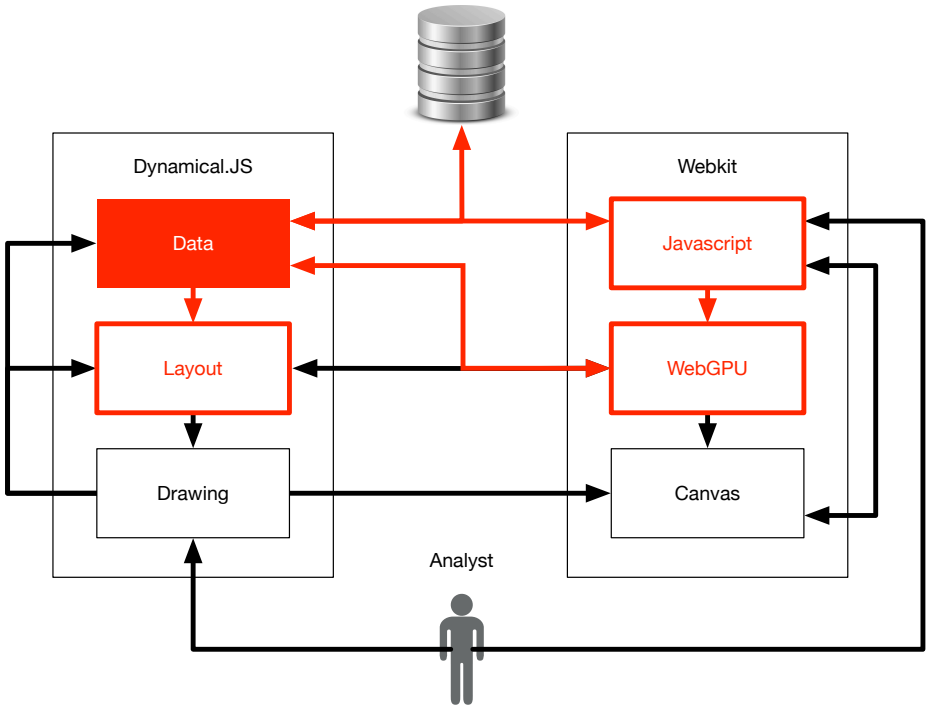


Figure 2.2. A block diagram of the Dynamical.JS Data Module.

and tuning such strategies to increase performance.

2.3.2 Layout Module

The Layout Module is responsible for placing nodes within a coordinate system independent of the analyst's UI, called *layout space*. Using composable GLOs (Graph Layout Operations), this module encapsulates the algorithmic logic required to initialize and calculate a graph layout and reflect the target graph's morphological changes as necessary for animation. This module works in tandem with the *Data Module* to index and retrieve graph elements and their attributes by generating and responding to appropriate events. The Layout Module then notifies the *Drawing Module* when the intermediate or final layout stages are ready for rendering and animation. This module also listens for events generated by the *Drawing Module* to effect changes in morphology due to user interaction, such as selection or filtering.

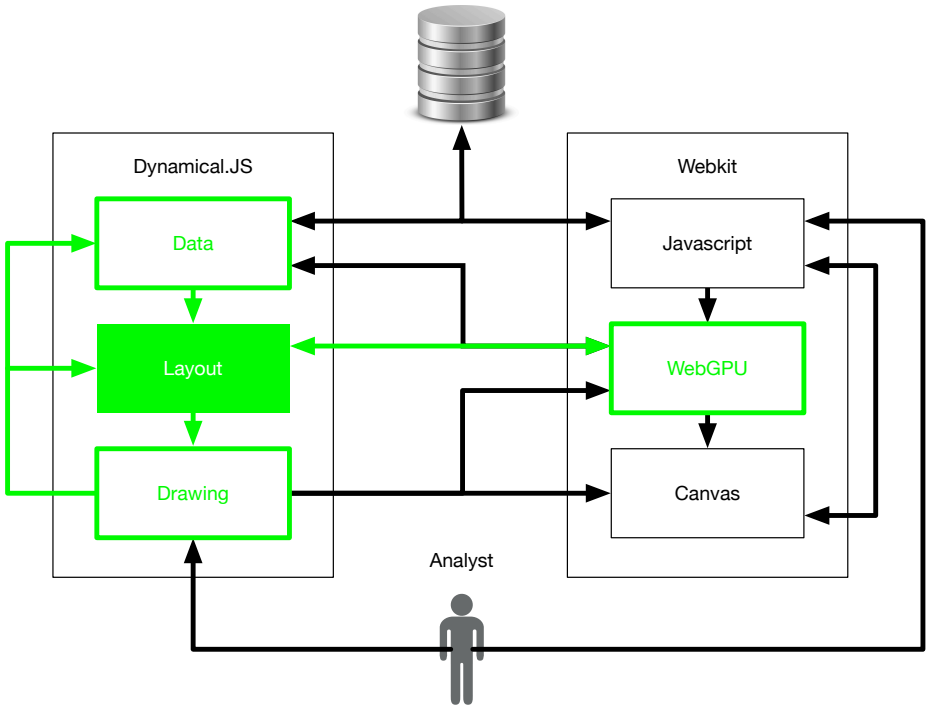


Figure 2.3. A block diagram of the Dynamical.JS Layout Module.

2.3.3 Drawing Module

The Drawing Module has two primary responsibilities: rendering graph layouts to the UI and responding to events generated by the analyst. All drawing functionality is controlled by exchanging asynchronous events with the UI and sister modules: (1) responding to layout change events from the *Layout Module*; (2) exchanging graph selection and query events with the *Data Module*; (3) mediating user input events from HIDs (Human Interface Devices) such as mice or keyboards via the Javascript runtime. This module also determines the animation framerate, view-coordinate transformations, and image texture creation.

2.3.4 Testing & Performance Module

Unlike the other modules described above, the Testing & Performance module is not a segregated package. Still, it is a *logical* grouping of components for measuring,

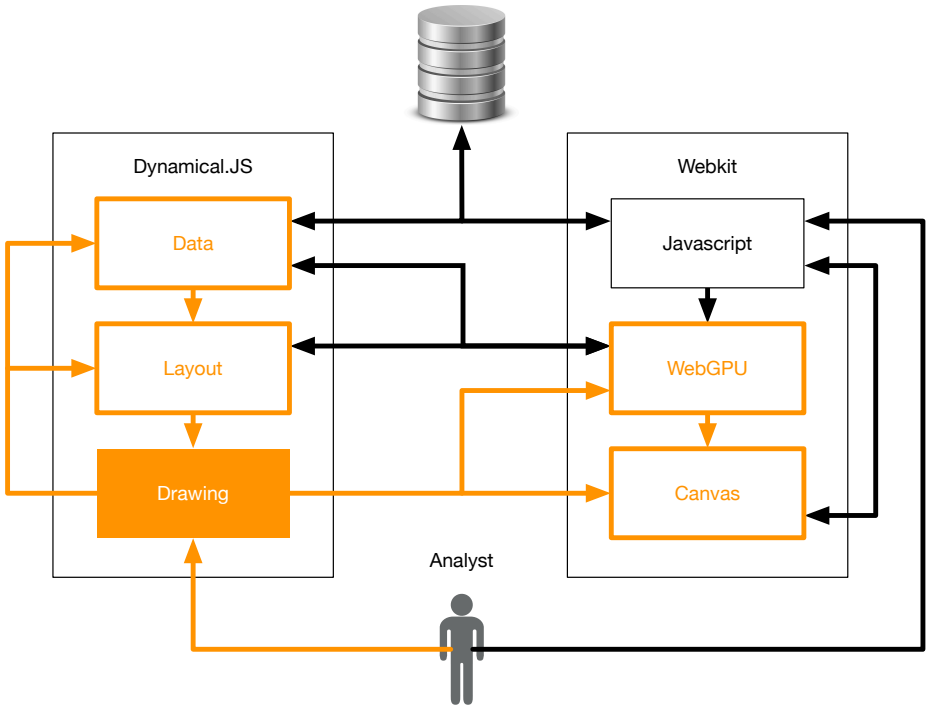


Figure 2.4. A block diagram of the Dynamical.JS Drawing Module.

validating, and tuning the other modules. This is useful during development and allows for building performance profiles the analyst can use as they switch between systems. This module supports regression/fuzz testing of each code package and child sub-packages during framework development and will form the basis of a yet-to-be-designed benchmarking suite. Not all components of the Testing/Performance Module accompany the published Dynamical.JS package.

2.4. Graph & Layout Types

The problem of visualizing large graphs consists of two significant aspects that must be considered: (1) *structure*, the inter-relations between graph elements and their attributes, and (2) *temporality*, the dynamics of the graph and its morphology (Hadlak et al., 2011).

2.4.1 Static Graph Layouts

A *static graph* $G \leftarrow (V, E, A)$, is defined as a set of nodes $v \in V$, a set of edges $e \in E \subseteq V \times V$, and a set of attributes $a \in A$ (Tutte, 1963; Eades, 1984). Attributes may be variables or functions that encode characteristics such as position, weight, labels, &c. For example, the node position attribute $\vec{\boldsymbol{p}}_v \leftarrow P_G(v)$ maps a node v onto its coordinate in the n-dimensional *layout space* $P_G(v) : v \in V \rightarrow \mathbb{R}^n$, e.g., $\vec{\boldsymbol{p}}_v = P_G(v) = \langle 1, 3, \ldots \rangle$, and is the core attribute used to compute and store the graph layouts (Simonetto et al., 2020). Graphs may be *sparse*, with few edges ($|E| \ll |V|^2$), or *dense*, where the number of edges approaches the maximum ($|E| \approx |V|^2$) (Cormen et al., 2009). Weighted graphs are graphs where each edge has an associated *weight* attribute w, typically given by a *weight function* $w : E \rightarrow \mathbb{R}$ (Cormen et al., 2009). In all cases, static layout algorithms are step-wise local refinements of suitable initial node positions subject to an arbitrary combination of aesthetic (Section 2.5), environmental, and domain-specific constraints (Section 2.12).

Many graph types are defined in the literature and classified by edge type into four broad categories:

Undirected Graphs Each pair of nodes $\langle u, v \rangle$ may have at most one linking edge ($|E_{uv}| \leq 1$); edges can be traversed in either direction (Figure 2.5a).

$$e_{uv} = E_{uv} \leftarrow \{u \leftrightarrow v\} \tag{2.1}$$

Directed Graphs Each pair of nodes $\langle u, v \rangle$ may have at most one linking edge ($|E_{uv}| \leq 1$); edges can be traversed in one direction only (Figure 2.5b). Di-

rected graphs may also allow self-loops, where an edge connects a node to itself ($u \circlearrowleft u$).

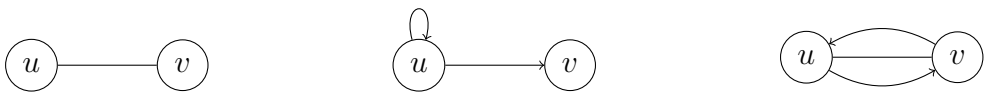
$$e_{uv} = E_{uv} \leftarrow \{u \leftarrow v \oplus u \rightarrow v\} \tag{2.2}$$

Mixed Graphs Each pair of nodes $\langle u, v \rangle$ may have at most one linking edge ($|E_{uv}| \leq 1$); edges can be either directed or undirected (Figure 2.5c).

$$e_{uv} = E_{uv} \leftarrow \{u \leftarrow v \oplus u \rightarrow v \oplus u \leftrightarrow v\} \tag{2.3}$$

Multigraphs Multiple edges can be defined between each pair of nodes ($|E_{uv}| \geq 0$). Multigraphs can also be directed, undirected, or mixed (Figure 2.5c).

$$e_{uv} \in E_{uv} \subseteq \{u \leftarrow v, u \rightarrow v, u \leftrightarrow v\} \tag{2.4}$$



(a) Undirected edge (b) Directed edge (c) Mixed, multiple edges

Figure 2.5. Graph edge classification.

The graph taxonomy further classifies graphs by connectivity, e.g., (a)cyclic graphs, trees, or other hierarchical structures. In general, these are special cases of the above graph types and are excluded from this text for the sake of brevity.

2.4.2 Dynamic (Offline) Graph Layouts

A *dynamic graph* $D \leftarrow (G, T) \leftarrow (V, E, A, T)$ is defined as a graph whose attributes are functions of time, where T is the time domain defined as an interval in $\mathbb{R}$ (Cohen et al., 1992). Attributes are functions defined in the domain $V \times T$ for nodes and $E \times T$ for edges. Dynamic graphs can be thought of as a map that links each node and edge to a sequence of functions that describe graph behavior in disjoint intervals of time T_i, with a default value returned for $t \notin T$ (Simonetto et al., 2020). For example, the dynamic node position attribute $P_D : V \times T \rightarrow \mathbb{R}^n$ is the function that describes the position $\vec{\boldsymbol{p}}_v$ for each time $t \in T$. A *timeslice* $D_t \leftarrow (G, t)$ or $G_t \leftarrow (V_G, E_G, A_G, t)$ of graph D represents its current state at time t. Static graph G can therefore be reframed as a dynamic graph where $|T| = 0$, or $G = (V, E, A, \emptyset)$.

Offline dynamic graph drawing entails a variety of techniques: (1) aggregating all morphological changes into a single super-graph, (2) linking the same node in consecutive timeslices together with inter-timeslice edges, (3) providing support for animation and small multiples, (4) and metric evaluations of the above (Simonetto et al., 2020). Offline dynamic graph layout algorithms pre-compute the graph's temporal evolution for global optimization. These algorithms have the advantage they are supplied with all relevant information for the relevant period and can optimize across it; however, these approaches require all graph permutations to be known beforehand, potentially wasting computational resources if portions of the graph are never viewed by an analyst (Simonetto et al., 2020; Hadlak et al., 2011).

As implied above, most dynamic graph drawing techniques are designed to animate a sequence of static graph layouts (Du et al., 2017). Conceptually, the dynamic graph D is embedded within an $n + 1$-dimensional spacetime cube, and a timeslice is a hyperplane through this cube. Timeslicing is integral to dynamic

graph visualization, and temporal visualization in general for several reasons: (1) dynamic algorithms that use a timeslice model can be designed by a simple extension of existing static graph layout algorithms, (2) if input data is naturally structured into timeslices, it is also natural to think of a dynamic graph as a series of snapshots that encode graph evolution, and (3) to look at the graph structure between timeslices, one may *retroject* or interpolate the graph onto an intermediate timeslice and redraw the entire dataset (Simonetto et al., 2020).

Because offline approaches only consider the totality of *past* changes, dynamic graphs must be rebuilt *from the beginning* to incorporate new data into the layout. This ensures the graph's structural and temporal continuity is maintained (Hadlak et al., 2011). To encourage layout stability, inter-timeslice edges connect nodes in the current timeslice to the same nodes in immediately antecedent and subsequent timeslices (if present), allowing for easy identification of nodes across time without special highlighting (Archambault & Purchase, 2013). If node sets remain congruent between timeslices, the inter-timeslice graph layouts required for animation can be generated simply by linearly interpolating visible nodes' position attributes. Each animation frame presumptively corresponds to a single timeslice, but that need not be the case. If an element is present in the current timeslice G_t but not in the following timeslice (G_{t+1}), it must be deleted; likewise, if it is not present in the previous timeslice (G_{t-1}), it must be added (Simonetto et al., 2020). Otherwise, either or both layouts must be expanded with the set difference of the other via *data amplification* (Smith, 1984). This renders interpolated inter-timeslice impracticable. This demonstrates an unfortunate caveat of data amplification: optimizing over the entire dynamic graph requires each timeslice to reserve enough memory to lay out every element, *even if some elements are only present in a single frame*!

A common method to overcome this issue is to resample the graph over the

time axis and aggregate or delete frames where only "minor" changes occur (Archambault & Purchase, 2013; Simonetto et al., 2020). How many timeslices should be deleted? If too few, computational resources are wasted. If too many, information is lost through aggregation across time. The Nyquist frequency states that to prevent information loss, we must sample timeslices at a rate equal to the smallest gap between events (Nyquist, 1928). Either a method of using continuous time with arbitrarily fine or nonuniform sampling rates is required (Landau, 1967), or a novel (re)materialization method must be employed (Frishman & Tal, 2008; Wang et al., 2017; Qu et al., 2017).

2.4.3 Dynamical (Online) Graph Layouts

Often, graph data manifests as a series of events where graph elements have real-valued time coordinates (Simonetto et al., 2020). To view these data using offline methods, regular timeslices must be imposed on the data and projected down onto the nearest timeslice (Simonetto et al., 2020). Such limitations of timeslice-based drawing methods serve as good motivation to pursue a model for dynamic graph drawing that is not based on timeslices. This adds another layer of complexity: as the focus changes throughout an analytical task, the layout techniques must also vary (Hadlak et al., 2011); therefore, flexibility is critical.

Simonetto et al. (2020) describe a method where a node's position attribute $\vec{\boldsymbol{p}}_v \in P_V$ are defined as piecewise functions $P_V(v,t)$ describing a trajectory through the spacetime cube represented by polylines with a given start point, endpoint, and any bends (the junctions of consecutive line segments within the polyline). Edges are then defined in the spacetime cube as a plane that connects the two trajectories. Nodes removed (i.e., the trajectory does not intersect a timeslice) cannot affect the resulting graph layout and can be discarded. Layouts only need to be generated when

new node trajectories are created or extant trajectories *bend*; intermediate layouts are again generated as simple linear interpolations between these non-uniformly spaced pseudo-timeslices by appropriately scaling the time parameter t.

An alternate approach from Beck et al. (2013) simplifies the graph via a rules-based filtering approach and aggregating salient temporal changes into *change sets* which are re-integrated into bounded spacetime cubes lazily to minimize the modifications applied to subsequent layouts without sacrificing quality or wasting memory space. A third *time-space mapping* method (Tufte, 1990; Archambault et al., 2011; Simonetto et al., 2020) presents disjoint volumes of the spacetime cube simultaneously via *multiple representation* (Section 2.5.2) or *small multiples* visualization (Section 2.5.3).

Dynamical.JS facilitates all three graph drawing techniques by separating the underlying data structures from the operations (EGOs) applied. Graphs can be visualized statically, as timeslices through the spacetime cube, or as a sequence of events, by caching and retrieving *materializations* (Section 5.8.1) of graphs and subgraphs, only generating layout frames when requested via *lazy evaluation* (Smith, 1984).

2.4.4 Context Aware Layouts

Though counter-intuitive, for the vast majority of graph visualizations described in the literature, a node's position attribute conveys no specific meaning to the analyst, relying exclusively on the *connectivity* and *proximity gestalt principles* (Ware, 2013) to prompt semantic inference. Further, the inherent complexity of encoded relationships and limited display resolution limit many graph drawings to simple node-link diagrams with trivial *on-node* and *on-edge* encodings using circles and lines as visual marks (Bertin, 2011) encoding nodes and edges respectively (Figure 1.1). *Context-aware* algorithms may apply appropriate weighting functions or

specific visual encodings to imbue layouts with semantic meaning or morph a pre-existing layout (such as a map) into a new task-specific coordinate space. Dunne & Shneiderman (2013) apply a technique, *motif simplification*, to reduce the complexity of a graph drawing and increase its information density. This method aggregates common graph structures and subgraphs into *motifs*, represented in the drawing as complex visual marks or *glyphs*. Algorithmic methods of drawing these information-rich and aesthetically pleasing node-link diagrams is an active field of current research; the creation and optimization of context-aware, semantic visualizations require human studies (Nobre et al., 2019) beyond the scope of this thesis; however, these visualization types are supported via appropriate EGOs (Section 2.7).

2.5. Aesthetics and the Mental Map

"Attractive" drawing of arbitrary static graph diagrams is an NP-hard problem due to seven general visual constraints (Section 2.12.2): (1) display symmetry, (2) edge crossing minimization, (3) edge bend minimization, (4) uniform edge lengths, (6) uniform node distribution, and (7) angular resolution (Tutte, 1963; Argyriou et al., 2012). Unfortunately, each of these constraints is "competitive:" the optimality of one prevents the optimality of the others (Tutte, 1963; Di Battista et al., 1994). Gaining insights is a key requirement of any graph visualization (Ware, 2013). The analyst's application domain may signify these requirements differently to ensure readability or performance (Cohen et al., 1992; Beck et al., 2013). Graph-theoretical characteristics may also limit the number of layout approaches and impose additional requirements on the output. For example, not every aesthetic approach can represent weighted, hierarchical, or bipartite graphs (Beck et al., 2009).

2.5.1 Preserving the Mental Map

The "mental map" refers to graph morphology represented within the analyst's mind (Misue et al., 1995). Mental map preservation facilitates readability and minimizes the analyst's cognitive load while keeping track of nodes over time (Beck et al., 2009; Archambault et al., 2011; Archambault & Purchase, 2013). Mental map preservation requires maintenance of the overall shape of the layout by reducing node movements between successive frames (Frishman & Tal, 2008). As the number of graph elements grows, the analyst's ability to interact and analyze the data decrease significantly, exacerbated as the analysis proceeds over long periods (Dunne & Shneiderman, 2013; Nobre et al., 2020). Hadlak et al. (2011) conceptualize this problem as a *visual entity budget*: the upper limit for the number of graph elements that can be displayed due to limited screen space, processing capability, and the limits of human perception and understanding. These limits can be addressed by limiting the structural or temporal components of the graph or *abstracting* and clustering these components into comprehensible meta-structures generated via algorithmic merging or splitting elements from the parent graph (Dunne & Shneiderman, 2013), filtering or querying nodes with particular properties, or explicit manual selection of salient graph elements by the analyst (Namata et al., 2007).

2.5.2 Multiple Representation

Analysis of large graphs can be made more efficient via *multiple representation*. The increase in efficiency is due to size reduction: selecting and aggregating significant subnetworks and presenting each aggregate to the analyst separately (Namata et al., 2007; Simonetto et al., 2020). This facilitates manual control and highlighting of client graph elements at the analyst's discretion: rendering multiple coordinated layouts

across separate views, coarse layouts overlaid with transient, detailed sub-layouts, or *embedding* graph layouts within a more extensive visualization (Hadlak et al., 2011). Thus, the analyst can focus on two sections of the same network by zooming into one while simultaneously filtering another or interacting with an overview layout to control background evolution. Dynamical.JS allows any combination of subnetworks or graph sequences to be represented as distinct interactive visualizations where each can be manipulated with the appropriate representation and controls (Namata et al., 2007). This allows a hierarchical subgraph to be visualized using an appropriate TreeMap. In contrast, a larger subgraph (or the entire graph) could be represented as a node-link diagram using a force-directed layout (Section 3.3) (Namata et al., 2007).

2.5.3 Small Multiples

Graph layout animations require the analyst to rely heavily on memory to gain insight from a dataset. Animated visualizations of long dynamic graph sequences are often ineffective because analysts must compare events from the beginning of the animation with events that occur at the end (Simonetto et al., 2020). *Small multiples*, a special case of multiple representation, are sets of complex visual marks which depict salient structural differences between coincident graph layouts in a single visualization (Bertin, 2011; Simonetto et al., 2020). Small multiples are employed to maximize the readability and memorability of dynamically evolving data (Ware, 2013). Visualization of dynamic graphs with small multiples can be a more effective means of displaying dynamical data because it abstracts structure *and* temporality using analyst-specified criteria (Tufte, 1990). For example, in the approach of Simonetto et al. (2020), using screen space constraints (Section 2.12), several regions in the spacetime cube containing highly dynamic node trajectories are selected. Each small multiple in

this set represents the structural state of graph elements for a particular timeslice, enabling the analyst to simultaneously compare many (or all) timeslices (Archambault et al., 2011).

2.6. Cache Consciousness

As RAM (Random Access Memory) gets cheaper, substrates with large main memories become increasingly affordable. Cache memories are small, fast, hierarchical memories that bridge main memory and compute cores, improving performance by holding recently referenced data and instructions. Caches are parameterized by capacity, block (cache line) size, and associativity. Capacity is the overall size of the cache, block size is the fundamental transfer unit between the cache and main memory, and associativity determines how many cache slots are destinations for a given memory reference. A facile assumption that memory references have uniform cost is invalid, given the widening gap between cache and main memory access speeds and associated inter-memory transfer costs. Memory latency can only be reduced when the requested data is *resident* in the cache. The programmer must keep useful data within or as close to the cache as possible.

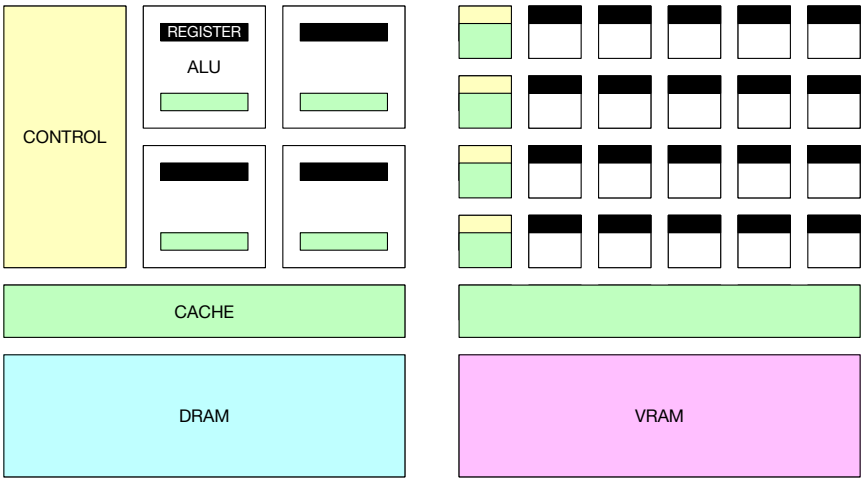
Memory references satisfied by the cache, called *hits*, proceed at processor speed; unsatisfied references, called *misses*, incur cache miss penalties and must be fetched into the corresponding cache block from lower (and slower) levels of the memory hierarchy. Consequently, a significant portion of execution time is wasted on data and instruction cache misses (Idreos et al., 2018b). To ensure requested data is accessible to the processor as quickly as possible, the programmer must use data structures that prioritize the *locality of reference*, both *temporal* (repeated references to the same memory location) and *spatial* (repeated references to different but nearby memory locations) (Ward & Halstead Jr., 1999). The converse, *cache underutilization*, must also

be prioritized. When caches remain empty or contain "useless," rarely-accessed data, cache misses incur additional write-stall penalties when pre-cached data is pushed back down the memory hierarchy (Patterson & Hennessy, 2017). If these priorities are ignored, memory latency becomes an acute performance bottleneck that prevents the application from fully exploiting the power of modern computing substrates.

2.6.1 GPU Memory

CPUs and attached GPUs have distinct memory circuitry, even if the underlying substrate is built upon a UMA (Unified Memory Architecture) (Patterson & Hennessy, 2017; N & Murali, 2019). The intricacies of the CPU memory hierarchy hide behind a *virtual memory system* that pages RAM to and from non-volatile secondary storage (Patterson & Hennessy, 2017). Like CPU memories, GPU memories are also hierarchical, divided into four or more levels (Figure 2.6): registers reside within each compute core, "local" (L1) caches are shared between groups of compute cores (known as *workgroups*), shared (L2) caches are shared among all cores, and VRAM (Video RAM) (Jeowicz et al., 2013; Patterson & Hennessy, 2017).

Unlike the CPU memory hierarchy, all GPU memory is "wired:" its hierarchy does not have a virtual memory system. L1 data caches are typically write-evict and write-no-allocate, whereas L2 caches are write-back with write-allocate (Patterson & Hennessy, 2017; Tan et al., 2020). To simplify their design, GPU caches are non-inclusive and non-exclusive, lacking the hardware coherence necessary to execute programs written in high-level languages such as C++ or Java (Singh et al., 2014; Patterson & Hennessy, 2017). Instead, the developer must write GPU-specific programs called *kernels* (Section 4.10) with substrate-specific limits in mind. L1 caches are restricted to kernel "threads" executing within the same *workgroup*, whereas L2 caches are available to all executing threads.



(a) CPU memory layout (b) GPU memory layout

Figure 2.6. A comparison of CPU and GPU memory layouts.

Like CPU programs, kernels must also use cache-conscious data structures wherever possible to ensure efficient cache usage, maximize hits in the cache, and reduce overall bandwidth requirements (Cebenoyan, 2004). Data must also be pre-sorted and streamed in L2 cache-sized blocks, aligned to the L1 cache boundaries to ensure *sequential scans* (Patterson & Hennessy, 2017; Idreos et al., 2018b; Tan et al., 2020). Due to the massive thread-level parallelism and the limited capacity, GPU caches are rarely idle but often underutilized. Experiments suggest GPU L2 cache lines spend 95.6% of their time storing data referenced only once, obviating the purpose of the cache (Li et al., 2019; Tan et al., 2020). Underutilization can be ameliorated by exploiting *texture memory*. Texture memory lookups are incredibly efficient: one-, two-, or three-dimensional texture *scans* can be used as lookup tables for complex functions or index values without requiring costly CPU memory mapping, even when executing divergent algorithms with irregular access patterns (Marshall, 2004; Bleiweiss, 2008; Qu et al., 2017; Li et al., 2019; Tan et al., 2020). In both cases, the workgroups must be sized to prioritize local (intra-workgroup) interactions over global (inter-workgroup) interactions (Patterson & Hennessy, 2017; Sorensen et al.,

2021).

Memory copies are generally the most significant performance bottleneck in graphical programs, even when the CPU interface uses specialized hardware to transfer data in bulk. This bulk transfer of data is known as *mapping*: segments of VRAM are mapped into the CPU's virtual address space, data is copied in or out, and the segments are then unmapped. The CPU exclusively "owns" mapped VRAM segments; control is not returned to the GPU until the segments are unmapped (Singh et al., 2014; Patterson & Hennessy, 2017). The inverse is *never* true: CPU memory cannot be mapped into the GPU's address space, and the GPU never gains control over CPU memory. Mapping is a relatively wasteful, *synchronous* operation: the CPU must cycle while waiting for the (very deep) GPU pipeline to idle before it can take control of requested memory segments, and the GPU must cycle when pipelines (Section 4.8) sit idle waiting for buffers to refill (Cebenoyan, 2004; Patterson & Hennessy, 2017). Data structures and access patterns must be designed to avoid unnecessary mapping of GPU memory into CPU space wherever possible. Otherwise, the GPU will stall, waiting for requested data to ascend the CPU memory hierarchy. To minimize these costs, cross-architecture data access primitives should always be *sequential scans* (Idreos et al., 2018b) to avoid stalling the GPU; random reads (*probes*) of dereferenced pointers to CPU memory must be avoided wherever possible (Patterson & Hennessy, 2017; Kester et al., 2017).

2.7. Exploratory Graph Operations (EGOs)

Most graph visualization techniques and all transitions between them can be decomposed into simple procedures called *Graph Level Operations* (Cohen et al., 1992; Lee et al., 2006; Stolper et al., 2014; Bach et al., 2017). The recomposition of these simple operations (and their underlying data structures) enables the straightforward

synthesis of existing algorithms using preexisting code components (Idreos et al., 2019, 2018a). One of the essential features of composition reflects a core framework design concept: *abstraction*. Abstraction de-emphasizes the differences between algorithms and highlights their commonalities. Common code patterns are re-expressed as *idiomatic* procedures, combined, reordered, or substituted as required. Storage details are no longer explicitly specified but implicit; detailed specifications and performance requirements are delegated to the framework by the application.

Another feature of idiomatic composition is *parametric control*: a parameter is assigned to a meaningful concept (e.g., the desired distance between nodes or limits of inter-frame movement) and adjusted to *tune* an algorithmic sequence to analyst-specified or environmental constraints (Ioannidis & Wong, 1987a). Parametric control amplifies the analyst's efforts: combining a few parameters may yield large amounts of detail without burdening the analyst with low-level details. Smith (1984) conceptualizes this phenomenon as *data(base) amplification*. Algorithmic synthesis through idiomatic composition also offers *flexibility*. An algorithm can capture the essence of the graph layout and its evolution without being constrained by the complex rules of mathematics, and allow the graph layout to express any desired amount of "accuracy:" the analyst may prioritize readability, information density, or the inclusion of purely artistic effects (Jargstorff, 2004). The analyst may also gain serendipitous insights through novel combinations of these procedures, particularly stochastic ones (Ebert et al., 2003).

Below, we augment, rename, and reorganize these graph level operations into a taxonomy of four sequentially dependent classes of EGOs (Exploratory Graph Operations): (1) GDOs (Graph Data Operations), graph-theoretical methods which manipulate a graph and its data; (2) GLOs (Graph Layout Operations), which operate on *materializations* (Section 5.8.1) to arrange graph elements within an n-dimensional

layout space; (3) GROs (Graph Rendering Operations), which render a graph layout onto a 2D (two-dimensional) canvas for display and animation; and (4) GAOs (Graph Analysis Operations), which permit the analyst to interactively explore a graph visualization and gain insight (Lee et al., 2006; Hadlak et al., 2011). Some of the defined EGOs are *atomic*, meaning they cannot be subdivided into other EGOs, while others are *composites* of other EGOs. These operations form the core functionality of the Dynamical.JS framework design and provide hints to control the underlying data structure layouts needed for execution. Each EGO and relevant control parameters (henceforth, *attributes*) defined in Dynamical.JS are *scoped* to particular entities (graphs, layouts, or views) and are inherited by dependent EGOs where applicable. Chapter 3 demonstrates graph layout algorithm synthesis methods using composable EGOs, and Chapter 5 describes how we implement these EGOs using object-oriented *design patterns* (Gamma et al., 1995).

2.8. Graph Data Operations (GDOs)

Graph Data Operations are methods which manipulate a graph ADT (Abstract Data Type) and its elements in memory. These operations are common throughout the literature (Cormen et al., 2009), this document will not contain an exhaustive list. Specifically, the taxonomy described by Lee et al. (2006) is excluded, as they refer to graph-theoretical operations familiar to the reader and outside the scope of this thesis. GDOs (Graph Data Operations) are implemented in the *Data Module*'s abstract `DGGDOInterface` within the `DGGraph` object (Listing C.7).

2.8.1 Load Graph

Return new graph G by *deserializing* a stored or transmitted representation G_S using the function f_{load}.

Table 2.1. Common symbol notation for GDOs.

Symbol	Definition
G	A graph $G \leftarrow (V, E, A)$
G_S	A serialized representation of graph G
D	A dynamic(al) graph $D \leftarrow (G, T)$
G_t	A timeslice of D at time t
V	A set of nodes (vertices)
V_G	The set of nodes in G
E	A set of edges
E_G	The set of edges in G
X	The set of elements[1] in G, $x \in X \leftarrow G \cup V_G \cup E_G$
X_G	The set of elements in G
A	A set of attributes
A_x	The set of attributes of element $x \in X$
k_x	The identifying key k for element $x \in X$
K	A set of keys, $K \leftarrow \{k_x : x \in X\}$
M_G	The materialization of G
M_{G_t}	The materialization of G at time t

$G \leftarrow \textsc{load-graph}(G_S, f_{load})$

2.8.2 Store Graph

Serialize graph G into a representation G_S appropriate for storage or transmission using the function f_{store}.

$G_S \leftarrow \textsc{store-graph}(G, f_{store})$

2.8.3 Clone Graph

Copy graph G and all of its elements into a new graph G'.

$G' \leftarrow \textsc{clone-graph}(G)$

[1] The shorthand for graph "elements," x, refers to nodes, edges, and subgraphs.

2.8.4 Subgraph

Return a subset of elements $X \subset X_G$ from graph G as a new graph instance G'. Modifications to this subgraph also apply to the "parent" graph (Mi et al., 2016). Alias of FILTER-ELEMENTS.

$G' \leftarrow \text{SUBGRAPH}(G)$

2.8.5 Merge Graphs

Merge two graphs G_A, G_B together, merging all intersecting elements ($x_{AB} \leftarrow \{x_A, x_B\} : x_A \in X_A, x_B \in X_B, k_{x_A} \equiv k_{x_B}$), and adding the symmetric difference ($G_A \triangle G_B$) (Wei et al., 2018).

$G_{AB} \leftarrow \text{MERGE-GRAPH}(G_A, G_B) \leftarrow X_{AB} \cup (G_A \triangle G_B)$

2.8.6 Retrieve Element(s)

Retrieve one or more elements $X_K \subset X_G$ from graph G as identified by key k.

$X_K \leftarrow \text{RETRIEVE-ELEMENTS}(G, K_X)$

$V_K \leftarrow \text{RETRIEVE-NODES}(G, K_V)$

$E_K \leftarrow \text{RETRIEVE-EDGES}(G, K_E)$

$x_k \leftarrow \text{RETRIEVE-ELEMENT}(G, k_x)$

$v_k \leftarrow \text{RETRIEVE-NODE}(G, k_v)$

$e_k \leftarrow \text{RETRIEVE-EDGE}(G, k_e)$

2.8.7 Filter Elements

Retrieve a collection of elements $X \subset X_G$ in graph G with filter criteria f_{filter}. Alias of RETRIEVE-ELEMENTS.

$X_{f_{filter}} \leftarrow \text{FILTER-ELEMENTS}(G, f_{filter})$

2.8.8 Add Element(s)

Add one or more elements X to a graph G. Inverse of REMOVE-ELEMENTS.

$G \leftarrow \text{ADD-ELEMENTS}(G, X) \leftarrow G_X \cup X$

$G \leftarrow \text{ADD-NODES}(G, V) \leftarrow G_V \cup V$

$G \leftarrow \text{ADD-EDGES}(G, E) \leftarrow G_E \cup E$

$G \leftarrow \text{ADD-ELEMENT}(G, x) \leftarrow G_X + x$

$G \leftarrow \text{ADD-NODE}(G, x) \leftarrow G_V + v$

$G \leftarrow \text{ADD-EDGE}(G, e) \leftarrow G_E + e$

2.8.9 Remove Element(s)

Remove one or more elements $X \subset X_G$ from graph G. Inverse of ADD-ELEMENTS..

$G \leftarrow \text{REMOVE-ELEMENTS}(G, X) \leftarrow G_X \setminus (G_X \cap X)$

$G \leftarrow \text{REMOVE-NODES}(G, V) \leftarrow G_V \setminus (G_V \cap V)$

$G \leftarrow \text{REMOVE-EDGES}(G, E) \leftarrow G_E \setminus (G_E \cap E)$

$G \leftarrow \text{REMOVE-ELEMENT}(G, x) \leftarrow G_X \setminus x$

$G \leftarrow \text{REMOVE-NODE}(G, x) \leftarrow G_V \setminus v$

$G \leftarrow \text{REMOVE-EDGE}(G, e) \leftarrow G_E \setminus e$

2.8.10 Update Element(s)

Update attribute a of one or more elements $X \subset X_G$ of graph G.

$X_G \leftarrow \text{UPDATE-ELEMENTS}(G, X_G, a)$

$V_G \leftarrow \text{UPDATE-NODES}(G, V, a)$

$E_G \leftarrow \text{UPDATE-EDGES}(G, E, a)$

$x \leftarrow \text{UPDATE-ELEMENT}(G, x, a)$

$v \leftarrow$ UPDATE-NODE(G, x, a)

$e \leftarrow$ UPDATE-EDGE(G, e, a)

2.8.11 Merge Element(s)

Combine the attributes of two or more elements $x \in X \subset X_G$ into a new element x'.

$x' \leftarrow$ MERGE-ELEMENTS(G, X)

$v' \leftarrow$ MERGE-NODES(G, V)

$e' \leftarrow$ MERGE-EDGES(G, E)

2.8.12 Materialize

Take a "snapshot" of the current state of the graph G at time t, and store it in a format that allows for cache-conscious processing of layout and rendering operations (Section 5.8.1).

$M_{G_t} \leftarrow$ MATERIALIZE(G, t)

2.8.13 Key Frame

Clone a graph G at time t. Alias of CLONE.

$G'_t \leftarrow$ KEYFRAME(G_t)

2.8.14 Coalesce

Combine all sequential Key Frames $G_{t_0}, \ldots, G_{t_n}$ into a new graph G'. Alias of MERGE-GRAPHS..

$G' \leftarrow$ COALESCE$(G_{t_0}, \ldots, G_{t_n})$

2.8.15 Calculate Graph Statistics

Calculate relevant graph-theoretical statistics of graph G (centrality, size, order, &c).

$S_G \leftarrow \textsc{calculate-statistics}(G)$

2.8.16 Find Path

Find a path P between source node $u \in V_G$ and target node $v \in V_G$ (Cormen et al., 2009).

$P \leftarrow \textsc{find-path}(G, v_s, v_t)$

2.8.17 Neighbors

Collect all nodes V_v adjacent to node v in graph G. Alias of FILTER-NODES.

$V_v \leftarrow \textsc{neighbors}(G, v)$

2.9. Graph Layout Operations (GLOs)

Graph Layout Operations position nodes V_G of graph G within an abstract n-dimensional layout space, independent of the viewing coordinate system. Generally, all GLOs operate on graph *materializations* M_{G_t}, not the graphs themselves, to enable *multiple representation* (Section 2.5.2) of the same graph data. When drawing static graphs, the time attribute is assumed to be zero ($t = 0$). These GLO methods are defined in the Layout Module's abstract `DGGLOInterface` interface and the abstract `DGLayoutEngineBase` superclass (Listing C.9).

2.9.1 Layout

Apply layout algorithm f_{layout} to the graph G_t.

Table 2.2. Common symbol notation for Layout GLOs.

Symbol	Definition
L_t	A layout of graph G_t
L_0	The layout of graph G_t at time $t = 0$
$\vec{F_\Sigma}(v)$	The sum of all forces acting on node v
$\vec{F_+}(v)$	The sum of repulsive forces acting on node v
$\vec{F_-}(v)$	The sum of attractive forces acting on node v
$\vec{F_a}(v)$	The sum of forces due to attribute a
$\vec{\boldsymbol{p}}_v$	The position attribute of node v
P_V	A set of position attributes of nodes $V \leftarrow \{\vec{\boldsymbol{p}}_v : v \in V\}$

$L_t \leftarrow \textsc{layout}(G_t, f_{layout})$

2.9.2 Initial Layout

Apply layout algorithm f_{init} to the graph G_t. Alias of LAYOUT..

$L_0 \leftarrow \textsc{init-layout}(M_{G_t}, f_{init})$

2.9.3 Merge Layouts

Generate a new layout by merging the previous layout L_{t-1} with the current layout L_t, or generate an initial layout L_0.

$L_t \leftarrow \textsc{merge-layouts}(L_{t-1}, L_t) \leftarrow (\exists L_{t-1},\ otherwise\ \textsc{init-layout}(M_{G_t}, f_{init}))$

2.9.4 Calculate Layout Statistics

Calculate relevant statistics L_{S_t} and set tunable parameters to optimize layout L_t. Layout statistics such as layout quality, temperature, and total energy are used to determine when a layout algorithm should terminate, or impose constraints on inter-frame node movements. Alias of CALCULATE-GRAPH-STATISTICS.

$L_{S_t} \leftarrow \textsc{calculate-statistics}(L_t, M_{G_t})$

2.9.5 Simplify

Simplify the graph's materialization or aggregate elements M_{G_t} via merge function f_{motif}. By breaking the large graph into smaller subgraphs, the motion of clusters of nodes may be calculated in concurrently (Qu et al., 2017; Bleiweiss, 2008). Common subgraph structures or edge groups may be consolidated into *motifs* to enhance readability or to simplify presentation (Dunne & Shneiderman, 2013). Inverse of ELABORATE.

$L' \leftarrow \text{SIMPLIFY}(L, f_{motif})$

2.9.6 Elaborate

Extract previously aggregated graph elements to facilitate multiple representation (Section 2.5.2), transformation, or animation. Inverse of SIMPLIFY.

$L \leftarrow \text{ELABORATE}(L', {f_{motif}}^{-1})$

2.9.7 Partition

Slice graph materialization M_{G_t} into partitions $m_0, \ldots, m_n$ using partition function $f_{scatter}$ to facilitate parallel computation (Lu & Si, 2020; Kreylos & Hamann, 2001; Mi et al., 2016). Inverse of MERGE-PARTITIONS.

$m_0, \ldots, m_n \leftarrow \text{PARTITION}(M_{G_t}, f_{scatter})$

2.9.8 Merge Partitions

Merge materialization partitions $m_0, \ldots, m_n$ together using function f_{gather} for rendering. Inverse of PARTITION.

$M_{G_t} \leftarrow \text{MERGE-PARTITIONS}(m_0, \ldots, m_n, f_{gather})$

2.9.9 Calculate Forces

Calculate cumulative force(s) on node v by nodes $V_G \setminus v$ in graph materialization M_{G_t} (Godiyal et al., 2009; Mi et al., 2016). Alias of CALCULATE-OFFSETS.

$\vec{F_\Sigma}(V) \leftarrow$ CALCULATE-FORCES$(M_{G_t}, v) \leftarrow \Sigma_{i=0}^{|V_G|} \vec{F_a}(v)_i$

2.9.10 Calculate Repulsive Forces

Calculate repulsive force(s) on node v by nodes $V_G \setminus v$ in graph materialization M_{G_t}. Alias of CALCULATE-FORCES.

$\vec{F_+}(v) \leftarrow$ REPULSE(M_{G_t}, v)

2.9.11 Calculate Attractive Forces

Calculate attractive force(s) on node v by nodes $V_G \setminus v$ in graph materialization M_{G_t}. Alias of CALCULATE-FORCES.

$\vec{F_-}(v) \leftarrow$ ATTRACT(M_{G_t}, v)

2.9.12 Calculate Offsets

Calculate position offsets ΔP_V for nodes V_G in graph materialization M_{G_t} based on an arbitrary combination of attributes (weight, mark size, force, &c.). Alias of CALCULATE-FORCES.

$\Delta P_V \leftarrow$ CALCULATE-OFFSETS(M_{G_t}, v)

2.9.13 Centroid

Determine the layout space coordinate $\langle x, y, \ldots \rangle \in S$ of the geometrical center of nodes $V \subset V_G$ in graph materialization M_{G_t}.

$\vec{\boldsymbol{p}}_\odot =$ CENTROID(M_{G_t}, V)

2.9.14 Place Node(s)

Modify the layout space position attribute of nodes $\vec{\boldsymbol{p}}_v \in P_V$ in graph materialization M_{G_t}.

$P_V \leftarrow \text{PLACE}(M_{G_t}, V_G) \leftarrow P_V + \text{CALCULATE-OFFSETS}(M_{G_t}, v)$

2.9.15 Quality

Determine the quality metric σ for graph layout L_{G_t} using function $f_{fitness}$. Alias of CALCULATE-STATISTICS.

$\sigma \leftarrow \text{QUALITY}(L_t, f_{fitness}, \omega)$

2.9.16 Finesse

Refine the layout L_t to meet requirements defined by the analytical domain or output medium.

$L \leftarrow \text{FINESSE}(M_{G_t})$

2.10. Graph Rendering Operations (GROs)

Table 2.3. Common symbol notation for GROs (Graph Rendering Operations).

Symbol	Definition
S	A viewing surface.
S_{view}	The visible area of viewing surface S
R_t	A rendered graph layout L_t

GROs render a graph materialization M_{G_t} into a specified viewing surface S. Viewing surfaces may be the view canvas, a texture, or other memory buffer. GROs are implemented in the *Drawing Module*'s abstract `DGGROInterface` interface.

2.10.1 Update Dimensions

Update the dimension attributes (width, height, depth) of the viewing surface S.

$S \leftarrow \textsc{update-dimensions}(S)$

2.10.2 Apply Transform

Apply transform function $f_{transform}$ to surface S_{view}. Transforms include, but are not limited to, linear-algebraic operations known as *affine-transformations*, which map both geometric position and color attributes within/between coordinate systems or color spaces respectively (Bailey & Cunningham, 2009). These affine-transforms include: rotation, scaling, translation, shearing, perspective, reflection, and orthographic projection (Foley et al., 1996; Akenine-Möller et al., 2008).

$S \leftarrow \textsc{transform}(S, f_{transform})$

2.10.3 Render Layout

Render graph layout L_t to surface S, applying the surface's projection transform attribute to map layout positions PV into the view coordinate space.

$R_t \leftarrow \textsc{render}(L_t, S)$

2.10.4 Render Region

Render only the parts of a graph layout L_t bounded by a viewing rectangle S_{view}. Alias of RENDER-LAYOUT.

$R_{view,\ t} \leftarrow \textsc{render-region}(L_t, S_{view})$

2.10.5 Overlay

Render layout L'_t of subgraph G' over the previously rendered graph R_t.

$R'_t \leftarrow \textsc{overlay}(R_t, L'_t)$

2.10.6 Show Element(s)

Add one or more graph elements X_G to the final output rendering R_t. Alias of UPDATE-ELEMENTS. Inverse of HIDE-ELEMENTS.

$R_t \leftarrow \textsc{show-elements}(R_t, X_G)$

2.10.7 Hide Element(s)

Remove one or more graph elements X_G to the final output rendering R_t. Alias of UPDATE-ELEMENTS. Inverse of SHOW-ELEMENTS.

$R_t \leftarrow \textsc{hide-elements}(R_t, X_G)$

2.10.8 Apply Visual Mark to Element(s)

Update one or more graph elements X_G with a specific visual mark m. Alias of UPDATE-ELEMENTS.

$X_G \leftarrow \textsc{apply-mark}(X_G, m)$

2.10.9 Apply Visual Channel to Element(s)

Update one or more graph elements X_G with a specific visual channel c. Alias of UPDATE-ELEMENTS.

$X_G \leftarrow \textsc{apply-channel}(X_G, c)$

2.10.10 Interpolate

Interpolate between two (possibly non-adjacent) graph layouts $L_0, \ldots, L_1$, at intermediate time t. Simple linear interpolation is a familiar technique (Foley et al., 1996): given a general data value f with values f_a and f_b at two endpoints a and b of a line segment, interpolation with parameter t is shown in Equation 2.5. If the data are in homogeneous coordinates $f \leftarrow \langle r, s, t, q \rangle : q \neq 1$, the coordinates are converted into standard form by dividing each f by q and interpolate f/q as shown in Equation 2.6. Simple linear interpolation is only effective for interpolating simple values such as position or color in *layout space* S. However, to interpolate complex values such as texture coordinates, we must defer this interpolation until we reach clip space (Section 4.8.1). Instead of linear interpolation, we must perform *perspective interpolation*, which reduces to linear interpolation if clip and layout space are congruent. The perspective interpolation function is shown in Equation 2.7, where $\alpha = 1$ unless interpolating texture coordinates, and w_a and w_b are the fourth coordinates of the endpoints a and b in homogeneous clip space as in Equation 2.7.

$L_t \leftarrow \textsc{interpolate}(L_0, L_1, t)$

$$(1-t)f_a + tf_b \tag{2.5}$$

$$(1-t)f_a/q_a + tf_b/q_b \tag{2.6}$$

$$\frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)\alpha_a/w_a + t\alpha_b/w_b} \tag{2.7}$$

2.11. Graph Analysis Operations (GAOs)

GAOs (Graph Analysis Operations) manipulate the clipped view space S_{view} and current render state R_t of the current graph layout L_t, by reshaping the canvas, manually adding and removing graph elements from a "focus set" $X \subset X_G$, and initiating animations or other graph transformations. Like the GDOs listed above (Section 2.8), this list concerns itself only with the general *computational* tasks required to facilitate graph exploration by the analyst. Analytical tasks, such as those described Ahn et al. (2014) are excluded, as they are apropos to the application-domain not the computational frameworks which underlie them. GAOs are implemented in the *Drawing Module*'s abstract `DGGAOInterface` interface.

2.11.1 Find Element(s) by Position

Return the element x rendered at a specified view coordinate $\vec{\boldsymbol{p}}_{x,y} \in S_{view}$. Alias of FILTER-ELEMENTS.

$x \leftarrow \text{FIND-ELEMENT}(R_t, \vec{\boldsymbol{p}}_{x,y})$

2.11.2 Find Bounded Element(s)

Return the set of elements $X \subset X_G$ bounded by a rectangle defined by the view space S_{view} positions $\vec{\boldsymbol{p}}_0$ (bottom left) and $\vec{\boldsymbol{p}}_1$ (top right). Alias of FILTER-ELEMENTS.

$X \leftarrow \text{FIND-BOUNDED-ELEMENTS}(R_t, (\vec{\boldsymbol{p}}_0, \vec{\boldsymbol{p}}_1))$

2.11.3 Select Element(s)

Add selected graph elements $X \subset X_G$ to a collection X_{select}. Alias of ADD-ELEMENTS. Inverse of DESELECT.

$X_{select} \leftarrow \text{SELECT}(X_{select}, X_G)$

2.11.4 Deselect Element(s)

Remove selected elements $X \subset X_G$ from a collection X_{select}. Alias of REMOVE-ELEMENTS. Inverse of SELECT.

$X_{select} \leftarrow \text{DESELECT}(X_{select}, X_G)$

2.11.5 Filter Element(s)

Create a collection of elements $X \subset X_G$ which meet some analyst-defined criteria f_{filter}. Alias of FILTER-ELEMENTS.

$X \leftarrow \text{FILTER}(M_{G_t}, f_{filter})$

2.11.6 Highlight Element(s) of Interest

Render elements of interest $X \subset X_G$ with distinguishing visual marks and visual channels to aid in visual analysis. Alias of APPLY-MARK and APPLY-CHANNEL.

$R_t \leftarrow \text{HIGHLIGHT}(R_t, X)$

2.11.7 Export Image

Serialize the rendered image R_t using a specified graphics format function f_{format}, such as SVG (Scalable Vector Graphics), PDF (Portable Document Format) or JPEG (Joint Photographic Experts Group).

$R_S \leftarrow \text{EXPORT}(R_t, f_{format})$

2.11.8 Step Backward and Forward through GRO History

Retrieve and display image R_t and/or (re)render a specific materialization M_{G_t} at time $t \pm \Delta$. Combined with INTERPOLATE, this method forms the basis of

animation in Dynamical.JS.

$$R \leftarrow \textsc{step}(R_t, \Delta t)$$

2.12. Tunability

Several algorithms define tunable constraint parameters, some favoring mental map preservation (Du et al., 2017; Wei et al., 2018) or analyst-specified aesthetic qualities (Sheng et al., 2019). Others control the number of successive timeslices presented or change the animation framerate to reveal or obscure temporal features (Simonetto et al., 2020). Each parameter is expressed as a constant or functional attribute $a \in X_A$, enabling the developer and the analyst to customize applications to optimize domain-specific criteria (Beck et al., 2013).

2.12.1 Environmental Constraints

Substrate-defined environmental constraint parameters are attributes that form the basis of optimization procedures. Generally, these constraints are not mutable by the analyst.

Table 2.4. Environmental constraint parameters.

Constraint	Definition
Max. Threads	The maximum number of parallel thread contexts available.
Max. GPU Workgroup Size	The maximum of threads per GPU workgroup.
Max. Buffer Size	The maximum CPU buffer size. Defaults to $4GiB$.
Max. GPU Buffer Size	The maximum GPU buffer size.
Memory Page Size	The substrate-dependent memory page size.
L1 & L2 Cache Size	The width of the L1 & L2 caches.
Output Resolution	The pixel dimensions of view space S_{view}.

2.12.2 Layout Constraints

Table 2.5. Layout constraint parameters.

Constraint	Definition
Max. Intermediate Offset	The maximum node displacement per iteration.
Max. Temporal Offset	The maximum node displacement per timeslice.
Movement Acceleration	A cost function that rewards consistent inter-timeslice movement and penalizes opposing movements.
Min. Sampling Frequency	The minimum sampling interval between timeslices.
Avg. Sampling Frequency	The mean time sampling frequency across an optimized spacetime cube.
Desired Edge Length	The ideal edge length between any two nodes in *view space* S_{view}. Dependent on the size of the graph $\|G\|$ and the resolution of the output medium.
Max. Edge Length	In *layout space* S, the distance between any pair of connected nodes or between any pair of *disconnected* nodes where their respective positions have no effect upon the movement of the other.

Adding or removing a small number of critical elements may have a dramatic impact on subsequent layouts, thus breaking the analyst's mental map (Du et al., 2017). Layout constraints limit the distance between nodes and their movement throughout *layout space* S. Node movement constraints apply to static graph layout algorithms to prevent undesirable translations within layout space, and to ensure smooth animation of both event-driven dynamic graph layout algorithms and offline dynamic graph layout algorithms over a range of the spacetime cube (Simonetto et al., 2020). Dynamic(al) constraint methods vary widely by algorithm, including "swarm" methods (Du et al., 2017), geometric methods (Xu et al., 2018), node aging

methods, or adaptive, multilevel approaches (Du et al., 2017); therefore the list shown in Table 2.5 is inclusive but not exhaustive.

2.12.3 Quality Metrics

In addition to standard graph-theoretical metrics (connectedness, structural integrity, &c.), there are several other quality (Section 2.9.15) metrics that determine the "fitness" of a particular graph layout by comparing the output of a fitness function $f_{fitness}$ with an associated threshold attribute $\Omega = f_{fitness}(L_{G_t})$.

Table 2.6. Layout quality metrics.

Metric	Definition
Temperature	The average inter- and intra-frame movement of all nodes within a graph layout (Fruchterman & Reingold, 1991).
Stress	A measure of how well the average node position reflects the shortest path (Euclidean) distance within the graph layout. Stress values are averaged across the entire graph for static layouts. For dynamic and dynamical layouts, the stress is computed per timeslice and then averaged across the spacetime cube. Stress may be calculated over an arbitrary subset of graph elements, such as a neighborhood, cluster, or the entire graph (Simonetto et al., 2020).
Crowding	The total number of times nodes pass very close to each other (desired edge length $\geq \| \vec{\boldsymbol{p}}_v - \vec{\boldsymbol{p}}_u \|$) in an animation of the dynamic graph. Crowding adversely affects the identification of nodes and negatively influences object tracing. Multiple overlaps of any two nodes during a dynamic(al) graph sequence are counted as a single crowding event (Simonetto et al., 2020).

Chapter III.

Layout Algorithm Decomposition

Research on the production of aesthetically pleasing graph drawings appears throughout the broad spectrum of computer science, and the literature documents thousands of techniques (Di Battista et al., 1994; Nobre et al., 2019). Graph layout algorithms read as input a combinatorial description of a graph G and produce as output a layout L_G of G according to a given aesthetic standard (Section 2.5). For example, *orthogonal* layouts are restricted to grids (Valiant, 1981), symmetrical layouts are restricted to concentric circles (Lin & Yen, 2005; Xu et al., 2018), and hierarchical layouts are restricted to parallel lines (Sugiyama et al., 1981). Graph drawing algorithms take these layouts as input and produce drawings (or *renderings*) R according to relevant domain-specific standards (Tutte, 1963). This section will use these terms interchangeably, as is common in the literature.

Nodes may be drawn using arbitrary visual marks; edges may likewise be drawn using straight, polygonal, or curved line visual marks, with line style, thickness, length, or curvature as visual channels that encode relational information (Kamada & Kawai, 1989). Choosing appropriate marks and channels relevant to a particular dataset or application domain remains an open problem (Beck et al., 2013) we do not address here. Instead, this section surveys several standard algorithms applied to node-link diagrams, analyzes their shared features, and describes their decomposition into the GLOs (Graph Layout Operations) defined in Section 2.7. This discussion is limited to graph layouts within a 2D (two-dimensional) layout space but is generally applicable to higher-dimensional layout spaces.

3.1. Algorithm Stages

Most graph layout algorithms are expressed as *iterative-convergent* processes (Kamada & Kawai, 1989; Wang et al., 2017). *Iterative* operations repeatedly run a sequence of steps; *convergent* iterations approach the "correct" answer and terminate when that answer is reached. Mathematically, an iterative-convergent process is a series of operations $f_0, f_1, \ldots, f_{n-1}$ that operate on graph data G, where $G_i = f(G_{i-1})$, until at iteration i, the terminal condition function $f_{stop}(G_i, i)$ returns `true`. However intuitive, these processes are not standalone; the number of iterations is only one of many factors affecting convergence. Aesthetic, heuristic, and computational constraints (Section 2.12) require setup before and refinement after the iteration process.

Following the main themes of this document, *abstraction* and *composition*, we derive a staged layout strategy by analyzing well-documented algorithms from the literature, abstracting common procedures into GLOs (Graph Layout Operations), and composing these operations into a general algorithmic structure. We expand on the *multi-scale* strategy of Hadany & Harel (2001), which should be familiar to electrical engineers as a PID (Proportional Integral Derivative) loop: (re)position nodes to yield a locally organized configuration, perform coarse-scale relocations, then perform fine-scale relocations that correct local disorders introduced by the previous step. We add extra setup and tear-down stages to increase the strategy's flexibility. Algorithm 3.1 presents the staged layout structure, and Listing 5.8 shows the default implementation. Each algorithm stage corresponds to one or more of the GLOs identified in Section 2.9:

1. Initial placement (INIT-LAYOUT, Section 2.9.2)
2. Statistics calculation (CALCULATE-STATISTICS, Section 2.9.4)
3. Partitioning (PARTITION, Section 2.9.7; SIMPLIFY, Section 2.9.5)

4. Iterative placement (PLACE, Section 2.9.14)
5. Merge (MERGE-PARTITIONS, Section 2.9.8; ELABORATE, Section 2.9.6)
6. Finesse (FINESSE, Section 2.9.16).

While this sequence generally applies to most graph layouts, specific implementations require additional refinements. Several layout algorithms extend, replace, re-order, or skip stages altogether. Most use the output of a "basic" layout algorithm (Section 3.2) for initial node placement. Others refine prior layouts using additional aesthetic criteria (Section 3.4.2) or generate a long sequence of "final" layouts as timeslices in the spacetime cube (Section 2.4.2).

Algorithm 3.1 The LAYOUT algorithm

Input:
$G_t \leftarrow (V, E, A, t)$
$f_{init} \leftarrow$ the initial layout function
$f_{stop} \leftarrow$ the terminal condition function

Output:
$L_{G_t} \leftarrow$ a nice layout of G_t

procedure LAYOUT(G,f_{init})
    $M_{G_t} \leftarrow$ MATERIALIZE(G_t) ▷ *Stage 0*
    $L_{G_t} \leftarrow$ INIT-LAYOUT(M_{G_t}, f_{init}) ▷ *Stage 1*
    EMIT(`layout_begin`)
    $L_{S_t} \leftarrow$ CALCULATE-STATISTICS(L_{G_t}, M_{G_t}) ▷ *Stage 2*
    $M_{G_t} \leftarrow$ PARTITION($M_{G_t}, f_{scatter}$) ▷ *Stage 3*
    repeat ▷ *Stage 4*
        for all $m \in M_{G_t}$ **do**
            $\Delta P_V \leftarrow$ CALCULATE-OFFSETS(m) ▷ *Stage 4a*
            $P_V \leftarrow$ PLACE(m) ▷ *Stage 4b*
        EMIT(`layout_step`)
    until $f_{stop} = true$
    $M_{G_t} \leftarrow$ MERGE-PARTITIONS($m \in M_{G_t}, f_{gather}$) ▷ *Stage 5*
    $L_{G_t} \leftarrow$ FINESSE(L_t) ▷ *Stage 6*
    EMIT(`layout_end`)

    return L_{G_t}

3.1.1 Initial Placement

The rough initial placement of nodes in the high-dimensional layout space reduces the number of iterations required to minimize the energy function and move nodes to their final positions. Experiments suggest that initial node placement does not significantly influence the resultant graph layouts for *static* graphs, except for special cases, such as when all nodes lie on a single line (Kamada & Kawai, 1989; Kobourov, 2013). Therefore, an initial placement algorithm need only exclude these special cases from consideration, and any other input, such as randomly-generated node positions or precomputed positions from prior graph layouts (to preserve the mental map), is sufficient (Di Battista et al., 1994). The algorithms defined in Section 3.3 use variants of randomized node placement; however, complex and dynamic versions of these algorithms routinely take the output of simpler algorithms as input for fine-tuning (Di Battista et al., 1994).

3.1.2 Statistics Calculation

This stage calculates relevant statistics such as the number of layout iterations, the width of partitions, or the maximum allowed neighborhood size using tunable aesthetic and performance constraints. Statistics derived from the underlying graph-theoretical structure (connectedness, structure, or degree), neighborhood count, or relevant path lengths are used to calculate a "metric of similarity" and may limit the number of possible drawings to those "easiest" to achieve (Harish & Narayanan, 2007; Xu et al., 2018). This stage allows for optimizing the partition stage and intermediate rendering of the placement stage, as described below.

3.1.3 Partition

Graphs are considered *very large* if the number of constituent elements is in the order of $100,000$ ($|G| = (|V| \times |V_A| + |E| \times |E_A| \times t) \geq 10^5$ where $|V_A|$ and $|E_A|$ are the number of node and edge attributes respectively). Both processing and analysis of large graphs are inefficient; serial computation of positional fitness values is computationally expensive (Qu et al., 2017) and screen resolution limits the number of discriminable elements, obscuring the inherent complexity of encoded relationships between salient elements (Dunne & Shneiderman, 2013). To effectively process or present very large graphs, this stage decreases the computational task by reducing the graph size, the number of timeslices, or both (Hadlak et al., 2011).

The monolithic *partitioning* (PARTITION, Section 2.9.7) task reduction technique groups graph elements into logical blocks or subgraphs for parallel processing. Multi-level *coarsification* techniques (SIMPLIFY, Section 2.9.5) reduce the task via selective filtering of elements or interpreting graphs as a hierarchy of progressively simpler structures laid out in reverse order of complexity (Walshaw, 2003; Kobourov, 2012; Tamassia & Rosen, 2013). *Motif simplification* aggregates common subgraph structures into *motifs* to enhance readability and simplify presentation (Dunne & Shneiderman, 2013), while *clustering* and *neighborhood beautification* techniques focus temporal development of graph morphology and optimizing information density, respectively (Galán & Mengshoel, 2018). The graph's combinatorial structure is significantly simplified, but essential visualization features are preserved (Hadany & Harel, 2001).

3.1.4 Placement Loop

This is the iterative-convergent stage of the layout sequence, which refines node positions until they approximate "ideal" positions compatible with the aesthetic criteria (Hadany & Harel, 2001). During each iteration, relative offset vectors are calculated for each node (CALCULATE-OFFSETS, Section 2.9.12). The force-directed methods calculate inter-node attractive and repulsive forces, calculate weights for weighted graphs, and additional offsets due to custom attributes. Computed offsets and node positions are summed element-wise to "correct" node placement within the layout space (PLACE, Section 2.9.14). This stage iterates until reaching some *terminal condition* f_{stop}. The iteration may terminate after a fixed number of iterations M, when the aggregate node movement converges to some threshold, or after some other quality metric is achieved (QUALITY, Section 2.9.15).

3.1.5 Merge Partitions

This stage is the inverse of the partition stage (Section 3.1.3). Merging recombines the position values in the materialization with the underlying graph data structure (MERGE-PARTITIONS, Section 2.9.8), extracts the results of previous coarsification steps back into the parent graph (ELABORATE, Section 2.9.6), modifies element attributes, or scales relevant weighting factors.

3.1.6 Finesse

Fine-scale revision of node positions occurs in the final stage (ELABORATE, Section 2.9.6) to facilitate the addition or removal of individual nodes or modification of edges or their attributes and prevent visual artifacts such as pixel anti-aliasing. This stage finally prepares the graph layout for *view space* interpolation to ensure

smooth animation, an essential quality for maintaining the analyst's mental map. This stage may also leave laid-out positions intact yet declutter the graph by hiding or revealing any number of elements (e.g., setting an element's *visibility attribute*) as required for subsequent analytical tasks.

3.2. Basic Algorithms

Algorithm 3.2 The BASIC layout algorithm

```
Input:
    G_t ← (V, E, A, t)
    f_init ← ∅

function f_stop
    return true

function PLACE(M_{G_t})
    ▷ Clear the position and offset vectors to zero.
    for all v ∈ V_G ⊂ M_{G_t} do
        p⃗_v ← ⟨0, 0⟩
        δ⃗_v ← ⟨0, 0⟩

procedure BASIC(G_t,∅)
    M_{G_t} ← MATERIALIZE(G_t)
    EMIT(layout_begin)
    PLACE(M_{G_t})
    EMIT(layout_end)

    return L_{G_t}
```

Basic graph layout algorithms comprise only a single iteration of the placement stage (Section 3.1.4). All other stages and sub-stages are redefined to `no-op`s (Null-Operations) with no computational cost or effect, reducing the algorithm as shown in Algorithm 3.2. With few exceptions, these algorithms require only a single linear scan over the materialization data to set the position vector $\vec{\boldsymbol{p}}_v$ and simultaneously reset the offset vector $\vec{\delta}_v$, therefore having a computational complexity of $\mathrm{O}(|V_G|)$.

All future algorithms described below build on this structure, gradually (re)defining new stages as required.

3.2.1 Random Placement

Algorithm 3.3 The RANDOM layout algorithm

```
Input:
    G_t ← (V, E, A, t)

▷ Randomize node positions
function RANDOMIZE(V)
    for all v ∈ V do
        p⃗_v ← ⟨RANDOM(), RANDOM()⟩
        δ⃗_v ← ⟨0, 0⟩

▷ Override BASIC
function PLACE(M_{G_t})
    RANDOMIZE(V_G ⊂ M_{G_t})

procedure LAYOUT-RANDOM(G,∅)
    return BASIC(G,∅)
```

Many graph layout algorithms, especially force-directed layouts (Section 3.3), begin with a random placement stage to generate the initial layout. This distributes nodes evenly throughout the layout space and forces the nodes into a maximal energy state (Algorithm 3.3).

3.2.2 Geometric

A "geometric" layout places nodes around a strictly convex polygon P with $|V|$ vertices. In practice, this reduces to evenly spacing nodes around a circle inscribed within the bounds of layout space S (Algorithm 3.4).

Algorithm 3.4 The POLYGON layout algorithm

Input:
$G_t \leftarrow (V, E, A, t)$
$r \leftarrow \min(S.width,\ S.height)/2$

▷ *Place nodes $v \in V$ around the circle inscribed in S*
function CIRCUMSCRIBE(V)
  for all $v_i \in V$ **do**
    $\theta \leftarrow i \times \frac{2\pi}{|V|}$
    $\vec{\boldsymbol{p}}_v \leftarrow \langle \cos\theta, \sin\theta \rangle \times r$
    $\vec{\delta}_v \leftarrow \langle 0, 0 \rangle$

▷ *Override* BASIC
function PLACE(M_{G_t})
  CIRCUMSCRIBE($V_G \subset M_{G_t}$)

procedure LAYOUT-POLYGON(G,$\emptyset$)
  return BASIC(G,$\emptyset$)

Algorithm 3.5 The FIXED-FREE layout algorithm

Input:
$G_t \leftarrow (V, E, A, t)$

▷ *Filter nodes by their fixed attribute*
function $f_{filter}(v)$
  return $v.fixed =$ `true`

▷ *Override* BASIC
function PLACE(M_{G_t})
  $V_0 \leftarrow$ FILTER-ELEMENTS($V_G \subset M_{G_t}, f_{filter}$)
  $V_1 \leftarrow V_G \setminus V_0$
  CIRCUMSCRIBE(V_0)
  RANDOMIZE(V_1)

procedure FIXED-FREE(G,$\emptyset$)
  return BASIC(G,$\emptyset$)

3.2.3 Fixed & Free

The "Fixed/Free" layout mixes the random and geometric methods above. Unlike the previous methods, this method introduces the filter function f_{filter}, which divides the nodes into those which are fixed (position attributes are immutable) and free using the FILTER-ELEMENTS GDO (Graph Data Operation) (Section 2.8.7). Nodes are partitioned $V = V_0 \cup V_1$ of V into a set V_0 of at least three *fixed* nodes and a set of V_1 *free* nodes. The fixed nodes are placed at the vertices of strictly convex polygon P and remain fixed throughout the layout algorithm. Free nodes are placed randomly throughout the layout space or allowed to remain zero (Algorithm 3.5).

3.3. Force-Directed Layouts

This thesis focuses on physical model-based heuristics, the most frequently documented in the literature. These heuristics are natural physical metaphors balancing the listed general requirements for aesthetic drawings and their analytical cost calculations (Nobre et al., 2019, 2020) and are more easily decomposed for optimization of a specific quality. Some of the most flexible physical models for calculating undirected graph layouts belong to a class known as force-directed layout algorithms. Graphs drawn with these algorithms tend to be aesthetically pleasing, exhibit symmetries, and produce crossing-free layouts for *planar* graphs (Tutte, 1963; Tamassia & Rosen, 2013).

Spring Embedders are “mechanical” systems that model edges as springs with forces that behave similarly to Hooke’s law (Eades, 1984; Fruchterman & Reingold, 1991). Strong repulsive forces exist between all nodes, but springs attract the nodes together if they are too far apart and repel them if they are too close (Di Battista et al., 1994). Spring embedders calculate the graph layout using only information

contained within the structure of the graph itself rather than relying on domain-specific knowledge (Tamassia & Rosen, 2013).

Electrical Field Potential algorithms are "electrical" systems where all nodes have identical "polarity," thus repelling each other and establishing a minimum inter-node distance. Edges are modeled as virtual "nodes" of the opposite polarity, which attract the connected nodes to the center of each edge. A variant of this algorithm refines the final graph by adding a step where the edges repel *each other* and attract the nodes (Tamassia & Rosen, 2013).

Gravitational Field Potential algorithms are "gravitational" systems where massless nodes move randomly throughout the layout space. However, they are connected by multiple "weighted" edges, which attract the nodes, decelerating them to their final positions. The initial random motion prevents overcrowding of nodes while summing the "heavy" weighted edges draws strongly-connected nodes close together (Tutte, 1963).

Each of these models has the same goal and results: the minimization of the total "energy" contained within the system of nodes and edges in a graph as represented by an *energy function* (Tutte, 1963). These energy minimization methods produce graphs with balanced aspect ratios, high degrees of display symmetry, few edge bends, and uniform node distributions. This energy function is a natural target for understanding and optimization, but it is quite challenging to implement in practice. Once done, however, extending these algorithms to high-dimensional spaces is trivial (Nobre et al., 2019).

Traditional force-directed approaches are generally poor for graphs with more than a few hundred nodes (Tamassia & Rosen, 2013). As these models have many local minima, they lack scalability and cannot consistently produce good layouts for large graphs (Tamassia & Rosen, 2013). Additionally, except for Tutte (1963)'s

Algorithm 3.6 The FORCE-DIRECTED algorithm

Input:
$G_t \leftarrow (V, E, A, t)$
$f_{init} \leftarrow$ the initial layout function

Output:
$L_{G_t} \leftarrow$ a nice layout of G_t

▷ *override* CALCULATE-OFFSETS
function CALCULATE-OFFSETS(M_{G_t})
  return CALCULATE-FORCES(M_{G_t})

▷ *define* CALCULATE-FORCES
function CALCULATE-FORCES(M_{G_t})
  for all $v \in V_M$ **do**
    ▷ *calculate repulsive forces*
    $\vec{\delta}_v \leftarrow$ REPULSE(M_{G_t}, v)
    ▷ *calculate attractive forces*
    $\vec{\delta}_v \leftarrow \vec{\delta}_v +$ ATTRACT(M_{G_t}, v)
    ▷ *calculate forces due to attributes of* v
    $\vec{\delta}_v \leftarrow \vec{\delta}_v + \Sigma_{a \in A_v} \vec{F}_a(v)$
  return ΔP_V

procedure FORCE-DIRECTED(G_t, f_{init})
  return LAYOUT(G_t, f_{init})

barycentric method (Section 3.3.1) and the edge-repellent variant of the electrical field potential algorithm (Section 3.4.2), force-directed layouts can lead to unpredictable edge crossings and collapse (Lin & Yen, 2005).

Modern versions of these algorithms alleviate these problems through *partitioning*, *coarsification*, or *filtration* (Section 3.1.3). Newer variants use Laplacian eigenvectors to simultaneously compute the "weighted" centroids of all nodes utilizing a method known as *power iteration*. This method constructs a high-dimensional graph representation, which is then, with minimal effort, projected into a lower-dimensional space (Koren, 2005). Alternatively, the algorithms described by Kamada & Kawai (1989) compute forces between nodes based on their graph-theoretic dis-

Table 3.1. Common symbol notation for force-directed algorithms.

Symbol	Definition
$\vec{\boldsymbol{p}}_v$	The position vector of node $v \in V$.
$\vec{\delta}_{uv}$	The difference vector between two nodes u and v ($\vec{\boldsymbol{p}}_v - \vec{\boldsymbol{p}}_u$).
$\hat{\delta}_{uv}$	The unit vector pointing from $\vec{\boldsymbol{p}}_u$ to $\vec{\boldsymbol{p}}_v$ ($\vec{\delta}_{uv}/\lvert\vec{\delta}_{uv}\rvert$).

tances proportional to the shortest path lengths between them.

Exploratory graph visualizations mitigate these problems by simply focusing on regions of interest and zooming and translating the viewing surface appropriately; combined, all extend the usefulness of force-directed methods to graphs of thousands or even hundreds of thousands of nodes (Tamassia & Rosen, 2013).

3.3.1 The Barycentric Method

Tutte (1963)'s barycentric method is one of the first force-directed graph layout algorithms for obtaining straight-line, crossing-free layouts for any given 3-connected planar graph. Moreover, this method guarantees that all faces contained in the drawing are convex.

The basic idea of the barycentric algorithm is as follows: affix a single face of the planar graph in the layout space and place the remaining nodes by solving a system of linear equations, where the final position of each node is a convex combination of the positions of its neighbors (Tutte, 1963)—setting the partial derivatives of the force function to zero results in independent systems of linear equations for the x-coordinate and the y-coordinate (Tutte, 1963; Tamassia & Rosen, 2013).

Di Battista et al. (1994) transformed Tutte (1963)'s method from a planar into a force-directed layout algorithm (Algorithm 3.7) wherein the force due to an edge is proportional to the distance between nodes u and v and edges have an ideal length of zero; there are no explicit repulsive forces (Equation 3.1). The algorithm

terminates when it *converges*, i.e., when the movement of all nodes is below some threshold $\epsilon \approxeq 0$.

$$F(t) = \sum_{\langle u,v \rangle \in E} \vec{\delta}_{uv} \tag{3.1}$$

Equation 3.1 describes the force at a given node $v \in V$, where $\vec{\boldsymbol{p}}_u$ and $\vec{\boldsymbol{p}}_v$ are the positions of nodes u and v, respectively. Partitioning the node set into fixed and free nodes guarantees this function has a trivial minimum with all free nodes placed in the same location.

Algorithm 3.7 The barycentric layout algorithm by Tutte (1963). Pseudocode from Di Battista et al. (1999)

▷ *Input: $G \leftarrow (V, E)$ with $V = V_0 \cap V_1$ with fixed vertices V_0 and free nodes V_1; a strictly convex polygon P with $|V_0|$ vertices.*
▷ *Output: a position $\vec{\boldsymbol{p}}_v$ for each nodes of V, such that the fixed vertices form a convex polygon P.*
V_0: place fixed nodes $u \in V_0$ at vertices of P
V_1: place free nodes $v \in V_1$ at the origin.

repeat
  for all free vertices $v \in V_1$ **do**
    $x_v \leftarrow \frac{1}{deg(v)} \sum_{\langle u,v \rangle \in E} \; x_u$
    $y_v \leftarrow \frac{1}{deg(v)} \sum_{\langle u,v \rangle \in E} \; y_u$
until x_v and y_v converge for all free nodes $v \in V_1$

The equations in the **for** loop are linear, and the number of equations equals the number of unknowns, which in turn equals the number of free nodes $|V_1|$. Solving these equations results in placing each free node at the barycenter of its neighbors. The solution is unique when solved using the Newton-Raphson method (Tamassia & Rosen, 2013). This **for** loop is trivially converted into *linear scan*-based operations, costing at most $4|V|$ scans (Algorithm 3.8).

Barycentric methods are generally considered inappropriate for large graphs;

Algorithm 3.8 The BARYCENTER layout algorithm with EGOs.

Input:
$G_t \leftarrow (V, E, A, t)$
$f_{init} \leftarrow$ FIXED-FREE ▷ *Partition the vertices into fixed V_0 and free V_1*
Output:
$L_{G_t} \leftarrow$ a reasonably nice layout of G_t

▷ *Get all connected nodes U, equivalent to the* NEIGHBORS *GDO*
function $f_{filter}(M_{G_t}, v)$
    $E \leftarrow \forall \langle v, u \rangle \in E_G \subset M_{G_t}$
    $U \leftarrow$ MAP$(u \leftarrow e.u)$
    return U

▷ *Vector-sum and scale positions of all nodes $u \in U$*
function SUM-SCALE(U, s)
    return $s \times \sum_{u \in U} \vec{\boldsymbol{p}}_u$

▷ *Test convergence for all free nodes $v \in V_1$*
function CONVERGED(ΔP_V)
    for all $\vec{\delta}_v \in \Delta P_V$ **do**
        if $\vec{\delta}_v > \epsilon$ **then**
            return `false`
    return `true`

function $f_{stop}(\Delta P_V)$
    return CONVERGED(ΔP_V)

▷ *Override* LAYOUT
function CALCULATE-OFFSETS(M_{G_t})
    for all $v \in V_1$ **do**
        $U \leftarrow$ FILTER-ELEMENTS(M_{G_t}, f_{filter})
        $s \leftarrow \frac{1}{deg(v)}$
        $\vec{\delta}_{uv} \leftarrow$ SUM-SCALE(U, s)
        $\vec{\delta}_v \leftarrow \vec{\boldsymbol{p}}_v - \vec{\delta}_{uv}$
    return ΔP_V

procedure BARYCENTER(G, f_{init})
    return FORCE-DIRECTED(G, f_{init})

the minimal node and edge separation tends to be very small, leading to resolution problems and unreadable drawings (Tamassia & Rosen, 2013). Currently, the biggest drawback to this method is the lack of constraint: for every $|V| > 1$, there exists a graph G such that the barycenter method computes a drawing with an exponential area (Tamassia & Rosen, 2013).

3.4. Spring Systems

In Eades (1984)'s *spring model*, the forces acting along each edge are modeled on coiled springs connecting nodes u and v.

Table 3.2. Common symbol notation for spring algorithms.

Constant	Definition
c_ϱ	The repulsion constant.
$\vec{c_\delta}$	The maximum translation offset vector for time t.
ℓ_{spring}	The natural spring length constant.
ℓ_{uv}	The length of the spring connecting u and v.
d_{uv}	The distance between nodes u and v ($\|\vec{\delta}_{uv}\|$).

Generally, the following assumptions are made: (1) spring strength is logarithmic, according to the spring Equation 3.2, where ℓ_{uv} is the length of the spring and c_ϱ and ℓ_{spring} are constants indicating the tensile force and the natural length of the spring, respectively; and (2) non-adjacent nodes repel each other using the inverse square law force, according to Equation 3.3 or Equation 3.4, where d_{uv} is the distance between the nodes.

$$F_{spring} = c_\varrho \times log(\ell_{uv}/\ell_{spring}) \tag{3.2}$$

$$F_r = \ell_{spring}/d_{uv}^2 \tag{3.3}$$

$$F_r = \ell_{spring}/\sqrt{d_{uv}} \tag{3.4}$$

The length ℓ_{uv} of the spring between nodes $u, v \in V$ corresponds to the desirable or ideal distance between any two nodes in the final layout:

$$\ell_{uv} = L \times d_{uv} \tag{3.5}$$

As mentioned earlier, aesthetically pleasing graph layouts are generated by minimizing the total energy E in the system as defined by some *energy function* (Equation 3.6). The optimal layout is where E is minimized for a particular algorithm (Kamada & Kawai, 1989).

$$E = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} \frac{1}{2} k_{uv}(d_{uv} - \ell_{uv})^2 \tag{3.6}$$

The distance d_{uv} between u and v is defined as the shortest path length between them; therefore, the length ℓ_{uv} is defined using Equation 3.5, where L is the desired length of any given edge in the layout space (Kamada & Kawai, 1989). As L is based on the distance between the farthest pair of nodes in a given graph, as shown in Equation 3.7, where L_0 is the minimum of the width and height of the layout space $L_0 = \min(S.width, S.height)$.

$$L = L_0 / \max_{|\vec{\boldsymbol{p}}_u|,|\vec{\boldsymbol{p}}_v|} d_{uv} \tag{3.7}$$

The parameter k_{uv} is the strength of the spring between u and v and is regarded as the square summation of the differences between the desirable distance and the actual distances for all pairs of nodes. According to Kamada & Kawai (1989), we wish to calculate these differences per unit length, defining k_{uv} using an approximation of Hooke's law thus:

$$k_{uv} = \frac{c_\varrho}{d_{uv}^2} \times \hat{\delta}_{uv} \tag{3.8}$$

The parameters ℓ_{uv} and k_{uv} are symmetric, thus $\ell_{uv} = \ell_{vu}, k_{uv} = k_{vu} \mid (\vec{\boldsymbol{p}}_u \neq \vec{\boldsymbol{p}}_v)$. Node positions not at equilibrium indicate a system with positive internal stress. As shown in Algorithm 3.9, node positions are randomized to ensure the stresses are at a maximum. The system is lowered into a minimal energy state via iteratively moving nodes at time t according to the *net force vector* $\vec{F_\Sigma}(v)$, the sum of all attractive and repulsive spring forces acting on v. After computing $\vec{F_\Sigma}(v)$, each node is moved a constant c_δ times this vector. This constant corresponds to the constrained maximum displacement per iteration to prevent excessive movement. By iteratively computing the forces on all nodes and updating positions accordingly, the system approaches a stable state where no further local improvements are possible. However, (Eades, 1984; Kobourov, 2013; Kamada & Kawai, 1989). Jeowicz et al. (2013) note that Equation 3.8 can cause undesirable scaling of either large or small graphs.

Eades (1984) further stipulates that the values $c_\varrho = 2$, $\ell_{spring} = 1$, and $c_\delta = 0.1$ are appropriate for most graphs, which approach their minimum energy state when

Algorithm 3.9 The spring embedder layout algorithm by Eades (1984). Pseudocode from Kobourov (2013)

```
▷ initial placement
for all v ∈ V do
    p⃗_v = RANDOM-POSITION
▷ repeat the placement process M times
for i = 1, ..., M do
    for all v ∈ V do
        F ← Σ forces acting on v
        p⃗_v ← |c⃗_δ| × F⃗_Σ(v)
```

$M = 100$. This algorithm succinctly demonstrates the elegance and natural intuition for all spring algorithms, with a compute cost of $\mathrm{O}(M \times |V|^2 + |V|)$. Building on Algorithm 3.6, we reimplement the spring layout algorithm using $\mathrm{O}(3(|V| \times M))$ scan operations. However, Eades' limit of 30 nodes (Eades, 1984) is insufficient for dynamic graphs where arbitrary numbers of nodes are added or removed; further refinements are required.

3.4.1 Fruchterman-Reingold

Fruchterman & Reingold (1991) redefined the attractive and repulsive forces on springs using Equation 3.10 and Equation 3.9, respectively, in terms of the distance d_{uv} between the nodes and the ideal distance between nodes to eliminate the cost of calculating the square root. ℓ_{spring} is likewise redefined using Equation 3.11. The cost of this algorithm is $\mathrm{O}(M \times (|V|^2 + |V| + |E|))$.

$$F_r = \frac{\ell_{spring}^2}{d_{uv}} \times \hat{\delta}_{uv} \tag{3.9}$$

$$F_a = \frac{d_{uv}^2}{\ell_{spring}} \times \hat{\delta}_{uv} \tag{3.10}$$

$$\ell_{spring} = \sqrt{\frac{area}{|V|}} \tag{3.11}$$

Algorithm 3.10 The SPRING layout algorithm with EGOs.

Input:
$G_t \leftarrow (V, E, A, t)$
$f_{init} \leftarrow$ RANDOM
$f_{filter} \leftarrow$ NEIGHBORS
Output:
$L_{G_t} \leftarrow$ a reasonably nice layout of G_t

▷ *calculate repulsive forces*
function REPULSE(V,v)
  return $\sum_{u \in V} \frac{c_\varrho}{d_{uv}^2} \times \hat{\delta}_{uv}$

▷ *calculate attractive forces*
function ATTRACT(U,v)
  return $\sum_{u \in U} c_\varrho \times \log \frac{d_{uv}}{\ell_{spring}} \times \hat{\delta}_{uv}$

function $f_{stop}(i)$
  return $i < M$

▷ *Override* LAYOUT
function CALCULATE-FORCES(M_{G_t})
  for all $v \in V$ **do**
    $V \leftarrow V_M \setminus v$
    ▷ *find connected nodes*
    $U \leftarrow$ FILTER-NODES(V, f_{filter})
    ▷ *calculate repulsive forces*
    $\vec{\delta}_v \leftarrow$ REPULSE(V, v)
    ▷ *calculate attractive forces*
    $\vec{\delta}_v \leftarrow \vec{\delta}_v +$ ATTRACT(U, v)
    ▷ *constrain movement*
    $\vec{\delta}_v \leftarrow \vec{\delta}_v \times c_\delta$
  return ΔP_V

procedure SPRING(G, f_{init})
  return FORCE-DIRECTED(G, f_{init})

For sparse graphs with uniform distributions of nodes, this can shrink to approximately $O(M \times (|V| + |V| + |E|))$. Jeowicz et al. (2013) further optimize this algorithm by swapping the calculation order in Equation 3.9 and Equation 3.10 to minimize the number of floating point calculations. Fruchterman & Reingold (1991) also introduce the concept of *temperature*, which steadily lowers the maximum displacement c_δ as the layout approaches the ideal configuration. The temperature cooling method ($\textsc{cool}(t)$) is not described but can be any monotonically decreasing function. The final positions $\vec{\boldsymbol{p}}_v \in P_G$ are also refined, so nodes are not displaced beyond the frame.

With appropriate materialization, the cost of ATTRACT is reduced to $|V|$ sequential scans but at the cost of $|V|$ equality conditionals which can cause threads to get out of sync, inducing contention (Jeowicz et al., 2013). For REPULSE, if nodes are unsorted, this devolves to $|E|$ random probes, a potentially undesirable outcome.

3.4.2 Edge-Edge Repulsion

Generally speaking, the notions of repulsion in the setting of conventional force-directed layouts fall into two categories: (1) *vertex-vertex repulsion*, where nodes repel each other, or (2) *vertex-edge repulsion*, where both nodes and edges repel each other (Lin & Yen, 2005). Theoretical and experimental results indicate both methods usually enjoy the merit of producing graph layouts with a high degree of symmetry and uniform edge length (Eades, 1984).

Angular resolution is an aesthetic quality of a graph layout, which refers to the smallest angle formed by two adjacent edges incident to a common node in a straight line drawing (Tutte, 1963; Formann et al., 1993). Both vertex-vertex and vertex-edge repulsion methods optimize the inter-node distance and the distance between nodes

Algorithm 3.11 Pseudocode for the force-directed layout algorithm by Fruchterman & Reingold (1991)

```
area ← W × L ▷ width and length of a frame
G ← (V, E) ▷ random initial positions for the vertices
k ← sqrt(area/|V|) ▷ the natural length of a spring
t ▷ temperature, max displacement
M ▷ the number of iterations

function F_a(x)
    return x^2/k

function F_r(x)
    return k^2/x

for i ← 1 to M do
    ▷ calculate repulsive forces
    for all v ∈ V do ▷ cost: O(|V|^2)
        ▷ each vertex has two vectors: .pos and .disp
        v.disp ← 0
        for all u ∈ V do
            if u ≠ v then
                δ ← v.pos − u.pos ▷ δ is the difference vector between u, v
                v.disp ← v.disp − (δ/|δ|)× F_r(|δ|)

    ▷ calculate attractive forces
    for all e ∈ E do ▷ cost: O(|E|)
        ▷ each edge e is an ordered pair of vertices ⟨u, v⟩
        δ ← e.v.pos − e.u.pos
        e.v.disp ← e.v.disp − (δ/|δ|)× F_a(|δ|)
        e.u.disp ← e.u.disp + (δ/|δ|)× F_a(|δ|)

    ▷ limit max displacement to the temperature t
    for all v ∈ V do ▷ cost: O(|V|)
        v.pos ← v.pos + (v.disp/|v.disp|) × min(v.disp, t)
        v.pos.x ← min(W/2, max(−W/2, v.pos.x))
        v.pos.y ← min(L/2, max(−L/2, v.pos.y))

    ▷ reduce the temperature as the layout condition improves
    t ← COOL(t)
```

Algorithm 3.12 The FRUCHTERMAN-REINGOLD layout algorithm with EGOs.

Input:
$G_t \leftarrow (V, E, A, t)$
$area \leftarrow S.width \times S.height$
$\ell_{spring} \leftarrow \sqrt{area/|V_G|}$
$f_{init} \leftarrow$ RANDOM
$f_{filter} \leftarrow$ NEIGHBORS

▷ *calculate repulsive forces*
function REPULSE(V,v)
    return $\sum_{u \in V} \frac{\ell^2_{spring}}{\vec{\delta}_{uv}} \times \hat{\delta}_{uv}$

▷ *calculate attractive forces*
function ATTRACT(U,v)
    for all $u \in U$ **do**
        $\delta \leftarrow (|\vec{\delta}_{uv}|^2/\ell_{spring}) \times \hat{\delta}_{uv}$
        $\vec{\delta}_u \leftarrow \vec{\delta}_u - \delta$
        $\vec{\delta}_v \leftarrow \vec{\delta}_v + \delta$
    return $\Delta\vec{\boldsymbol{p}}_v$

function $f_{stop}(i)$
    $c_\delta \leftarrow c_\delta \times .9$ ▷ *lower the temperature*
    return $i < M$

▷ *Override* LAYOUT
function CALCULATE-FORCES(M_{G_t})
    $V \leftarrow V_M \setminus v$
    for all $v \in V$ **do**
        ▷ *find connected nodes*
        $U \leftarrow$ FILTER-NODES(V, f_{filter})
        ▷ *calculate repulsive forces*
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow$ REPULSE(V, v)
        ▷ *calculate attractive forces*
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow \Delta\vec{\boldsymbol{p}}_v +$ ATTRACT(U, v)
        ▷ *constrain movement*
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow \Delta\vec{\boldsymbol{p}}_v \times c_\delta$
    return ΔP_V

from edges. Still, they do not prevent edges from being drawn co-linearly, a phenomenon known as "edge collapse" due to *zero angular resolution*, which indicates

function PLACE(M_{G_t})
    for all $v \in V_G$ **do**
        ▷ *set the position*
        $\vec{\boldsymbol{p}}_v \leftarrow \vec{\boldsymbol{p}}_v + \hat{\vec{\delta}}_v \times \min(|\vec{\delta}_v|, c_\delta)$
        ▷ *fit within bounds*
        $w \leftarrow S.width/2$
        $h \leftarrow S.height/2$
        $\vec{\boldsymbol{p}}_v.x \leftarrow \min(w, \max(-w, \vec{\boldsymbol{p}}_v.x))$
        $\vec{\boldsymbol{p}}_v.y \leftarrow \min(h, \max(-h, \vec{\boldsymbol{p}}_v.y))$

procedure FRUCHTERMAN-REINGOLD(G, f_{init})
    return SPRING(G, f_{init})

at least two co-incident edges overlap, resulting in a bad drawing with simultaneous edge-edge and node-edge crossings (Chuang & Ahuja, 1998; Lin & Yen, 2005). The *simulated annealing* method considering an angular resolution term (Ioannidis & Wong, 1987b; Fruchterman & Reingold, 1991) can be applied, but it is not regarded as efficient (Lin & Yen, 2005).

Drawing graphs without zero angular momentum is found in various computer science and engineering applications. Still, it is essential in exploratory graph visualizations, mainly when rendering graphs with non-uniform nodes (i.e., nodes with differing visual marks or whose visual channels vary considerably across the graph) (Lin & Yen, 2005). Analytically derived formulas of repulsive forces between two charged edges are unnecessarily complicated to implement practically; therefore Lin & Yen (2005) describes useful approximations which are straightforward to implement (Algorithm 3.13).

Lin & Yen (2005) describe a novel repulsion mechanism where edges are replaced with charged springs (Lin & Yen, 2005), instead of adding charges to nodes (Eades, 1984). This maximizes the angular resolution of the drawing but still minimizes the energy at any given node. This *edge-edge repulsion* algorithm (Algorithm 3.13) is based on the theory of *potential fields* to draw graphs without zero angular

resolution. The main aesthetic criteria (Section 2.5) addressed by the edge-edge repulsion algorithm are symmetry, uniform edge length, and maximization of angular resolution (Lin & Yen, 2005).

A drawback to this method is that it creates a more symmetrical drawing at the expense of allowing node-node overlap in the final output, which may, in turn, obscure features necessary to preserve the mental map (Lin & Yen, 2005; Argyriou et al., 2012). The attractive force function is identical to Equation 3.2. However, the repulsive force differs significantly, being defined in terms of the lengths of the two edges and the included angle between them: The magnitude of the repulsive force due to any pair of edges is *positively* correlated with their respective lengths, and *negatively* correlated with the angle (Lin & Yen, 2005). Calculation of this force requires a local sorting of nodes via a vector cross product, which may be expensive in both memory space ($|E|^2$) and random probes ($|V|$) for highly-connected graphs. REPULSE is trivially implemented with $6|V|$ linear FILTER scans, but we believe this could be reduced to $5|V|$, using four MAP and one REDUCE scan. Testing this hypothesis is reserved for future work.

3.5. Other Algorithms

Most force-directed layout algorithms restrict graph layouts to Euclidean geometry, typically $\mathbb{R}^2$, $\mathbb{R}^3$, and more recently, $\mathbb{R}^n$ for larger values of n. However, are cases where Euclidean geometry may not be the best option, such as when graphs are embedded on a torus and must be laid out without edge crossings (Tamassia & Rosen, 2013). In general, these algorithms take as input a force-directed layout and apply non-linear modifications in the FINESSE stage to adjust node positions *post hoc*. Such drawings are out of the scope of this thesis and will not be discussed further, although Dynamical.JS is capable of extension to support non-Euclidean and spectral

algorithms.

Algorithm 3.13 The edge-edge repulsion algorithm by Lin & Yen (2005)

Input

G_t A reasonably nice layout of graph $G \leftarrow (V, E, A, t)$

function $f_a(d)$
  return $C_1 \times log(d/C_2)$

function $f_1(f,u)$
  return $|f|u_{f_1}$

$F_{tmp}[|V|]$ ▷ *Temporary forces* $\forall v \in V$
P_{new} ▷ *Record new positions* $\forall v \in V$
P_{old} ▷ *Record old positions* $\forall v \in V$
C_6 ▷ *Magnitude of movement in each iteration*

▷ *Assign initial locations of vertices*
▷ *Determine the neighboring order of adjacency edges of each vertex using outer product*
while $converged \neq 1$ **do** ▷ *Convergence loop*
  $converged \leftarrow 1$
  $oldPosn \leftarrow newPosn$
  $tempForce[|V|] \leftarrow$ zeros matrix
  for all $v \in V$ **do**
    if ∃ at least two edges incident to v **then**
      for all pair (e_i,e_j) where $e_i = (v, v_i), e_j = (v, v_j)$ are neighboring edges incident to v, and e_i is the right edge of their included angle with smaller degree **do**
        ▷ *calculate the repulsive force* f_1 *at* e_i *due to* e_j *according to (8)*
        $tmpForce[v_i] \leftarrow tmpForce[v_i] + f_1$
        $tmpForce[v_j] \leftarrow tmpForce[v_j] - f_1$

  for all $e = (v_i, v_j) \in E$ **do**
    $tmpForce[v_i] \leftarrow tmpForce[v_i] + f_a$
    $tmpForce[v_j] \leftarrow tmpForce[v_j] - f_a$

▷ *Draw graph and simultaneously save new positions to newPosn according to* $C_6 \times tmpForce[|V|]$ *where* C_6 *is a constant to control the magnitude of movement in each iteration*
if $||newPosn - oldPosn|| > \epsilon$ **then**
  $converged \leftarrow 0$

return ▷ *A nice drawing of* G *without zero angular resolution*

Algorithm 3.14 The EDGE-EDGE-REPULSION algorithm with EGOs (Exploratory Graph Operations)

Input:
$G_t \leftarrow (V, E, A, t)$
$W \leftarrow S.width$
$H \leftarrow S.height$
$f_{init} \leftarrow \text{LAYOUT}(G_t)$ ▷ *a prior layout algorithm*
Output:
$L_{G_t} \leftarrow$ a nice layout of G_t without zero angular resolution

▷ *get the cross product of* $E \times E$ *as a* 5*-tuple*
▷ $f : E \times E \to Q$
▷ $q \leftarrow \langle v_0, v_1, \vec{\delta_0}, \vec{\delta_1}, f_0, f_1, \theta \rangle \in Q$
function $f_{map}(E)$
    for all $\langle e_0, e_1 \rangle \in E \times E$ **do**
        $\vec{\delta_0} \leftarrow e_0.\vec{\boldsymbol{p}}_v - e_0.\vec{\boldsymbol{p}}_u$ ▷ *the difference vector for* e_0
        $\vec{\delta_1} \leftarrow e_1.\vec{\boldsymbol{p}}_v - e_1.\vec{\boldsymbol{p}}_u$ ▷ *the difference vector for* e_1
        $\theta \leftarrow \tan^{-1}(\frac{|\vec{\delta_0}|}{W}) + \tan^{-1}(\frac{|\vec{\delta_1}|}{W})$ ▷ *the angle between* e_0 *and* e_1
        $q \leftarrow \langle e_0, e_1, \vec{\delta_0}, \vec{\delta_1}, |\theta| \rangle$
    return Q

▷ *filter out nodes with degree less than 2*
function $f_{degree}(v)$
    return $u.degree > 1$

▷ *filter out edges with tiny magnitudes*
function $f_\epsilon(q)$
    return $|q.\vec{\delta_0}| > \epsilon \wedge |q.\vec{\delta_1}| > \epsilon$

▷ *Override* FORCE-DIRECTED
function REPULSE(Q,v)
    $\mathbf{R} \leftarrow \left[\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right]$ ▷ *rotation matrix* $-\frac{\pi}{2}$
    $\theta \leftarrow 2\pi / v.degree$

    $Q_l \leftarrow \text{FILTER}(Q, q.e0.v = v)$ ▷ *lhs*
    for all $q \in Q_l$ **do**
        $\hat{\delta} \leftarrow q.\vec{\delta}_0 / |q.\vec{\delta}_0|$
        $\vec{\delta_v} \leftarrow \hat{\delta} \times \sin((\theta - q.\theta)/2)$
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow \Delta\vec{\boldsymbol{p}}_v + c_\delta \mathbf{R}\vec{\delta_v}$

    $Q_r \leftarrow \text{FILTER}(Q, q.e1.v = v)$ ▷ *rhs*
    for all $q \in Q_r$ **do**
        $\hat{\delta} \leftarrow q.\vec{\delta}_1 / |q.\vec{\delta}_1|$
        $\vec{\delta_v} \leftarrow \hat{\delta} \times \sin((\theta - q.\theta)/2)$
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow \Delta\vec{\boldsymbol{p}}_v - c_\delta \mathbf{R}\vec{\delta_v}$
    return $\Delta\vec{\boldsymbol{p}}_v$

function CALCULATE-STATISTICS(M_{G_t})
    $Q \leftarrow \text{MAP}(E, f_{map})$
    $Q \leftarrow \text{SORT}(Q, q.\theta)$
    $Q \leftarrow \text{FILTER}(Q, f_\epsilon)$
    return Q

function CALCULATE-FORCES(M_{G_t})
    ▷ *only repulsive forces for nodes with degree* > 1
    $V \leftarrow \text{FILTER-NODES}(f_{degree})$
    for all $v \in V$ **do**
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow \text{REPULSE}(Q, v)$

    ▷ *attractive forces on all nodes*
    for all $v \in V_G$ **do**
        $\Delta\vec{\boldsymbol{p}}_v \leftarrow \Delta\vec{\boldsymbol{p}}_v + \text{ATTRACT}(V_G, v)$
    return ΔP_V

procedure EDGE-EDGE-REPULSION(G_t,f_{init})
    return FORCE-DIRECTED(G_t, f_{init})

Chapter IV.

Computational Development

The following section outlines our development process, the strengths and weaknesses of our development platform, and rationales for each of our design choices ultimately implemented in Dynamical.JS.

4.1. External Requirements

To support our core design goal of self-containment, the Dynamical.JS framework neither embeds nor includes any software, libraries, plugins, or other code to function. The thesis author wrote all code included with Dynamical.JS. Consequently, Dynamical.JS does not require the acquisition of any external software licenses. Only publicly defined and generally-available Javascript APIs (European Association for Standardizing Information and Communication Systems, 2020) were used, and only those algorithms made public by standards organizations such as the W3C (World Wide Web Consortium) (World Wide Web Consortium, 2022d) or in the published literature are implemented. Citations for these organizations, their standards, and reference documents are listed in the References section of this thesis and are documented in the project source code using JSDoc (Mathews, 2011) `@cite` directives. Therefore, all code distributed within the Dynamical.JS framework can be run on any system running a modern Javascript engine with or without support for WebGPU (World Wide Web Consortium, 2022b).

4.2. Publishing

The reference implementation of the Dynamical.JS framework is published on GitHub. The final publication license has not been finalized. However, we intend that

the published license be fully compatible with the licenses released by the previously-mentioned standards bodies, specifically, the licenses for WebGPU (GPU Computing for the Web) (World Wide Web Consortium, 2022b) and WGSL (WebGPU Shading Language) (World Wide Web Consortium, 2022c), which are unavailable at the time of this writing.

4.3. Hardware Configuration

All code developed for Dynamical.JS was developed and tested on our own hardware, using the most current publicly-available versions of the following Webkit-based (Apple Inc., 2021) browsers: Google Chrome Canary (Google Inc., 2022), Apple Safari (Apple Inc., 2022b), and Microsoft Edge (Microsoft Inc., 2021). The Firefox (Mozilla Corp., 2021) web browser, which is not based on Webkit was also tested for completeness. All code was written in Javascript and WGSL and on the full suite of browsers before publication.

Code was run on an Apple MacBook Air (M2, 2022) running macOS Monterey version 12.6 and macOS Ventura version 14.0.1 with an Apple M2 SOC (System-On-a-Chip) containing an 8-core (four performance cores running at 3.5GHz, four efficiency cores running at 2.4GHz) CPU (Central Processing Unit) and a 10-core integrated GPU (Graphics Processing Unit) and sharing 16GiB of unified LPDDR5-6400 memory. Storage consisted of a single 512GiB SSD (Solid State Drive) formatted as a single APFS (APple File System) volume with a block size of 4096B. The memory cache sizes for the performance cores were 192KiB (L1i), 128Kib (L1d) 256KiB, 16MiB (L2); cache sizes for the efficiency cores were 128KiB (L1i), 64Kib (L1d) 256KiB, 4MiB (L2), with a page size of 16384KiB (65535B). Each GPU core contains 32 execution units, containing eight ALUs (Arithmetic Logic Units). The M2 GPU contains 320 execution units and 2560 ALUs, with a maximum floating point (FP32) performance

of 3.6TFLOPs (Tera- Floating Point Operations s^{-1}).

Although GPU hardware is ubiquitous in modern computing systems, there are several well-known memory transfer and caching caveats (Tan et al., 2020; Qu et al., 2017; Xiao et al., 2015), which Dynamical.JS aims to address in code. However, our substrate packages several CPU and GPU cores and a UMA (Unified Memory Architecture) (N & Murali, 2019) within a single SOC that ameliorates several of these problems and masks potential issues that plague substrates with discrete CPU and GPU cores. Dynamical.JS includes system-agnostic code wherever possible, and we intend to profile this code on multiple substrates in future iterations of this project.

4.4. Tools

Due to browser security constraints, Javascript programming requires code to be run via a local web server, as local file access is not possible in most cases due to browser security constraints (European Association for Standardizing Information and Communication Systems, 2020). Therefore, an IDE (Integrated Development Environment) which includes an embedded web server, is required. These embedded web servers facilitate local Javascript debugging yet are still quite limited when debugging new graphical frameworks such as WebGPU. The Nova (Panic, 2021) IDE was used exclusively to program Dynamical.JS. There is no particular motivation behind the choice except familiarity, ease of use, and convenience. From experience, most Javascript IDEs have comparable features. Node.js (OpenJS Foundation, 2022) was used to compile or 'minify' the project into a distribution package during testing but is not used otherwise and is not required to use, embed, or extend Dynamical.JS on any system. Superstatic (Firebase, 2022) is a minimal web server used during development to support testing over our LAN (Local Area Network). The Web Inspector built into all Webkit browsers (Apple Inc., 2021; Google Inc., 2021, 2022) was

used for realtime debugging and testing. Google Chrome Canary (Google Inc., 2022) is the browser with the most current and robust implementation of the WebGPU framework and was, therefore, the main development browser for Dynamical.JS. Safari Technology Preview (Apple Inc., 2021) is the primary Webkit implementation upon which several other browsers (Chrome, Edge) are based. As of the publication of this thesis, the initial WebGPU implementation is absent from compiled distribution. A complete replacement implementation is included in the Webkit source code but is not fully integrated into any currently available browser. It is expected to be re-integrated by the end of 2022 or early 2023. Firefox Nightly also supports WebGPU, but the implementation hasn't been updated in several months and no longer conforms to either the WebGPU or WGSL specifications. Development and testing on Firefox and Safari have been suspended until standards-conformance tests are complete.

4.5. Javascript

The Dynamical.JS framework is written in ECMAScript (European Association for Standardizing Information and Communication Systems, 2020), better known as Javascript, due to the increased usage of web-based systems and the ubiquity and power of modern web browsers. These browsers contain JIT (Just-in-Time) compilers, which generate bytecode representations of Javascript programs which run nearly as fast as their assembly-language counterparts. ECMAScript 6+ (European Association for Standardizing Information and Communication Systems, 2015) has a simplified object-oriented syntax similar to C, C++, and Swift, allowing Dynamical.JS to be more easily ported to these languages in the future should the need arise. Additionally, web-based multivariate network exploration applications will enable analysts to quickly and effortlessly filter and analyze networks on demand (No-

bre et al., 2019) without requiring specialized library code, compilers, or systems knowledge. The system-agnostic nature of the Javascript platform, which requires only a modern web browser, makes Javascript an ideal implementation language for this project. The current Javascript standard exposes a variety of computational substrate configurations, including SMP (Symmetric MultiProcessing), AMP (Asymmetric MultiProcessing), (limited) multi-threading, and direct GPU access through both WebGL (Web Graphics Library) (Khronos® Group, 2014, 2017) and WebGPU (Dakkak et al., 2016) interfaces. Lower-level programming languages such as C or C++ were considered and rejected for this project. These languages are robust and well-supported, but their need for external compilers and system-specific libraries rendered them unsuitable.

The intricacies of the Javascript runtime are beyond the scope of this document. Therefore the following section only discusses components relevant to the implementation of graph layout algorithms and Dynamical.JS.

4.5.1 Runtime & Concurrency

Unlike many development platforms, the Javascript runtime (also known as a *user agent*, *agent*, or *browser*) uses an *event loop* concurrency model in lieu of threads (European Association for Standardizing Information and Communication Systems, 2015). Each agent is a monolithic execution context consisting of a call stack and an event loop. At its core, an event loop simply waits for *messages* (disambiguated from *events*) indicating computational *jobs* have been submitted into one or more *job queues*. As messages are received, jobs are dequeued and executed in FIFO (First In, First Out) order: the job is removed from the queue, and its corresponding function is called with the message as input (European Association for Standardizing Information and Communication Systems, 2021). Each function call creates a new stack frame

Algorithm 4.1 Javascript event loop

```
while 1 do
    while jobQueue.empty = false do ▷ Empty the job queue.
        EXECUTE(jobQueue.dequeue())
        if isRenderingOpportunity = true then ▷ Update the screen.
            RENDER-SCREEN()
    if asyncJobQueue.empty = false then ▷ Enqueue the next async job.
        jobQueue.enqueue(asyncJobQueue.dequeue())
```

for that function's use (Mozilla Developer Network, 2022).

The above term "queue" is a logical misnomer: job queues are implemented as *sets*, not queues, and the event loop processing model grabs the oldest *runnable* job from the chosen queue instead of dequeuing the first job (Web Hypertext Application Technology Working Group, 2021a). Jobs are only runnable if they are neither waiting for another job to complete, e.g., an unresolved *promise* (Section 4.5.2), nor for a system resource to become available, e.g., a file handle or network data packet. All global functions and events triggered by a webpage or analyst, such as page loading, scrolling, or click events, are enqueued as jobs. Each job must empty its stack before any other jobs can be executed and cannot be preempted (European Association for Standardizing Information and Communication Systems, 2021). This is problematic because long-running jobs, such as graph layout algorithms, will delay the execution of salient tasks (such as screen rendering) until execution completes (Algorithm 4.1).

Screen rendering is only available when *rendering opportunities* arise: if the user agent attempts to achieve a 60Hz refresh rate, rendering opportunities occur *at most* every $60fs^{-1}$ or ≈ 16.7ms. If this rate is unsustainable, the user agent may drop to a more sustainable $30fs^{-1}$ rate or even lower rather than occasionally dropping frames. The sustainability threshold is implementation-defined and, therefore, unpredictable (Web Hypertext Application Technology Working Group, 2021b). Abrupt changes in refresh rate and frame-dropping result in stuttered animation, leading to

negative analyst experiences and disrupted mental maps.

Newer versions of the Javascript runtime implement a concurrent execution model but do not implement "true" threads. Instead, the runtime has the concept of a *worker*, a subordinate execution context with a separate event loop and job queue (European Association for Standardizing Information and Communication Systems, 2021). Unlike threads defined in languages like C or C++, these worker contexts do not share memory and cannot be synchronized. Worker interaction consists of exchanging messages containing *serialized* objects or special memory buffer references, so graph layout algorithms that rely on the multithreaded scatter-gather design pattern cannot be directly ported to Javascript. Instead, such algorithms must be adapted to work in a purely asynchronous environment; serialization is time-consuming for graphs with hundreds or thousands of nodes and may result in stuttered animations or unresponsive screens if large graphs must be [re/de]serialized during each message.

4.5.2 Asynchronous Execution

The `async`/`await` system introduced in ES7 (ECMAScript 7) is neither truly asynchronous nor concurrent; these execution models are emulated by issuing *promises* that execution will be completed sometime in the future. This allows long-running jobs to execute without blocking the main event loop by deferring execution until enough processing time is available. In practice, jobs labeled `async` are deferred until the current job queue is *empty*. When an `async` function `await`s the completion of another `async` function, it is simply re-queued so that its execution completes at some point after the `await`ed function returns, allowing other tasks (such as UI (User Interface) updates) to complete before resuming execution. Consequently, timers created by `setTimeout( delay )` are not executed until *at least* `delay` milliseconds

have elapsed. Therefore, they cannot ensure smooth animation because they cannot synchronize with the main event loop's rendering opportunities.

Section 5.5, Section 5.9, and Section 5.10 below describe how the Dynamical.JS framework modifies graph layout and rendering algorithms to take advantage of the Javascript event loop.

4.6. Computer Graphics

The creation of a real-time graph layout framework requires a holistic understanding of the computer graphics pipeline, and how optimizations targeting one component may introduce bottlenecks in another. This section examines the concepts and technologies which underlie the graphics pipeline, how they are employed to create images on the screen, and how they can be repurposed to perform useful general-purpose computing.

4.6.1 History

In the early days of computer graphics, there were no standard programming models. Each hardware vendor developed proprietary interfaces for rendering geometry onto graphical displays (Bailey & Cunningham, 2009). Primitive systems and substrates, such as the Macintosh or Amiga, had either a separate, monolithic VRAM (Video RAM) or a dedicated segment of DRAM (Dynamic RAM), which held the contents of a single framebuffer. Drawing commands submitted to the system renderer queue would "trap" the CPU, i.e., interrupt normal CPU operations, and execute functions hard-coded in ROM (Read Only Memory) to render graphical primitives into the framebuffer (Foley et al., 1996; Apple Computer, Inc., 1985). The contents of this framebuffer were then transferred to the display during the VBL (Vertical BLanking interval)—the interval during which no screen drawing is taking

place (Apple Computer, Inc., 1984). Systems running on other substrates, such as Windows/X86, would perform similar operations on code stored in SRAM (Static RAM) using specialized hardware (Foley et al., 1996). A few substrates improved to support multiple output displays, each with distinct framebuffers and graphical pipelines, while others focused on improving single-display performance (Foley et al., 1996). This divergence was neither efficient nor portable: each system required specialized software and hardware to perform even simple graphical rendering tasks, preventing code written for one system from being easily ported to another without enormous cost.

4.6.2 OpenGL

OpenGL (Open Graphics Library) (Segal & Akeley, 2022) was developed to create a standard, stable API (Application Programming Interface) for vendors to implement on top of their proprietary graphics platforms. The OpenGL API became the "gold standard" API because it does not assume hardware support; the standard only specifies supported graphical operations and calling conventions; the standard includes neither implementation details nor performance targets. OpenGL was designed primarily to provide convenient access to all of the capabilities of the underlying substrate rather than facilitate any particular use of the hardware (Ebert et al., 2003; Segal & Akeley, 2022). Compliant systems will therefore run any OpenGL-linked application, including those running on low-power devices (i.e., mobile phones) without dedicated graphics processing hardware (Bailey & Cunningham, 2009). However, realtime applications like games or physics simulations need to create images at interactive speeds, necessitating specialized high-speed circuitry. Simple VRAM/framebuffer-based substrates were replaced by "graphics cards" containing ASICs (Application-Specific Integrated Circuits), which included hardware-

optimized graphical operations and dedicated VRAM (Bailey & Cunningham, 2009). These ASICs became increasingly sophisticated, with performance many orders of magnitude higher than CPUs while supporting the same graphical operations originally defined initially in the OpenGL specification. Though it remains the "standard" graphics API, OpenGL often requires the use of vendor-specific GPU extensions to access the latest hardware features (Ebert et al., 2003); therefore, several new graphics frameworks, such as Vulkan (Khronos® Group, 2022b), Metal (Apple Inc., 2022a), and CUDA (Compute Unified Device Architecture) (NVIDIA Corp., 2022b) have been developed to supplement or supplant it. One such framework, WebGPU, underlies the core functionality of Dynamical.JS and is discussed in Section 4.12.2.

4.7. Graphical Object Models

Graphical *scenes* are conceptual arrangements of geometric objects to be rendered to a display. In order for an application to render these scenes, it must first articulate its requirements using graphical primitives and applicable operations into a graphical *model* (Bailey & Cunningham, 2009). Geometric objects are *modeled* by the application and stored in a local, display-independent coordinate system known as *model space*. These objects may be placed into a scene by transforming them into a shared "world" coordinate system shared by all objects in the scene known as *world space*. The placement of a model into a scene is called *instancing*: the modeled object is known as the *master*, and the transformed copy placed in a scene is called an *instance* (Ebert et al., 2003). Individual models can be combined into composite objects, and composite objects subsequently instanced into a scene, thus building up an object hierarchy known as a *scenegraph*, which arranges the logical and spatial organization of the graphical objects to be rendered. Each node of the scenegraph corresponds to an instance, composite, or transformational process. Simple

scenegraphs may be structured as trees, but those with many instances are generally directed acyclic graphs, such that any node may have more than one parent in the hierarchy (Ebert et al., 2003). Scenegraphs are designed to be spatially coherent, easily serialized for storage and retrieval, and efficiently processed into the graphical primitives required for rendering. Scenegraphs can also be queried or "pruned," thereby restricting processing to instances that lie on a particular path between two arbitrary nodes or fall within a specified bounding region (Ebert et al., 2003).

As a scenegraph is *traversed*, type-specific operations on each node are dispatched into a rendering queue or *pipeline* for execution. These operations transform model instances into geometrical *primitives* that are subsequently enqueued with appropriate rendering commands. Additionally, many graphical operations produce no output, only *side effects*: their only purpose is to modify or query the current contextual *state*. For example, when a transformation node is processed, an operational command to accumulate an affine transformation onto the current *model-view matrix* (Section 4.8.1) is issued, which then applies to all descendants of that node unless another transformational command is issued (Akenine-Möller et al., 2008).

Most graphical operations are beyond the scope of this document but can be categorized as operations that: (1) define and manipulate the current graphical *context*; (2) define and manipulate the drawing area, *canvas*, or *surface*; (3) define *geometry*; and (4) modify appearance *properties* of specified geometry.

1. Graphical *contexts* are defined by the current computational substrate's facilities, such as available CPUs and GPUs, memory size and layout, pixel format, and blending functions. Each context controls a set of memory segments– *framebuffers*, where interim data and final rendered output are stored. Contexts also maintain several auxiliary computational and storage buffers for transferring data between CPU and GPU and rules for inter-converting data as they

move through the pixel pipeline.

2. A graphical *surface* (also known as a *view*, *viewing area*, or *canvas*) is a bounded region of framebuffer memory containing rendered pixels. Each surface maps directly to a visible region on a display, a texture, or an off-screen buffer. The surface's contents may then be drawn directly to the display, mapped to new geometry, or returned to the application for further processing.

3. Element *geometry* is defined by vertices (e.g., spatial coordinates $\vec{\boldsymbol{p}}_v \leftarrow \langle x, y, \ldots, n \rangle \in \mathbb{R}^n$), graphical primitives, and normal vectors (vectors orthogonal to the primitive plane $\vec{\mathbf{n}} \leftarrow \vec{\mathbf{v}} \times \vec{\mathbf{u}}$). Graphical primitives may be points, lines, polygons, groups of primitives, or functions that generate primitives procedurally (Bailey & Cunningham, 2009). A sequence of viewing and projection matrices, collectively known as the *model-view matrix*, map geometry from the initial model space into a bounded, two-dimensional *view space* defined by a surface $\mathbf{M}^n : \mathbb{M}_0^n \cdots \mathbb{M}_i^n \rightarrow \mathbb{V}^2 \subset \mathbb{R}^2$. Geometry that falls outside the bounds of view space or whose mapped normals point away from the screen may be culled or clipped to prevent further processing, saving computational resources.

4. Geometrical primitives have appearance *properties* such as color, shading, materials, and lighting. Pre-defined and procedurally-generated images may be mapped as complex appearance properties, called *textures*, which are then mapped directly onto geometrical primitives. Output rendered to a particular surface may also be reprocessed by the application and re-used as a texture before being remapped onto new geometry, using a technique known as *multi-pass rendering*.

Traversing the scenegraph and generating the model is known as the *application stage*. This stage is generally executed in software running on general-purpose CPUs and is typically the slowest component of graphics processing. Some of the tasks traditionally performed in this stage include collision detection, global acceleration algorithms, and physics simulation, among others. It is within this stage that GPU-enabled graph layout algorithms spend most of their time and where the most aggressive optimization tactics are employed.

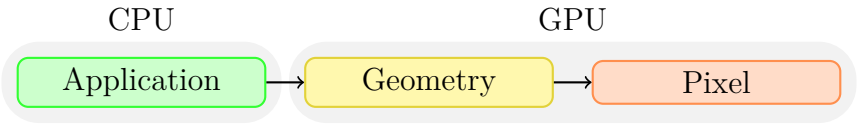


Figure 4.1. Graphics rendering pipeline stages.

At the end of the application stage, the model is streamed to the GPU rendering interface as a sequence of commands. The renderer then dispatches these commands to specialized queues called *pipelines*. Basic GPU renderers are a pair of linked pipelines corresponding to subsequent rendering stages: the *geometry pipeline* and the *pixel pipeline* (Figure 4.1). Each pipeline has a fixed set of functions and operates as a "true" queue: operations are always performed serially in FIFO order. Even though pipelined operations are executed sequentially, when dequeued for execution mid-stage, certain operations run simultaneously on many (hundreds to thousands) GPU ALUs (Akenine-Möller et al., 2008).

Further, each pipeline operates concurrently with the other: while the geometry pipeline processes modeled geometry recently submitted by the application stage (Figure 4.2, **M**), the pixel pipeline processes the transformed vertex output from the geometry pipeline (Figure 4.2, **V**). The final output of the pixel pipeline is an array

of pixels containing a rendered image (Figure 4.2, **P**), which is finally copied to the target surface's framebuffer for display. Render pipeline output may also be fed back into the input of either pipeline, facilitating multipass rendering, used by a significant number of graphical operations such as "picking" and selection, color filtering, edge detection, and smoothing (Bailey & Cunningham, 2009). Multipass rendering also forms the basis of several of the EGOs (Exploratory Graph Operations) defined in Section 2.7.

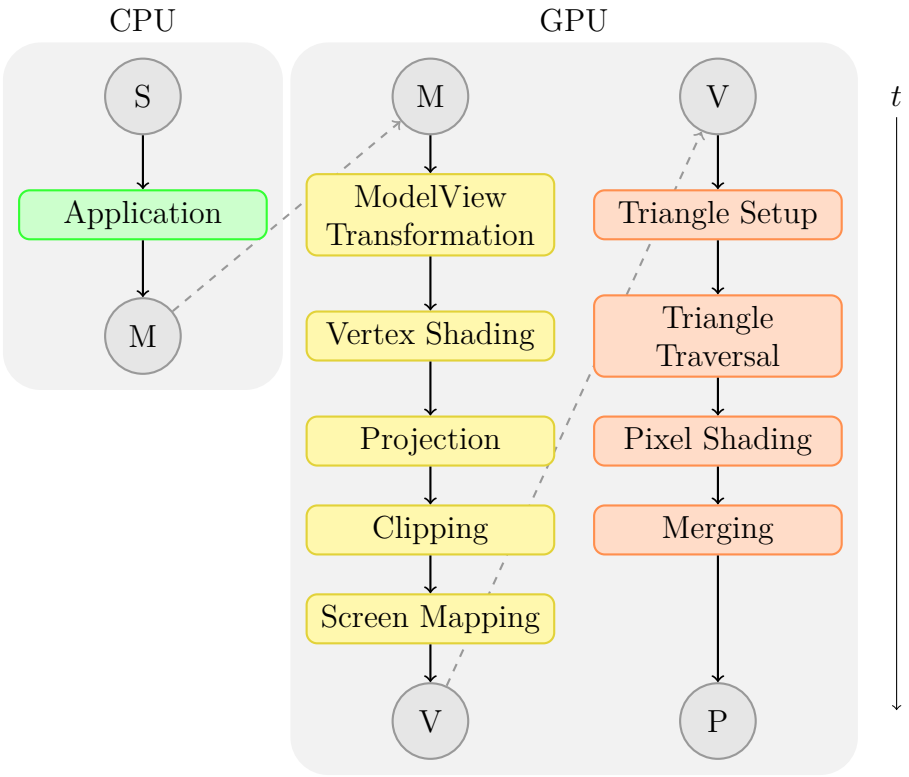


Figure 4.2. The "classic" rendering pipelines.

4.8.1 The Geometry Pipeline

The geometry pipeline processes most per-polygon and per-vertex operations submitted to the GPU (Ebert et al., 2003). This pipeline stage is divided into five basic sub-stages: (1) model-view transform, (2) vertex shading, (3) projection, (4)

clipping, and (5) screen mapping (Figure 4.2). However, some of these stages are combined in certain implementations, while others are further subdivided into more granular stages.

1. The first stage of the geometry pipeline is the *model-view transform stage*. Before a graphical element is displayed on a screen, the primitives contained within its model have transformed through several *spaces* or *coordinate systems*. Initially, the model resides in its own *model space* and has not yet been transformed. The model can then be associated with a *model transform* to be positioned and oriented. It is possible to have several model transforms associated with a single model. Each transform is a sequence of 4×4 matrix multiplications that may scale, rotate, reflect, or otherwise alter all vertices and normals within the model. This allows several *instances* of the same model to have different locations, orientations, and sizes within a scene without replicating the basic geometry. Once the transform has been applied, the model is located in a global coordinate system known as *world space* (Akenine-Möller et al., 2008). Optionally, there is an intermediate *vertex generation* stage, where new vertices are added or removed from a model to synthesize complex geometry such as curves or spheres or to deform the geometry of existing model instances. Only models visible to the "observer" (i.e., the *camera*) within world space are rendered. The camera has a location and orientation in world space used for placement and aiming. To facilitate projection and clipping, the camera and all visible models are then transformed with the *view transform* into what is known as *camera space*, *eye space*, or *view space* (Akenine-Möller et al., 2008). Some applications define more than one camera, allowing the same scene to be rendered from different perspectives or with other properties (Bailey & Cunningham, 2009; Segal & Akeley, 2022). This stage ends with *vertex transformation*, which takes the

set of vertex attributes (eye space coordinates, normals, texture coordinates, &c.) and produces a new set of attributes suitable for clipping and rasterization (such as homogeneous clip-space position, vertex lighting, sampled texture coordinates, &c.) (Cebenoyan, 2004).

2. Once the model's geometry has been transformed relative to the camera in view space, the vertex *shading stage* determines which pixels must be filled to represent it (Cebenoyan, 2004). Basic vertex coloring is determined during a process known as *shading*: specified appearance properties (such as lighting or materials) are combined using a *shading equation* and are stored along with the vertex attributes. Only primary color values are assigned to each vertex; final pixel output coloring is deferred to later stages (Segal & Akeley, 2022).

3. In the *projection stage*, the coordinates of all models in the view space are transformed again. They are *projected* into a unit cube bounded by coordinates $\langle -1, -1, -1 \rangle$ and $\langle 1, 1, 1 \rangle$ known as the *canonical view volume*. This transform may be *orthographic* or *parallel*, and the final coordinates are said to be in *normalized device coordinates* (Akenine-Möller et al., 2008). While the projection matrix transforms one space into another, the z-coordinate is ignored, "projecting" the volume onto a two-dimensional surface.

4. Only primitives that fit within the canonical view volume need to be passed on to the following stages, therefore any primitives that fall outside are discarded (e.g., *culled*) or subdivided (e.g., *clipped*) during the *clipping stage* (Bailey & Cunningham, 2009).

5. All remaining primitives are passed on to the *screen-mapping stage* and are finally mapped into the two-dimensional *window* or *screen coordinates* corresponding to display pixels (Akenine-Möller et al., 2008; Segal & Akeley, 2022).

4.8.2 The Pixel Pipeline

Output from the geometry pipeline (vertex position, depth, color, texture coordinates, &c.) defines the set of pixels to be rendered. The pixel pipeline stages combine the input vertex specifications, appearance properties, and processing instructions to fill the output framebuffer with colored pixels (Akenine-Möller et al., 2008). A key concept in rendering is *triangulation*: GPUs use triangular polygons as their primary geometric primitives. All primitives, including points, lines, and other polygons, are reinterpreted as either triangle strips (e.g., lines, complex shapes) or triangle fans (e.g., points, convex polygons). Any convex polygon can be triangulated by choosing an arbitrary vertex and constructing a triangle fan or strip by processing the following vertices sequentially. All subsequent shading computations are performed by interpolating (Section 2.10.10) between the vertices of these triangles (Ebert et al., 2003).

The pixel pipeline stage is also subdivided into distinct sub-stages: (1) triangle setup, (2) triangle traversal, (3) pixel shading, and (4) merging.

1. The *triangle setup stage* first tessellates all geometry into triangles and determines the colors and other vertex parameters of those pixels that are touched by those triangles (Segal & Akeley, 2022). In some contemporary systems, the triangle setup stage only determines the edge equations for these pixels, leaving the next stage to perform the actual interpolations (Akenine-Möller et al., 2008).

2. Next, interpolation is used between each triangle's bounding edges to fill in the interior during the *triangle traversal stage*. All of the properties applied to a triangle's defining vertices (e.g., color, depth, texture, perspective &c.) are interpolated at once to produce a sequence of samples, named *fragments*, which

correspond to the set of pixels bounded by the triangle (Ebert et al., 2003).

3. During the *pixel shading stage*, illumination for each fragment is calculated. This results in a better-quality image than one shaded using interpolation. Many modern video games and graphics frameworks use this technique for increased realism and level of detail. Apart from interpolations, several other operations are performed at this stage, including texture mapping, per-pixel lighting, and reflections (Segal & Akeley, 2022).

4. The final *merging stage* is used to perform the final rasterization steps on the fragment data, such as blending the color of overlapping fragments for transparency or anti-aliasing effects (Akenine-Möller et al., 2008). This stage also determines which fragments are *occluded*. Fragments are occluded when they share view space coordinates with other fragments. An occluded fragment may be replaced or blended with the overlapping fragment or discarded (Segal & Akeley, 2022). Finally, the fragments are *merged* into *pixels*, and this pixel data is copied into the framebuffer.

Once the final stage is complete, the framebuffer is said to be *filled* with pixels, and the rendering is complete.

4.9. Programmable Pipelines

Fixed-function pipelines were common in the early days of GPU graphics but are rare in today's graphics substrates. They are constrained precisely as the name suggests: pipeline functionality is fixed and cannot be modified. For example, suppose the pipeline supports a set of methods to rasterize geometry and shade pixels once commands are committed to the pipeline. In that case, they will be executed in

order without further input from the application. The programmer cannot add operations or alter data once submitted (Bailey & Cunningham, 2009). Fixed-function pipelines are limited in their capabilities but are easy to design and use, especially in low-performance IoT (Internet-of-Things) devices. Although fixed-function pipelines are sufficient to render basic scenes, their inflexibility exposes severe deficiencies, especially when input data may change unexpectedly, as necessitated by exploratory graph visualizations.

In modern GPUs, all the pipelines are *programmable* (or *generic*), replacing certain shading stages of each pipeline with programmable *shader units*. In many cases, code developed to run on fixed-function pipelines can still run on programmable pipelines either by issuing commands making the programmable stages optional or through an intermediate compatibility framework that emulates the older, fixed-function pipeline (Segal & Akeley, 2022). The basic functionality of the pipelines remains the same: vertex mapping, tessellation, pixel color calculations, &c. are all supported as before, but the programmer now has control over whether and how these stages are executed. The programs that perform programmable-pipeline calculations are called *shaders* (Section 4.9.1) because they were initially created to perform customized "shading" (e.g., coloring, lighting, and blending) operations on 3D (three-dimensional) geometry, but have since evolved to perform many other tasks (Akenine-Möller et al., 2008).

4.9.1 Shaders

Shaders are special programs that run directly on GPUs, similar to the procedural programs even novice programmers are familiar with. Each shader has one or more entry points (akin to a `main()` function), static global constants (known as *uniforms*), functions, variables, and return values (Ebert et al., 2003). Due to the

vectorized nature of GPUs, shaders diverge from general-purpose programs in their scope and range of effect (Ebert et al., 2003; Marshall, 2004).

CPUs and GPUs are designed with very different goals: CPUs are designed to provide high performance on general-purpose *sequential* programs. Parallel programming on CPUs is complicated because most algorithms designed for CPU execution have data dependencies requiring communication and synchronization between processors to function (Ebert et al., 2003). Resource availability, context switching, and lock mitigation strategies further complicate parallel programming and tend to reduce overall processor utilization on the few CPU cores available (Ebert et al., 2003).

In contrast, GPUs are not serial processors but are *stream processors* organized around two concepts: *specialization* and *parallelization* (Ebert et al., 2003; Buck & Purcell, 2004). A stream processor works by executing a *kernel function* (e.g., a fragment program) on a *stream* of pipeline-specific input records (e.g., fragments) and producing a set of output records (e.g., shaded pixels) in parallel. Each input record passed to the kernel function is processed *independently*, and no data dependencies exist between them. GPUs have hundreds or thousands of compute cores working in concert and lack the extensive hardware support needed for locking and synchronization. Data independence permits the architecture to execute the kernel in parallel in any order without exposing any parallel constructs to the programmer, and no inter-core communication is required (Buck & Purcell, 2004).

Generally, there are four types of shaders defined for GPUs: (1) vertex shaders, (2) fragment shaders, (3) geometry shaders, and (4) compute shaders:

1. *Vertex shaders* perform calculations on *individual* vertices. Vertex shaders compute or modify the properties of their assigned vertex, such as the position in the n-dimensional model space, such as color, blending, clipping, position, or normal vector. These shaders have no access to neighboring vertices and have no

way of accessing them. Vertex shaders may also abort programming or discard vertices based on specified criteria. Vertex shaders may replace or supplement the vertex shading stage of a fixed geometry pipeline (Section 4.8.1).

2. *Fragment shaders* calculate color values for *individual* pixels. These shaders interpolate between a set of vertex properties defined by previously run vertex shaders to compute the color of their assigned fragment. This can mean linearly-interpolating color values between two vertices, or mapping texture coordinates to a polygon's vertices. Like vertex shaders, fragment shaders cannot operate on neighboring fragments and have no access to other fragments' properties or values. Fragment shaders may also choose to discard fragments if their values exceed some threshold, fall outside of the clipping area, or are transparent. All concurrent fragment shaders share a single program counter and operate in a synchronous, lock-step manner, known as SIMT (Single-Instruction, Multiple Thread). Fragment shaders may replace or supplement the fragment shading stage of a fixed pixel pipeline (Section 4.8.2).

3. *Geometry Shaders* generate *new* geometry from individual vertices output from vertex shaders. They can be used to generate simple geometry (e.g., spheres) from a single vertex or to create realistic 3D textures using procedural "noise" to roughen a smooth surface. Where available, geometry shaders are executed between the model-view transformation and vertex mapping stages of the fixed geometry pipeline (Section 4.8.1), as described above. A geometry shader may generate hundreds or thousands of new vertices from a single vertex, which are subsequently passed back to vertex and fragment shaders for rendering. Because of this performance degrading *data amplification*, geometry shaders are not well-supported in most platforms. Instead, much of this functionality

has been transferred to *compute shaders*, defined below.

4. *Compute Shaders*, also known as *kernels*, are *general-purpose* programs that are far more flexible than any of the other shader types and are neither limited to operating on single inputs nor generating single outputs. Instead, they may access any bound GPU memory and support limited synchronization and communication via shared L1 and L2 caches. Compute shaders execute exclusively through distinct *compute pipelines*, which offer no fixed functionality, and each thread has an independent program counter. Due to this flexibility, compute pipelines cannot interleave operations with geometry or pixel pipelines. However, if properly formatted, the output of a compute pipeline can be used as the input of the pipelines, and the output of a render pipeline can be passed back into a compute pipeline for further processing. In Section 4.10, we explore compute shaders in depth.

4.9.2 Shading Languages

Unlike fragment or vertex shader programs, GPGPU (General-Purpose Graphics Processing Unit) kernels are intended to run on heterogeneous substrates, using all available resources and executing on both CPUs and GPUs concurrently. Programmers concerned about performance on a particular substrate must understand these resource limits and the performance characteristics of that hardware. Computational substrates have vendor-specific limits on resources of various types: memory for storing instructions, registers for storing temporary variables, vertex-to-fragment interpolants, texture units, texture memory, and internal framebuffers. Most GPU vendors support one or more low-level programming interfaces for programming shaders, generally at the ASM (Assembly Language) level, but do not fully document implementation details. To exploit the execution substrates's potential using ASM, the

programmer must employ proprietary design information that vendors are unwilling to disclose publicly (Marshall, 2004). Given access, a clever programmer or algorithm designer can use vendor-specific ASM code interfaces to direct hardware components to perform novel tasks. However, we will not be discussing such interfaces here. Instead, we will focus on high-level GPGPU programming languages that provide some degree of hardware independence, virtualizing the hardware behind a HAL (Hardware Abstraction Layer) and selectively exposing the limits described above through a unified interface regardless of the underlying the substrate (Ebert et al., 2003).

High-level languages have the potential to provide better performance than typical handwritten ASM code. Like high-level programs written in C for CPUs, these languages must be compiled into substrate-specific ASM before being executed. The compiler can optimize shaders using detailed information about the substrate that would be too tedious to exploit when writing ASM code by hand. Even with proprietary information, small code changes may cause nonlinear performance impacts that are specifically large to be of concern to the programmer; adding a single ASM statement to a shader or kernel can cause its performance to drop by half on some GPUs. In general, GPUs can perform sets of four arithmetic instructions at the same time they perform a set of one, two, or three; rewriting code to benefit from optimizations like these can be a tedious and error-prone task for the programmer. Compilers, however, analyze and organize source code into *idioms* that recognize and exploit these optimization opportunities automatically (Marshall, 2004). High-level languages also make it easy for the programmer to create "libraries" of shaders organized by function. More importantly, shaders composed of library functions can be easily modified or combined to meet specific needs (Ebert et al., 2003).

4.10. General-Purpose GPU Computing (GPGPU)

Unlike CPUs, millions of concurrent threads can be launched on modern GPUs; GPU parallelism is limited only by the total number of cores and memory access times (Jeowicz et al., 2013). For example, per-pixel lighting is a highly parallel and data-intensive task. Because pixel processing is independent, all pixels can be processed in parallel: each pixel mapped onto a single thread is processed in O(1) constant time. The benefits of extending this data parallelism beyond image processing and computer graphics to general-purpose computing are obvious (Jeowicz et al., 2013).

GPGPU platforms provide transparent, efficient access to all resources available in a heterogeneous substrate, allowing code to run on the CPU, GPU, or both concurrently (Jeowicz et al., 2013). However, easy inter-architectural data movement comes at a substantial communication cost (Section 4.10.1). Even so, many high-performance algebra frameworks today, such as Matlab (The Mathworks, Inc., 2022) and CU-BLAS (CUDA Basic Linear Algebra Subprograms) (NVIDIA Corp., 2022a) harness GPGPU processing power to perform data-intensive mathematical operations. For example, matrix multiplication has a computational complexity of $O(n^3)$, or more specifically, to multiply two square $n \times n$ matrices, n^3 multiplications and $(n-1)n^2$ additions must be performed. That is, the entry $\mathbf{c}_{ij}$ of the product is obtained by multiplying term-by-term the entries of the ith row of $\mathbf{A}$ and the jth column of $\mathbf{B}$ and summing these n products. This operation is ripe for parallelization, and many GPGPU-optimized matrix multiplication algorithms are documented in the literature (Li et al., 2013; Dalton et al., 2015; Kelefouras et al., 2016).

GPGPU platforms are not limited to operating on linear data but are flexible enough to work with structured data and irregular workloads graph layout optimization tasks entail (Wang et al., 2017). Fast solutions of BFS (Breadth-First Search),

SSSP (Single Source Shortest Path), and APSP (All Pairs Shortest Path) are also well-documented (Harish & Narayanan, 2007; Zhong & He, 2014; Wang et al., 2017; Lin & Huang, 2021). However, random data access patterns and conditional control flow requirements of graph layout algorithms continue to pose significant challenges to developing flexible high-performance graph frameworks (Wang et al., 2017).

There are now many GPGPU programming frameworks out in the wild: CUDA (NVIDIA Corp., 2022b), Vulkan (Khronos® Group, 2022b), OpenCL (Khronos® Group, 2022a), and Metal (Apple Inc., 2022a). Henceforth, we use the Vulkan terminology because Vulkan is a well-supported *cross-vendor* framework descended from OpenGL.

4.10.1 GPGPU Memory & Concurrency

Like the GPU memory model, the GPGPU concurrency model is hierarchical. GPGPU programs are called *compute shaders* or *kernels* (Section 4.9.1). Modern GPUs support the simultaneous execution of heterogeneous kernels from different queues within the same or even different applications (Duţu et al., 2020). Compute shader execution generally adheres to the SPMD (Single-Program, Multiple Data) model: a single *entry point* function is executed from a memory-backed queue by many *compute invocations* called *threads* (Duţu et al., 2020). Threads are dispatched into *workgroups*; workgroup size is programmer-specified, up to the maximum number of compute cores available. Workgroups are further partitioned into *thread groups*. Thread group size is substrate-dependent but ranges between eight and sixty-four. Each thread has a workgroup id, subgroup id, and a unique, global thread id, positioning the workgroup into an execution "cube," allowing each thread to identify the data to act upon. Workgroups are often mapped to the same vector processing unit, in which case they can communicate via the shared L2 cache (Figure 2.6b). Threads

within each thread group are generally mapped to the same compute core and may synchronize and communicate via the shared L1 cache.

Within a thread group, execution generally follows the SPMD model. However, because compute shaders may evaluate *branch conditionals* on thread ids and threads do not share program counters, they can execute as MIMD (Multiple-Instruction, Multiple Data) programs with trivial modifications. MIMD thread groups with control flow divergence must be *sequentialized* and may require expensive synchronization methods to minimize resource contention (Sorensen et al., 2021).

Inter-workgroup execution is a critical concern, but only scant documentation for inter-workgroup synchronization exists. In particular, *independent forward progress* between threads of execution is not always guaranteed on a given GPU (Sorensen et al., 2021; Duţu et al., 2020). Many GPGPU framework specifications do not commit to any progress guarantees (Sorensen et al., 2021). However, GPU programmers can assume *some* level of forward progress support: though undocumented, all current GPUs support a limited form of forward progress known as the OBE (Object-Bound Execution) model, which states that any workgroup that starts execution will be fairly scheduled until it finishes execution (Sorensen et al., 2016, 2019).

If the programmer creates too many workgroups or sizes workgroups incorrectly, they may not all execute concurrently (Sorensen et al., 2021). *Barrier synchronization* across all workgroups will hang due to a starvation cycle on a GPU if executed with too many workgroups (Sorensen et al., 2021). Therefore, GPUs do not have primitive mutexes (Sorensen et al., 2021). Novel cross-vendor atomic lock-and-set operations or other inter-workgroup barrier primitives have been invented but are not yet standardized (Sorensen et al., 2016, 2021). This means that it is generally unsafe to write GPU programs where one thread relies on another thread making

progress (Sorensen et al., 2021). If thread X waits for another thread Y, X might wait indefinitely, causing a *starvation cycle* and failure to terminate (Sorensen et al., 2021). This can lead to the classic "dining philosophers" problem where each thread waits for the other, a cycle known as a *livelock*.

To avoid these problems, well-written GPGPU programs will prioritize local interactions (e.g., subgroup-level synchronization) over global interactions (e.g., inter-workgroup synchronization) and exploit cache-conscious memory layouts (Section 2.6.1) to avoid stalling out. If compute shaders are written correctly, adverse locking behaviors are extremely rare unless executed in a noisy environment, e.g., in the presence of other threads that cause *memory stress* by repeatedly accessing memory using irregular access patterns that cause cache contention (Sorensen & Donaldson, 2016) or execution substrates perform preemptive scheduling at the kernel level (Duţu et al., 2020).

4.11. GPGPU Frameworks

OpenGL and other graphics APIs are relatively low-level and were originally designed to implement 3D applications. Implementing graph layout algorithms or other general-purpose computing tasks directly using these APIs is an awkward task; graphics shader units must be repurposed to perform tasks they were not designed to execute. CUDA (NVIDIA Corp., 2022b) and OpenCL (Open Compute Language) (Khronos® Group, 2022a) are general-purpose frameworks modeled on OpenGL to provide heterogeneous GPGPU computation in C/C++ with other lower- or higher-level APIs. Code using these frameworks is written once in C or C++ and then cross-compiled or *transpiled* into CPU *and* GPU-specific ASM code simultaneously. Traditional C compilers, such as GCC (GNU Compiler Collection) or Clang/LLVM (Low-Level Virtual Machine), can compile the CPU code, but the GPU code needs a

particular compiler that understands the substrate-specific interfaces required. Along with CUDA and OpenCL, several other high-level GPGPU languages and graphics frameworks are available. However, our requirement for system-agnosticism using Javascript renders any language requiring a separate compiler unsuitable for Dynamical.JS.

4.12. Web Graphics

4.12.1 WebGL & GLSL

As mentioned in Section 1.4, other graph layout frameworks have been implemented in Javascript for execution on the web, and many algorithms have been implemented using WebGL and GLSL (OpenGL Shading Language) (Baldwin & Rost, 2019a). However, GPU access in WebGL and GLSL is limited to vertex and fragment shaders (Khronos® Group, 2014; Baldwin & Rost, 2019b); both require several workarounds and complicated pipeline pathways to emulate access to node and edge data. WebGL has no support for GLSL geometry shaders, which would be necessary to implement the most advanced placement algorithms and to support adding large numbers of new graph elements efficiently. This limitation has been explored thoroughly in the literature, and several complex algorithms have been efficiently ported to GLSL, using texture memory to store vertex data and copying that data into new vertex buffers for subsequent processing (Bleiweiss, 2008; Hadlak et al., 2011; Silva et al., 2013). The published algorithms are completely synchronous because WebGL access is only allowed from the main processing thread of any agent's web session. This is sufficient for the smooth drawing of individual graph layouts but not for background placement or animation of large graphs. Further, the complicated data pathways involved conflict with several of the key requirements of this thesis,

and therefore WebGL and GLSL are not used as a basis for Dynamical.JS.

4.12.2 GPU for the Web (WebGPU)

Like other GPGPU frameworks, WebGPU provides a standard interface to the computation facilities present in both operating systems and their underlying substrates. The WebGPU framework (World Wide Web Consortium, 2022b) is layered upon native graphics frameworks such as Direct3D 12 from Microsoft, Metal from Apple, and Vulkan from the Khronos Group (World Wide Web Consortium, 2022a). The WebGPU API is implemented as a native Javascript Web API that exposes these technologies in a performant, robust, and safe manner to web-based applications. WebGPU is on track to become a W3C standard, and reference implementations have been created for all major web browsers and server-side Javascript interpreters, such as node.js.

Using WebGPU and WGSL (Dakkak et al., 2016), many placement algorithms that require geometry shaders and multithreaded execution can be implemented directly on the GPU (Bleiweiss, 2008; Udupa et al., 2009; Jeowicz et al., 2013). Unlike WebGL, WebGPU is completely asynchronous and can be used by any Web Worker, allowing for multithreaded execution using background Workers. Using textures and wired memory buffers to store nodes, edges, and attributes in high-speed memory, at the cost of a single memory map operation, Dynamical.JS can perform cache-conscious and highly parallel implementations of many graph layout algorithms efficiently enough to allow online exploration and smooth animation (Udupa et al., 2009; Hadlak et al., 2011; Jeowicz et al., 2013; Silva et al., 2013; Sheng et al., 2019; Tan et al., 2020) within the Javascript runtime.

Algorithms already ported to GPGPU frameworks or WebGL are trivially ported to WebGPU as-is; WGSL provides a C-style coding interface familiar to most

GPGPU and shading languages. The most significant development challenge is the decomposition and translation of various serialized, CPU-bound algorithms to use WebGPU and WGSL compute shaders for placement and animation. This will limit the algorithms which could be implemented during the writing of this thesis. However, because many graph layout algorithms follow a well-defined evolutionary lineage, most can be decomposed into constituent parts and commonalities can be quickly abstracted.

WebGPU is still a nascent technology with a severe downside: the API standard and reference implementations are not yet complete. The API has evolved in unexpected and code-breaking ways since Dynamical.JS was conceived. This, unfortunately, delayed and stalled the development of critical components of Dynamical.JS, and several desired tasks could not be completed by the time this thesis was written. We intend to complete these tasks and reincorporate the missing functionality in a future iteration of this project.

Chapter V.

Implementation & Results

The Dynamical.JS framework will be published as a consolidated, minified Javascript module and a complete source tree, where its algorithms and methods can be scrutinized. The project will also be released under a yet-to-be-determined open-source license to facilitate ongoing maintenance and integration into future academic or commercial products. Akin to other visualization frameworks, Dynamical.JS will be extensible through the development and use of additional modules, which could be used to modify the framework at any operational phase. For example, new layout plugins would allow for novel placement strategies for extant algorithms or to facilitate the development of new algorithms. Coarsification plugins to optimize the underlying data structures, facilitate new caching strategies, or enable novel motif simplification types. Analytic modules could be loaded to tune the system or substrate for platform resource usages, such as on small-screen IoT (Internet-of-Things) devices or ultra-wide workstation configurations.

5.1. Code Syntax & Documentation

Javascript code for Dynamical.JS is documented using a JSDoc-like syntax (Mathews, 2011), which allows the automatic generation of HTML (HyperText Markup Language) or PDF (Portable Document Format) documentation output directly from the source code. JSDoc tags also facilitate type-checking and debugging in the IDE (Integrated Development Environment). However, JSDoc is not used during development, and the final deliverable will not contain generated documentation until published widely on the internet. The JSDoc comments will only be used to facilitate type-checking by the development IDE until the final

publication of the Dynamical.JS framework. The JSDoc comments will remain in the uncompressed source code but will be removed from minified versions of the library used for publication. Per standard industry practices, the Javascript code is written in *self-documenting* style.

5.2. Environment

As shown in Listing 5.1, the development and test "rig" is a simple one: an HTML5 (HyperText Markup Language, Version 5) (Web Hypertext Application Technology Working Group, 2021c) page, which embeds either the entire Dynamical.JS framework or specific testing components as Javascript modules, and contains a single `HTMLCanvas` object and a `div` to display error messages.

Listing 5.1. The HTML development "rig" for Dynamical.JS

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>DG Basic Test</title>
<link rel="icon" type="image/x-icon" href="data:," />
</head>
<body>
<!-- the graphical context where graphs will be drawn. -->
<canvas id="gfx"></canvas>
<div id="error"></div>
<!-- The Dynamical.js module -->
<script type="module" src="dynamical.js"></script>
<!-- The testing module -->
<script type="module" src="basic.test.js"></script>
</body>
</html>
```

Unit and component tests are created by sub-classing the `DGTestRig` and

`DGPerformanceRig` classes, and defining `static async test*()` methods containing the functionality to test. A complete listing of the abstract `DGTestRig` is shown in Listing C.2 and an example test is shown in Listing C.3. The Dynamical.JS includes a set of test directories within each package and sub-package as described below (Section 5.3).

5.3. Code organization

The Dynamical.JS code base is organized into four primary component packages: (1) `data`, (2) `layout`, (3) `drawing`, and (4) `common`. The first three correspond to the modules described in Section 2.3 and the `common` package, including shared code and other utility methods.

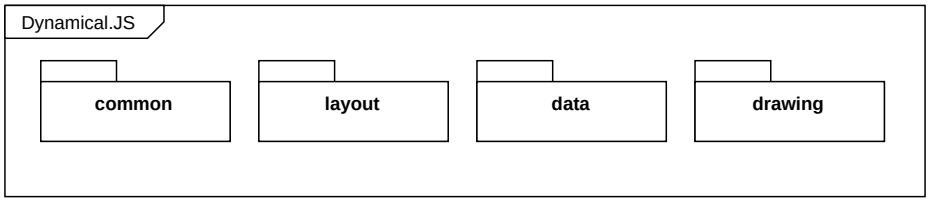


Figure 5.1. A high-level UML (Unified Modeling Language) package diagram of the Dynamical.JS framework.

While the compiled framework is *logically* organized into these three main packages, source code for several shared objects (including the testing/development module code) is contained within the `common` directory tree.

The `common` package includes many useful sub-packages, encapsulating various ADTs (Abstract Data Types), matrix and vector math, pseudorandom number generation, error handling, GPU (Graphics Processing Unit) manipulation functions, as well as testing, performance, and utility methods.

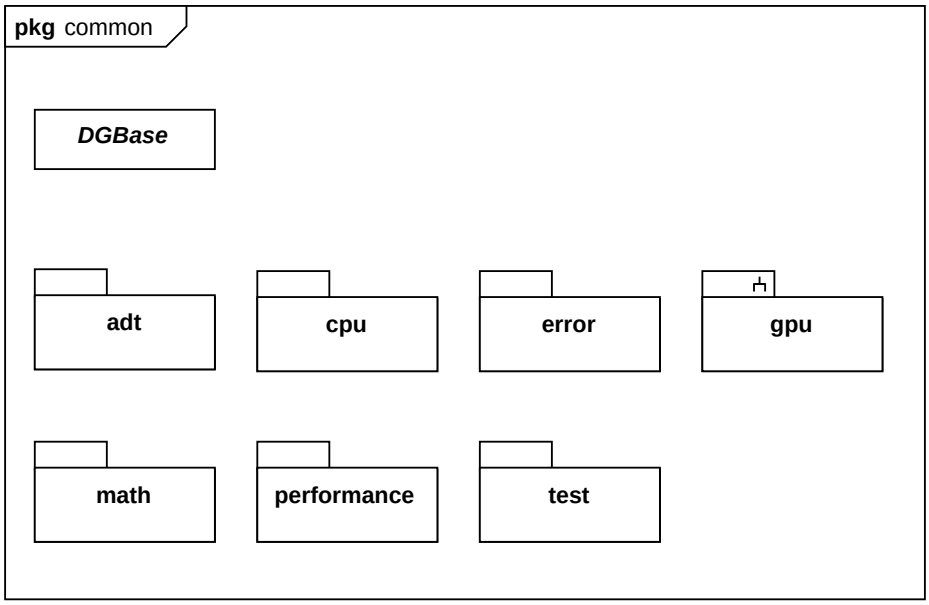


Figure 5.2. A UML package diagram for the Dynamical.JS `common` package.

5.4. The `DGBase` Object

Most objects defined by Dynamical.JS are implemented as concrete subclasses of the abstract `DGBase` object, as shown in Listing C.1. `DGBase` provides type introspection, equality checking, and interface mixin capabilities. `DGBase` also extends the Javascript `EventTarget` class to support event handling. Dynamical.JS also implements several abstract `DG*Base` subclasses and interfaces that enable extensibility (Section 5.4.1) via the *class cluster* and *factory* design patterns (Gamma et al., 1995). Application developers may extend the framework by defining new concrete subclasses or simply redefining existing classes' methods.

5.4.1 Extensibility

Extensibility is a *key quality* of any exploratory graph visualization framework; Dynamical.JS supports extensibility throughout. Class cluster factory classes extend or mix-in the `DGExtensibleBase` plugin interface:

Listing 5.2. The `DGExtensibleBase` interface.

```
class AbstractExtensibleBase extends ... {
    /**
    * A registry of valid class instances.
    * @type {Map<class>}
    */
    static #registry = new Map();

    /**
    * Return an array of registered class names.
    * @type {Array<string>}
    */
    static get registeredClasses() {
        return Array.from( AbstractExtensibleBase.#registry.keys() );
    }

    /**
    * Register a class with the registry.
    * @param {string} key - A string uniquely identifying a class.
    * @param {value} key - The class prototype to register.
    */
    static _registerClass( key, value ) {
        AbstractExtensibleBase.#registry.set( key, value );
    }
}
```

A class that wants to register with the class cluster must register its prototype during static initialization:

Listing 5.3. Concrete subclass registering with a class cluster.

```
export class ConcreteSubclass extends AbstractExtensibleBase {
    static #init = ( () => {
        super._registerClass( 'className', ConcreteSubclass.prototype );
    } )();
}
```

As soon as the class is imported as a static module or linked dynamically via `require()`, the class will be added to the class cluster from any code which has included Dynamical.JS and can be instantiated synchronously via its constructor or asynchronously via the parent cluster's `async build()` method (Section 5.5.1).

5.5. Asynchronous Interfaces

Dynamical.JS implements several asynchronous design patterns as object interfaces and prototype extensions to specific classes.

5.5.1 Asynchronous Initialization

As mentioned in Section 4.5.2, `async` functions are only executed after the current job queue is empty. It is impossible to extract the result of a promise within a synchronous function, so any object that must be initialized asynchronously cannot be used within a synchronous context. Therefore, calling `async` methods from object constructors will not yield usefully instantiated objects. For example, most objects requiring access to the GPU need significant time to initialize as resources are gathered. Further, the WebGPU (GPU Computing for the Web) framework is designed with asynchronous operation in mind; therefore, any object requiring GPU resources, such as `GPUDevice`, `GPUBuffer`, and `GPUShader`, *must* be initialized asynchronously, and only be accessed within purely asynchronous contexts.

Dynamical.JS supports asynchronous initialization through two methods built into `DGBase`: `init()` and `build()`, which allow the programmer to "chain" operations to these newly initialized objects once they have been initialized. Listing C.1 and Listing C.10 demonstrate the asynchronous initialization chain for the `DGBase` and `DGWebGPUBase` objects, and Listing 5.4 shows their usage in an example application.

Listing 5.4. Example code demonstrating asynchronous initialization.

```
(async () => {
    // Initialization options
    const options = { a:true, b:false };

    // Allocate an uninitialized class
    const first = new DGBaseSubclass();

    // wait for the object to initialize,
    // then perform some activities
    await first.init( options )
               .then( () => { /* do something */ } )
               .then( () => { /* do something else */ } )

    // A class which uses the build interface to
    // allocate and initialize a particular class,
    // but does not wait for initialization to occur.
    // The initialization of second will not occur until after
    // the enclosing function has completed.
    const second = DGBaseSubclass.build( options )
                        .then( () => { /* do something */ } )
                        .then( () => { /* do something else */ } )

    // This code runs before the second class has been initialized
    doSomethingElse();
})();
```

5.5.2 Asynchronous Loops

Looping is a critical operation in graph layout algorithms and consumes most of the computation time. Many algorithms partition a graph into several subgraphs, which may then be assigned to independent threads. However, as implied in Section 4.5.2, concurrent loop execution is not possible in Javascript but is available in WebGPU. To emulate this behavior in Javascript without locking up the UI (User

Interface), Dynamical.JS defines the `DGAsyncLoop` Interface:

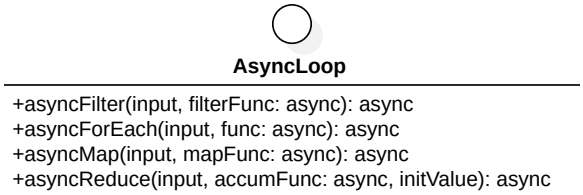


Figure 5.3. A UML class diagram of the Dynamical.JS `DGAsyncLoop` interface.

Using the `DGAsyncLoop` interface, as shown in Listing C.4, we can easily segment and parallel-process the `TypedArray` objects and `GPUBuffers` used as the base for graph *materialization* in Javascript, as discussed in Section 5.8.1. The `DGAsyncLoop` interface has been mixed into Javascript built-in `Array` class and is used extensively to extend asynchronous execution support to create async generator functions defined in `common/util/util.js`.

5.6. Mathematical Operations

Mathematical operations are critical for both graph layout and graph rendering operations. The most important mathematical operations required for these operations are random number generation, affine transformation, and statistical methods. To facilitate the design goal of self-sufficiency, we have implemented many of these operations. However, Dynamical.JS should not be considered a mathematical package. Due to time constraints, only those operations directly required for this project have been implemented.

5.6.1 Vectors & Matrices

Dynamical.JS supports a limited set of scalar-vector, vector-vector, matrix-vector, and matrix-matrix operations on vectors of length 2, 3, and 4, and 2×2,

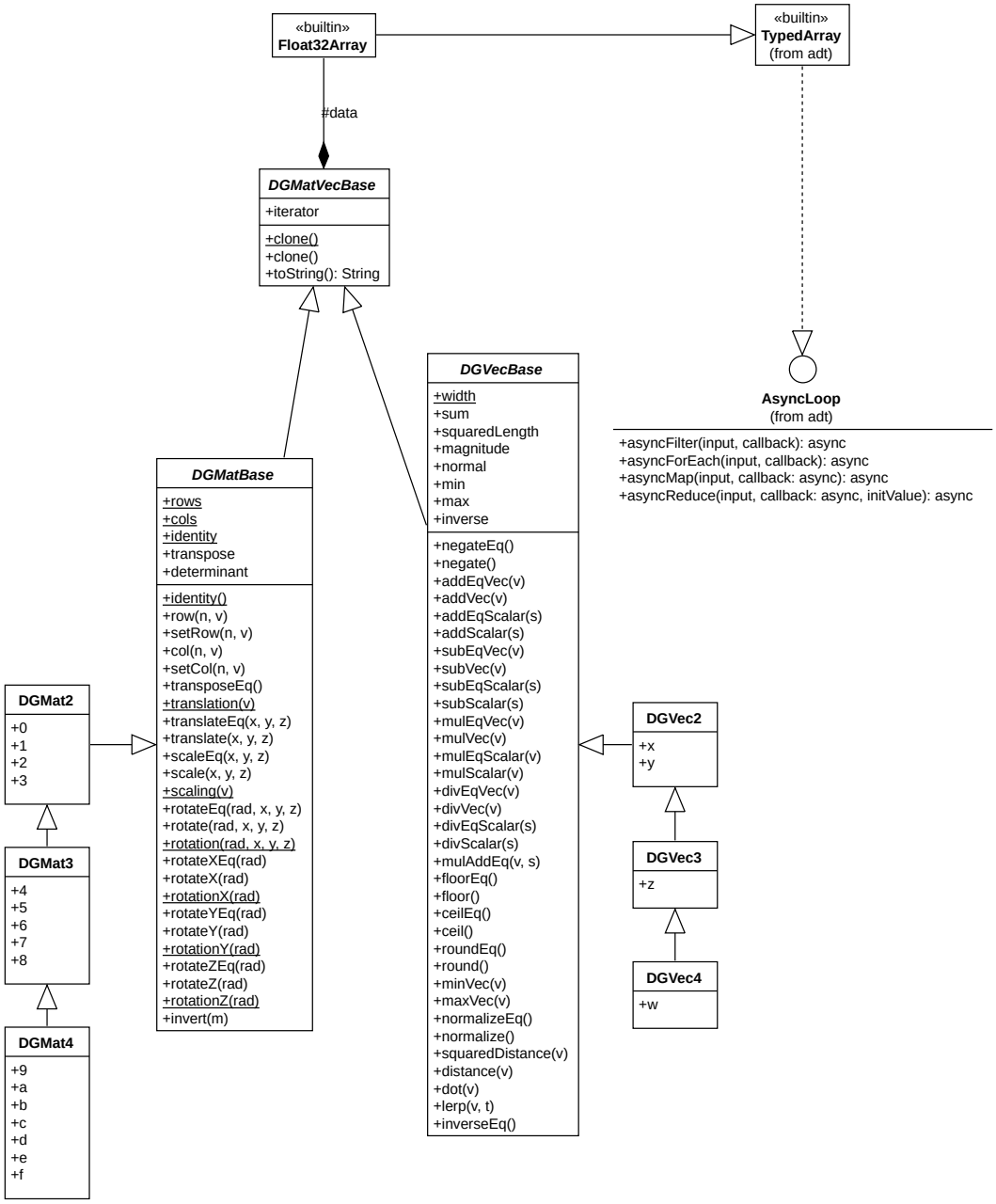


Figure 5.4. A UML class diagram of the Dynamical.JS matrix sub-package.

3×3, and 4×4 matrices to support *affine transformations* required for rendering (Section 2.10.2). The `DGVec*` and `DGMat*` classes extend the Javascript built-in `Uint32Array` class and provide byte compatibility and alignment suitable for WGSL (WebGPU Shading Language) shader access and basic BLAS (Basic Linear Algebra Subprograms) support.

5.6.2 (Pseudo) Random Number Generators (PRNGs)

Graph layout algorithms frequently begin with "random" layouts, which, when used with exploratory graph visualizations of very large graphs, can be highly taxing on the GPU hardware. When discussing terms like "random" and "stochastic," we almost always mean to say "apparently random" or "pseudorandom." True randomness is unusual in computer science and is often undesirable in computer graphics (Ebert et al., 2003). The obvious stochastic source of random numbers, *white noise*, is uniformly distributed without correlation between successive numbers, which is an unattainable goal. Instead, well-designed PRNGs (Pseudo-Random Number Generators) produce fair approximations of white noise. But is white noise necessary? We require functions that are *apparently* random but are, in fact, replicable functions of some dynamic input; truly random functions, such as white noise, don't have these inputs. Effective PRNG primitives will take the current state as its input and will always return the same value given the same state. Looking into the literature on hashing and PRNGs, we can find several ways to convert a set of coordinates into hashed values that can be treated as PRNs (Pseudo-Random Numbers) instead (Smith, 1984; Ebert et al., 2003).

By default, PRNs are generated using the system's built-in random number generation system, usually based on `/dev/random` or `/dev/urandom`, and generate uniform distributions of either unsigned integers or normalized floating point values

in the range $(0, 1)$. All modern CPU-based (Central Processing Unit) substrates offer PRNGs which are usually re-exported via standard interfaces in most CPU-bound platforms, and Javascript is no exception.

Dynamical.JS wraps random number generation in a class cluster based on the `DGRNG` class. By default, this method is based on the Javascript built-in `Math.random()` method, which must be run $|V|$ times across the entire vertex buffer inside the graph's current materialization. To prevent $|V|$ system calls, or switch to the Javascript built-in `Crypto.getRandomValues()`, which generates cryptographically-random values in blocks with sizes up to 64KiB. Because `Crypto` can only generate integer values, to transform these values into normalized floating point values, any optimization generated by batch generation is removed, requiring an additional $O(|V|)$ function calls to convert these values, and $O(|V|)$ space to hold the intermediate unsigned integers which must be copied back into the materialization. Further optimization is needed.

GPUs, as a rule, do *not* have sufficient entropy generation capability to produce random numbers and rarely contain any other PRNG hardware or generalized PRNG functions (Ebert et al., 2003; Couturier, 2014). Most PRNGs require "seed" values which are transformed each time a value is generated. As each of the GPU's ALU (Arithmetic Logic Unit) is independent, sharing this state is impossible. Instead, PRNs must be generated on the CPU and sent to the GPU, potentially stalling the graphics pipeline, or they must be reimplemented on GPUs using parametrically controlled *noise functions* (Ebert et al., 2003) or provided with a workgroup-sized buffer of seed values that can be accessed concurrently (Couturier, 2014; Demchik, 2014). These functions are generally used to add depth or to model fluid dynamics within textures or animation (Ebert et al., 2003; Salmon et al., 2011). However, PRNGs can be well optimized: given the number of GPU threads T, the $O(|V|)$

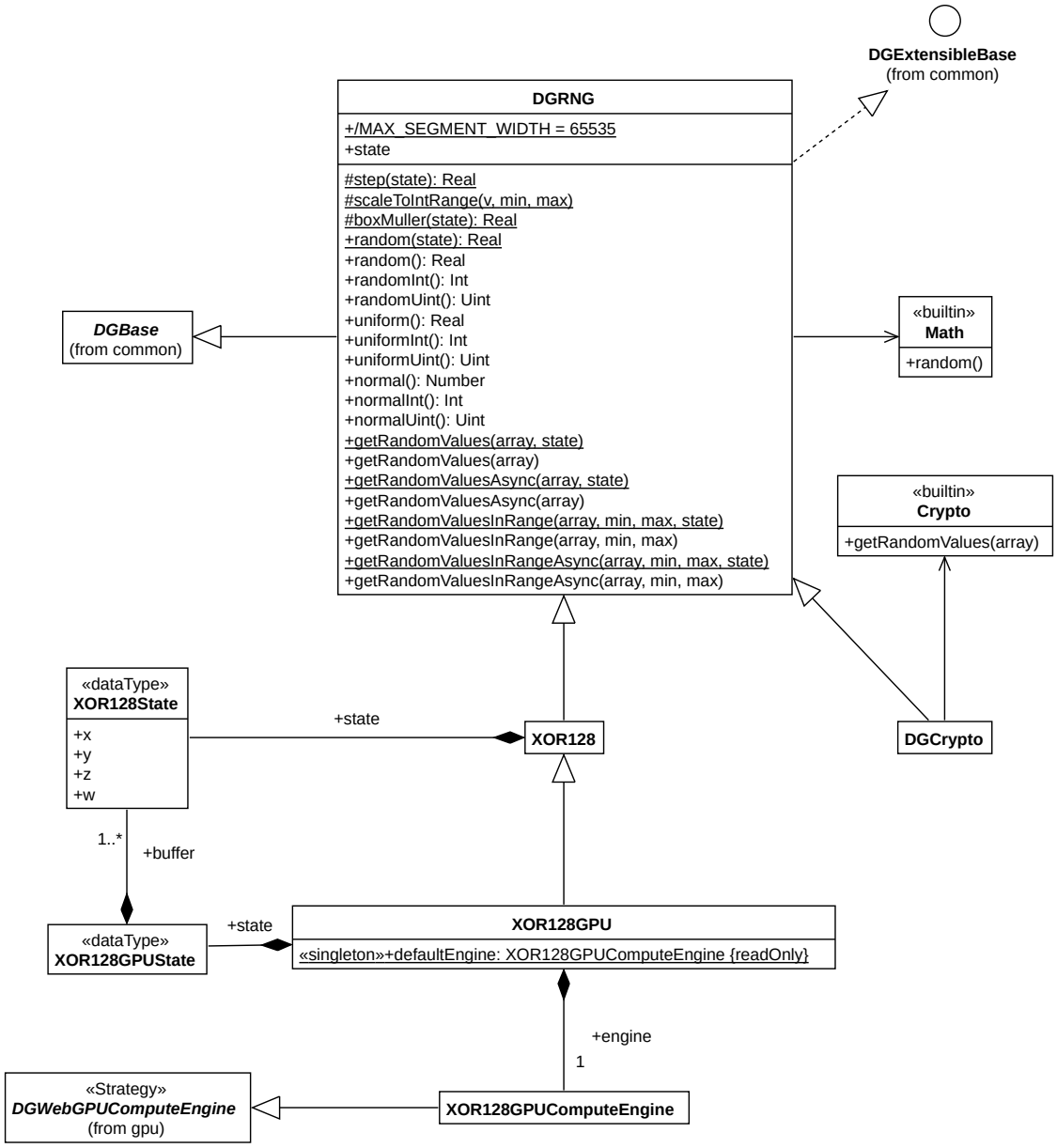


Figure 5.5. A UML class diagram of the Dynamical.JS `random` sub-package.

runtime can be reduced to $O(|V|/T)$ and an additional $O(T)$ storage cost to store the seed values (Couturier, 2014).

To support random number generation that can be replicated on CPUs and GPUs, we have chosen to implement the `XOR128` algorithm from Marsaglia (2003). `XOR128` is extremely fast, generating 220 million PRNs per second, with a period of

$2^{128} - 1$. The `XOR128` PRNG has been implemented under the `DGRNG` class cluster in both Javascript and WebGPU in the concrete `XOR128` and `XOR128GPU` classes, respectively. This algorithm's WGSL implementation is shown in Listing C.12.

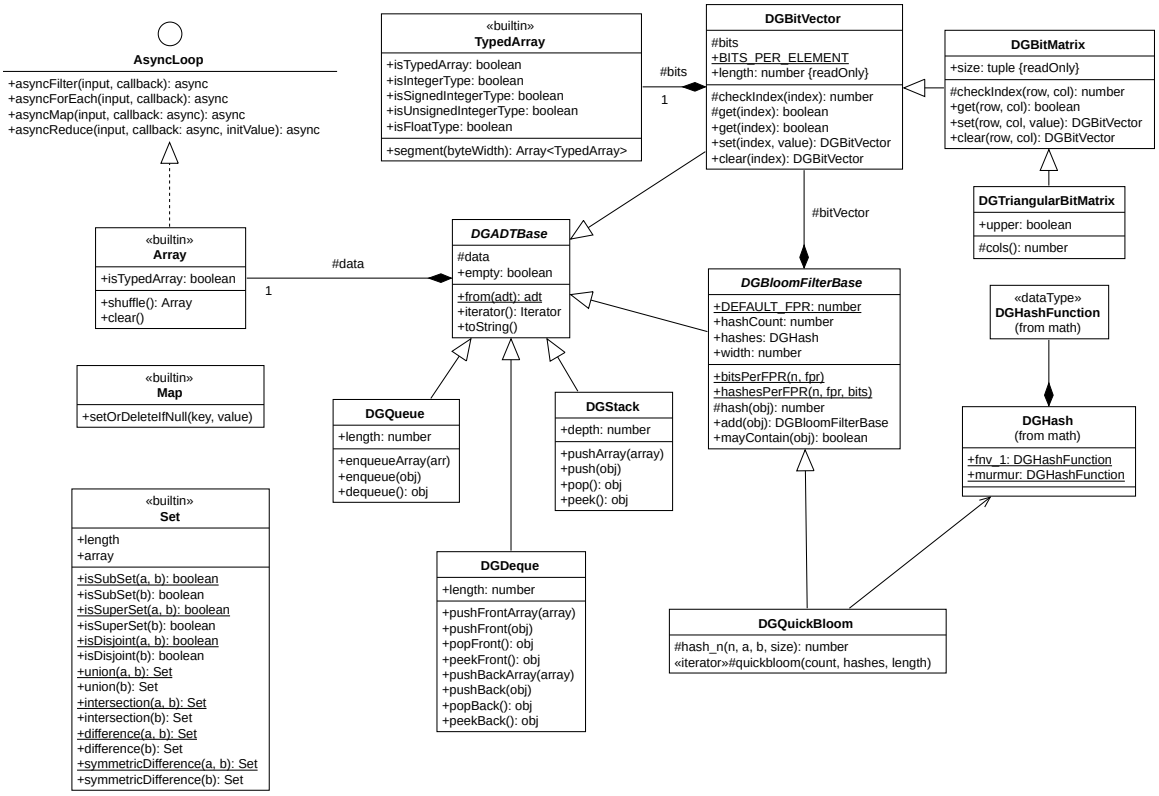


Figure 5.6. A UML class diagram of the Dynamical.JS ADT sub-package.

Graph theory and visualization define several ADTs and operations to organize and manipulate relational, informational data: graphs, nodes, edges, &c. To implement these abstract types in code, we must define data structures to organize our data and allow us to perform calculations using our computational substrate. In Dynamical.JS, these data structures are implemented as concrete subclasses of the abstract classes `DGADTBase` or `DGGraphBase`, which are, in turn, subclasses of the ab-

stract `DGBase` superclass. All `DGGraphBase` classes contain a `Map` (i.e., an associative array) of relevant attributes. Attributes node positions and impose layout constraints during layout, determine the visual marks and channels applied during rendering, and perform arbitrary numerical calculations. Supplementary ADTs have also been implemented to support hashing and mathematical set operations on `Collection`-type objects.

5.7.1 Graph ADT

In computer science and applied mathematics, graphs are pervasive data structures, and algorithms for working with them are fundamental to the field. There are hundreds, if not thousands, of interesting computational problems concerning graphs (Cormen et al., 2009). Because graph theory is a well-documented field, we do not discuss these operations in depth. Instead, we focus only on the aspects apropos to the design and implementation of Dynamical.JS.

For the analyst to glean information from the connectivity therein, they must use graph-theoretical operations to *search* or *traverse* the graph, i.e., systematically follow the edges of the graph to visit the nodes within. Many graph-theoretical operations begin by traversing the input graph to obtain structural information. Low-level operations such as path, loop, or neighborhood discovery, along with higher-level procedures like subgraph building and congruence testing, all require traversing the graph to determine various levels of connectivity. In practice, searches query, scan, or probe the graph's data structure using methods defined in an ADT (Kester et al., 2017). Finding a data structure that optimizes access times and storage requirements during these search operations is an ongoing research subject (Idreos et al., 2019). Graph ADTs can be implemented in several ways, including adjacency lists, adjacency matrices, and incidence matrices, among others, singularly or in combination (Cormen

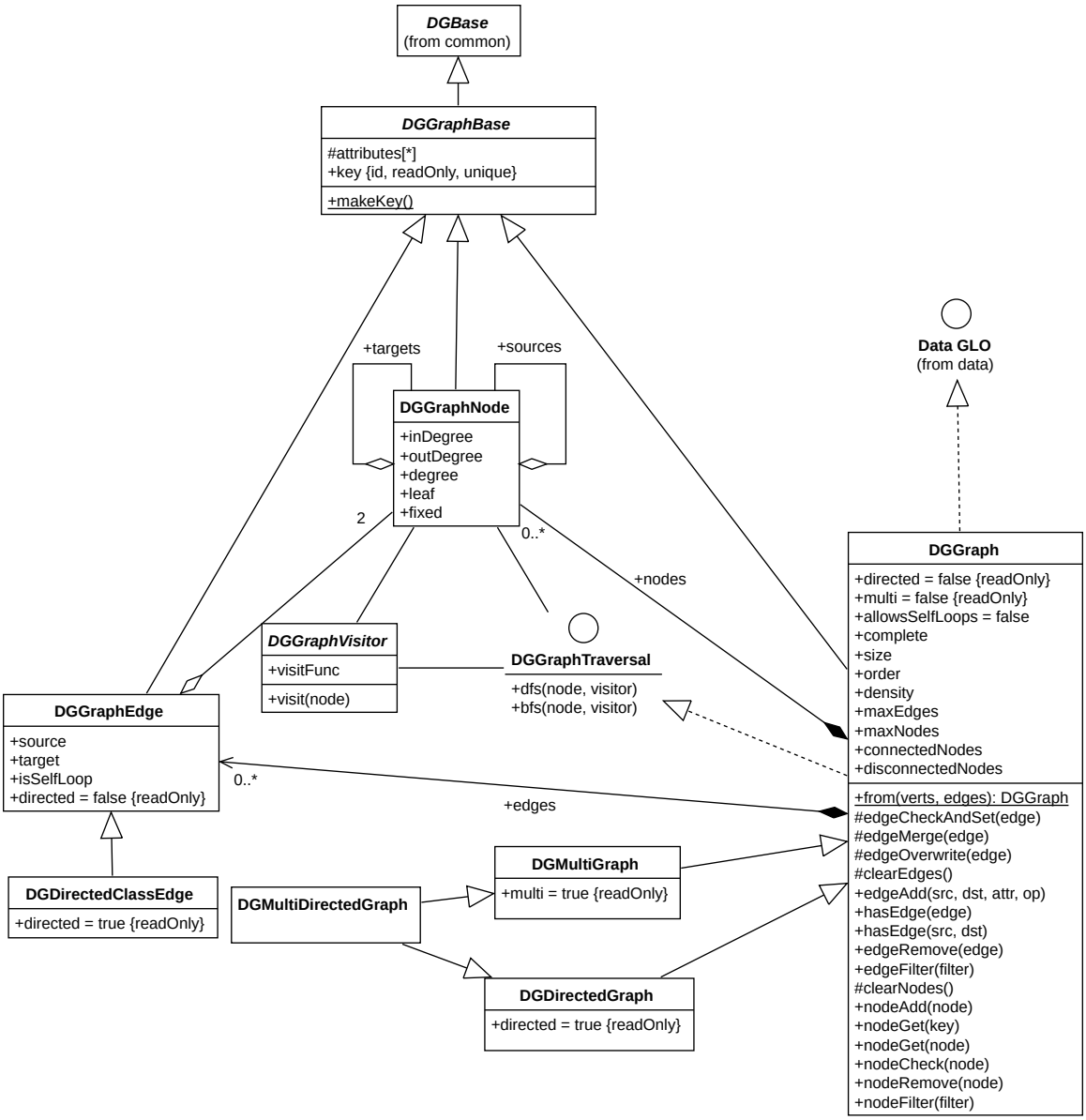


Figure 5.7. A UML class diagram of the Dynamical.JS graph ADT sub-package.

et al., 2009). While MultiMaps (maps of maps) provide the fastest access times for most operations using key-value storage, the memory cost $O(|V| \times |E|)$ is much greater, especially for fully connected or multi-graphs. We will discuss these data structures further in Section 5.8.3.

The Dynamical.JS graph ADT defines several critical operations, such as load-

ing from disk or memory, adding and removing nodes, and querying a graph's contents. Figure 5.7 shows a UML class diagram of the graph ADT sub-package, with auxiliary methods and type specifications omitted for clarity. Dynamical.JS supports all of the graph types described in Section 2.4. These graph types are implemented using the *class cluster* design pattern (Gamma et al., 1995) based on the *concrete* `DGGraph` class. When supplied with appropriate construction parameters, the `DGGraph` class cluster returns an appropriate concrete subclass: `DGDirectedGraph`, `DGMultiGraph`, or `DGDirectedMultiGraph`. By default, all `DGGraph` instances are undirected graphs that disallow self-loops. Future implementations of Dynamical.JS will include support for additional graph types as required.

Listing 5.5. The basic Javascript interface for `DGGraph`

```
class DGGraph extends DGGraphBase {
  get allowsSelfLoops() { return this._attributes[ 'allowsSelfLoops' ] || false; }
  get directed() { return this._attributes[ 'directed' ] || false; }
  get multi() { return this._attributes[ 'multi' ] || false; }
  get edges() { /* return the list of DGGraphEdge instances */}
  get nodes() { /* return this list of DGGraphNode instances */}
  from( V, E ) { /* load the graph from memory/disk */ }
  addVertex( v ) { /* add a single vertex */ }
  addEdge( e ) { /* add a single edge */ }
  removeVertex( v ) { /* remove a vertex */ }
  removeEdge( e ) { /* remove an edge */ }
  isAdjacent( v0, v1 ) { /* return true if adjacent, false otherwise */ }
  neighbors( v ) { /* return an array of neighboring vertices */ }
}
```

5.7.2 Index Structures

Indexes are data structures that organize data records within another data structure to optimize certain search and retrieval operations (Ramakrishnan &

Gehrke, 2000). We disambiguate the alternate plural *indices* to refer to the records contained within a given index data structure.

Bit Vectors & Matrices. `DGBitVector` and its subclass `DGBitMatrix` (Listing C.5) are indexed arrays of m bits where a 1 indicates inclusion in a set and 0 otherwise, and can be interpreted as described as m-bit integers. Dynamical.JS employs `DGBitMatrix` subclasses to implement query result sets, change sets (Section 5.8.5.2), adjacency matrices (Section 5.8.3.2), and bloom filters (Section 5.7.2.2).

Bloom Filters. Bloom filters are simple and space-efficient data structures for fast execution of membership queries on large datasets. A bloom filter for representing a set $S \leftarrow x_1, \ldots, x_n$ of n elements is implemented as a bit vector of length m initialized to 0. When querying set membership, the filter uses a combination of k independent *hash functions* to map each element to a uniform m-bit random number: $\forall x \in S$, bits $h_0(x), \ldots, h_k(x)$ are set to 1. To test set membership for element y, we check if all $h_0(y), \ldots, h_k(y)$ bits are set to 1. If any of these bits are 0, element y is not a member of S. Otherwise, we assume that y is a member of S, with the probability of a *false positive* being $\approx (1 - e^{-kn/m})^k$. Thus, bloom filters can definitively *exclude* elements but may require further queries to determine actual membership (Kirsch & Mitzenmacher, 2006).

The `DGQuickBloom` class (Listing C.8), a concrete subclass of `DGBitMatrix`, implements bloom filters using the Quick Bloom algorithm by Kirsch & Mitzenmacher (2006), using $k = 2$ hash functions, Murmur Hash (Appleby, 2016) and fnv_1 (Fowler et al., 2019).

5.8. Data Module

The Data Module's primary responsibility is the *materialization* `DGGraph` objects and defines several classes and class clusters to do so. The GDOs (Graph Data

Operations) described in Section 2.8 are defined in the `DGGDOInterface`.

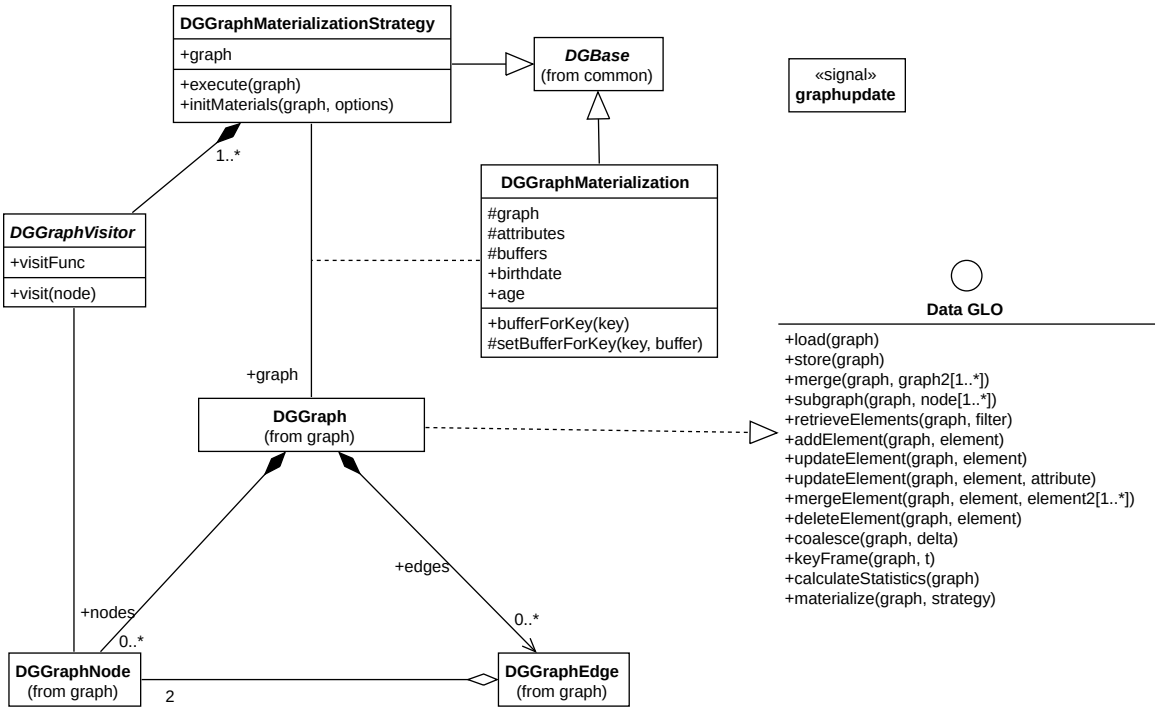


Figure 5.8. A UML class diagram of the Dynamical.JS Data Module.

5.8.1 Materialization

As mentioned above, for a graph to be laid out or rendered into a graphics context efficiently, it must be transformed or *materialized* into an appropriate format. Rather than focusing on the sequencing steps of computation, we focus on manipulating the data structures containing graph elements. Since we require a data-parallel programming model, the challenge is devising a mapping of a naturally unstructured graph into a well-partitioned structured one (Frishman & Tal, 2007). Efficient parallel processing on CPUs and GPUs requires uniformly structured data, such as texture images or matrices. Graphs do not possess a uniform structure; hence, they do not admit any intuitive and natural representation that suits parallel computation (Frishman & Tal, 2007). For example, multi-level graph layout algorithms require

recursively coarsened graphs using the PARTITION or SIMPLIFY GLOs (Graph Layout Operations). In these cases, the materialization strategy must support breaking a large problem into smaller and similarly-sized problems that suit GPU workgroup sizes or other data-parallel programming substrates (Frishman & Tal, 2007). Materializations should be laid out so that a graph layout algorithm may scan through the data without following many pointers that may cause large numbers of cache misses or disk accesses due to virtual memory paging. To alleviate these problems, we transform Javascript's reference-based object representations into cacheable, linear data structures appropriate for data-parallel layout and rendering. These data structures can be reused to prevent unnecessary memory copies between the CPU and GPU or purged when memory pressure requires it. For instance, computation on the nodes within a timeslice can be partitioned across vertices (i.e., materialized nodes). Any updates to neighboring timeslices after identifying common vertices can be similarly partitioned.

Materializations are "frozen" representations of graphs at a particular time as created by the MATERIALIZE GDO (Section 2.8.12) and are generated via *lazy evaluation* before layout commences. Materialization may be as simple as a read-only deep copy of a graph, such as when creating a timeslice or a subgraph of salient elements required by an analytical task. By design, materializations are immutable by the GLOs which operate on them. If required, GLOs (e.g., PARTITION or SIMPLIFY) create new sub-materializations without side effects. This materialization strategy is flexible enough to be expanded by new GLOs, so long as they operate on one or more input materializations and generate one or more output materializations. Computationally expensive layout and rendering operations are performed on materializations asynchronously, permitting the parent graph to update concurrently, fulfilling the project requirements. By separating the graph's materialization from the underlying

graph ADT, we can generate multiple representations of the same graph, either sharing a common materialization or independently generating their own. In addition to presenting the same timeslice with different views, the analyst may also compare distinct timeslices to discover periodic or other temporal patterns by animating only the subgraphs of interest. To support reuse, textures, images, or relational databases may be used to cache serialized materializations to disk or offline storage.

5.8.2 GPU Materialization

Graph layout algorithms often require irregular data access patterns. In addition to exposing sufficient parallelism, materializations benefit from coalesced memory access, effective use of the GPU memory hierarchy, and reduced scattered reads and writes. We manifest these qualities in Dynamical.JS in three ways: First, graphs are stored, whether partitioned or not, so that the nodes in each partition are geometrically close and the number of nodes in each partition is similarly sized. Storing nodes that belong to the same neighborhood within a single partition maximizes memory access locality. Thus, we efficiently use the GPU's memory bandwidth since information regarding neighboring nodes will likely reside in the cache (Frishman & Tal, 2007). Second, since the number of nodes in each partition is roughly equivalent, the computational effort expended per node is balanced. This conforms to the GPU's data-parallel architectural demands, which require lock-step execution. Finally, the position vectors of all nodes are transferred to the global memory once at the start of each iteration. Each workgroup thread reads the position vector of a single node from the global memory to the shared memory of its block and only computes the energy contribution of that node, which has the same index as the thread (Qu et al., 2017).

5.8.3 Materialization Data Structures

Many GPU-enabled graph layout algorithms map intermediate representations of graph elements using two-dimensional data arrays called textures. The challenge is to map the graph and its elements onto these textures, even though graphs do not admit any intuitive and natural representation as balanced arrays (Frishman & Tal, 2007). Proper choice of materialization data structures helps us meet this challenge. The design space for data structures is *vast*; there is currently no optimal way to enumerate all possible designs or predict the performance impact of real-world workloads (Idreos et al., 2019). We do not assert that any single materialization data structure is appropriate for all algorithms; the naive materialization presented below is meant to be a basis for expansion. It simply contains a binary representation of the contents of the graph in a format allowing linear scans of vertex arrays and adjacency lists.

In Dynamical.JS, `DGLayoutEngineBase` (Section 5.9.1) provides a concrete subclass instance of `DGMaterializationStrategyBase` to ensure materializations are created to optimize the data access patterns required by each layout stage, ensuring the CPU and GPU can maximize performance. Concrete `DGMaterializationStrategyBase` classes define the materialization data structure and traverse the graph using the *strategy* design pattern (Gamma et al., 1995) to fill the structure with data and create any index structures required to facilitate computation. The algorithms described in Chapter 3 operate on *all* graph elements and do not require partitioning or special processing, a feature of force-directed layout algorithms. Due to time constraints, the concrete `DGMaterializationStrategyNaive` class uses adjacency lists (Section 5.8.3.1) and matrices (Section 5.8.3.2) exclusively, but other data and index structures will be added in future iterations. We intend to

use the Data Calculator paradigm introduced by Idreos et al. (2018b) to facilitate the interactive or semi-automated design of advanced materialization data structures and integrate our results into the `DGMaterializationStrategyBase` class cluster in a future version of this project. Algorithms implementing these advanced materialization strategies must subclass `DGMaterializationStrategyBase` and embed an instance of this class within their concrete `DGLayoutEngineBase` subclasses.

Dynamical.JS follows the method of Qu et al. (2017): materialized graphs consist of three linear arrays: (1) a uniform buffer containing graph-level attributes and constants required by the algorithm, (2) a vertex array to store position, offset, and other attribute values for each vertex, and (3) an index array containing offsets into the vertex array to store edges.

$$
\begin{aligned}
M_{G_t} &\leftarrow \left\{A^M_{G_t}, V^M_{G_t}, E^M_{G_t}\right\} \\
A^M_{G_t} &\leftarrow [a_i, \dots] : a_i \in A_{G_t}, i \in (0, |A_{G_t}|] \\
V^M_{G_t} &\leftarrow [m(v_i), \dots] : v_i \in V_{G_t}, i \in (0, |V_{G_t}|] \\
m(v) &\leftarrow \langle \vec{\boldsymbol{p}}_v, \vec{\delta}_v \rangle \leftarrow \langle \mathbf{p}_v.x, \mathbf{p}_v.y, \delta_v.x, \delta_v.y \rangle \\
E^M_{G_t} &\leftarrow [\langle i_u, i_v \rangle, \dots,] : \langle u, v \rangle \in E_{G_t}, i \in (0, |V_{G_t}|]
\end{aligned}
\tag{5.1}
$$

Each element in the naive materialized vertex array $V^M_{G_t}$ is a 4-tuple of floating-point numbers corresponding to the current position of the vertex $\vec{\boldsymbol{p}}_v$ and the sum of forces acting on the vertex $\vec{\delta}_v$. Each element in the edge array $E^M_{G_t}$ is a tuple containing the indices of the source and target vertices, respectively. $V^M_{G_t}$ and $E^M_{G_t}$ are laid out linearly in memory to ensure the graph data are sequentially accessed by the GPU when processed by applicable GLOs. If WebGPU is supported on the

Table 5.1. Time complexity cost of operations on graph ADTs implemented as adjacency lists

Operation	Cost (Adj. List)	Cost (Adj. Matrix)	Cost (MultiMap)
LOAD(G, E, V)	O$(\|V\| + \|E\|)$	O$(\|V\| + \|E\|)$	O$(\|V\| + \|E\|)$
ADD-NODE(G, v)	O(1)	O$(\|V\| + \|E\|)$	O(1)
ADD-EDGE(G, e)	O(1)	O(1)	O(1)
REMOVE-NODE(G, v)	O$(\|V\|)$	O$(\|V\| + \|E\|)$	O(1)
REMOVE-EDGE(G, e)	O$(\|E\|)$	O(1)	O(1)
ADJACENT(G, v_0, v_1)	O$(\|V\|)$	O(1)	O(1)
NEIGHBORS(G, v)	O$(\|E\|)$	O$(\|V\|)$	O(1)

current platform, the underlying data type for each buffer is a `GPUBuffer` mapped at creation. Otherwise, the buffer type is `ArrayBuffer`.

To optimize CPU-bound lookup times for nodes and edges in Javascript, each adjacency list is implemented using the built-in `Map` object. `Map`s are associative arrays linking each element *value* with a unique *key*, which all `DGGraphBase` objects implement as instance variables. According to the ECMAScript specification, the underlying implementation of Javascript `Map`s may be implemented as `HashMap`s, `SearchTree`s, or other data structures with guaranteed sub-linear access times (European Association for Standardizing Information and Communication Systems, 2015). Table 5.2 lists the lookup costs of using Javascript `Array`s versus `Map`s.

Adjacency Lists. As shown in Listing 5.5, the basic graph ADT implementation in the `DGGraph` object uses an adjacency list to store edges. Adjacency lists are highly flexible and parallelizable, as the node and edge arrays are extensible and filterable via re-interpretation as *skip lists* or *queues*, depending on application requirements. Generally, adjacency lists are preferable for *sparse* graphs with few edges ($|E| \ll |V|^2$) as opposed to *dense* graphs, where the number of edges approaches the maximum ($|E| \approx |V|^2$) (Cormen et al., 2009). Filters and other operations based on the *sequential scan* data access primitive operate in linear time O$(|V|)$ when applied

Table 5.2. Time complexity cost of lookup operations on adjacency lists implemented with `Array` or `Map`.

Operation	Cost (Array)	Cost (HashMap)	Cost (SearchTree)
LOOKUP(G, e)	O($\|E\|$)	O(1)	O($\log \|E\|$)
LOOKUP(G, v)	O($\|V\|$)	O(1)	O($\log \|V\|$)

to arrays. The time complexity for other basic operations can be easily estimated (Table 5.1).

Adjacency Matrices. Adjacency matrices are simple and intuitive data structures useful for *dense* graphs due to their fast access times if used to determine whether or not an edge exists. Adjacency matrices are 2D (two-dimensional) `bool` arrays that store topology for non-weighted graphs (Bleiweiss, 2008). The most basic form is a matrix of $|V|^2$ bits, with each bit set to `1` if an edge exists and `0` otherwise. Using this method, undirected graphs can be represented either as upper- or lower-triangular bit matrices using $\frac{|V|^2}{2} - |V|$ bits. Directed graphs can be represented by combining upper and lower matrices and setting bits along the diagonal to `0`. The upper- and lower-triangular matrices are *strictly triangular* for graphs that disallow self-loops (i.e., where nodes may not connect to themselves). Multigraphs can be represented by either extending each matrix element to be n-bits wide or by creating n 1-bit matrices. Operations for looking up, adding, or removing edges from adjacency matrices are exceedingly fast (O(1)) and are ideal for rendering small, static graphs. However, adjacency matrices tend to be wasteful for large, sparse graphs. In addition, adding or removing vertices requires the adjacency matrix to be rebuilt anew, as all bits are stored linearly in memory. Therefore, adjacency lists are not generally appropriate for dynamical graphs, which may update unexpectedly while the graph is being explored. As Dynamical.JS is geared toward dynamical graphs, support for adjacency matrices is limited. Adjacency matrices are implemented with

`DGBitMatrix` classes.

5.8.4 Graph Traversal

The `DGMaterializationStrategy` class cluster creates materializations via graph traversal, combining the visitor (`DGGraphVisitor`), iterator, and memento design patterns (Gamma et al., 1995) to build a binary representation of the graph used in the specified layout algorithm (Figure 5.8, Figure 5.7). Many memory access patterns are described in the literature; however, most force-directed layouts use sorted, linear access patterns to enhance data locality in GPU memory (Krüger & Westermann, 2003; Lefohn et al., 2006). Sorting allows the materialization to proceed as a series of scan operations instead of random probes Frishman & Tal (2008); Wang et al. (2017); Qu et al. (2017). Graph traversal is supported over all graph types using the `DGGraphTraversal` interface, which supports *breadth-first* and *depth-first* search algorithms (Cormen et al., 2009) via the *visitor* design pattern (Gamma et al., 1995). Listing C.7 shows the naive implementation of these algorithms. Using basic *sequential scan* operations, materializations can be traversed by the CPU or the GPU. By default, four traversal-based sorting algorithms are supported by Dynamical.JS:

1. *Unsorted*: The nodes and edges are materialized in the order they were inserted into the graph ADT (Qu et al., 2017).

2. *Source-First*: Edges are sorted in source-target order ($u \rightarrow v$) using Khan's algorithm to perform a topological sort (Kahn, 1962; Bleiweiss, 2008).

3. *Destination-First*: Edges are sorted by target-source order ($v \rightarrow u$) using breadth-first search (Lefohn et al., 2006).

4. *Weighted Sort*: Partitioned graph nodes are bucket-sorted by attribute values (Frishman & Tal, 2008).

5.8.5 Graph Updates

Dynamism is the key feature of any exploratory graph visualization framework. As such, a method for keeping track of changes occurring during graph exploration tasks is an essential feature of all such frameworks. Because materializations are immutable *by the CPU*, changes to a graph's underlying ADT cannot be reflected immediately. Re-materialization is a computationally expensive process and cannot be performed in realtime during an exploratory graph session. Further, the layout and rendering EGOs (Exploratory Graph Operations) require graph stability, meaning neither nodes nor edges can be added to the graph while executing these operations. The literature speaks of adding new nodes or manipulating old nodes' attributes but not of deleting nodes between intermediate visualizations (Frishman & Tal, 2008; Wang et al., 2017; Sheng et al., 2019; Lin & Huang, 2021). Removing graph elements from materializations is a significant problem that can lead to noticeable breaks in animation. Dynamical.JS introduces two concepts to collect and merge changes in the background to limit re-materialization.

Key Graphs. A *key graph* is an immutable clone of a graph ADT encoding the graph state at time t and roughly corresponds to a timeslice. Key graphs may be generated periodically or when some analyst-specified condition is met. Key graphs are built using the CLONE GDO (Section 2.8.3). Each graph element has a mandatory `birthdate` attribute and optional `deathdate` attribute to determine if the element should be included in the current graph layout L_t or rendered frame R_t. To support arbitrary scaling of timeslices, the `birthdate` and `deathdate` attributes are offset from the `birthdate` of the parent graph, measured in milliseconds.

Change Sets. Graph elements may be added to or removed from the ADT singularly or in batches. These updates are known as *temporal events*, which prevent the

visualization algorithm from smoothly interpolating between them without wasting large amounts of data. Each frame must store empty values for non-existent graph elements or re-materialize the graph after each event. Change sets are aggregated temporal events that must be merged into the previous key graph using the MERGE-LAYOUT GLO. Change sets are subgraphs implemented using the basic graph ADT and contain all changes made to the parent graph over some time interval. Change sets are independently materialized to facilitate animation but are subsequently merged with appropriate key graphs once layout and animation are complete (Frishman & Tal, 2008).

5.8.6 File Formats

`DGGraph` objects serialize and deserialize data for the LOAD and STORE GDOs using the *memento* design pattern (Gamma et al., 1995), serialized as one of two JSON (JavaScript Object Notation) file formats. JSON files are ASCII (American Standard Code for Information Interchange) or UTF-8 (Unicode Transformation Format, 8-bit) encoded and consist of a single `graph` object abstracting the formula $G \leftarrow (V, E, A)$ such that:

1. `attributes` is an associative array of key-value pairs containing graph attributes.
2. `nodes` is an array of unique `node` objects, each with uniquely defined `key` and an arbitrary number of other parameters, and mirror the data structure implemented by `DGMaterializationStrategyNaive`.
3. `edges` is an unordered array of unique tuples $e \leftarrow \langle k_{v_0}, k_{v_1} \rangle$ which meet the following criteria:

$$e = \langle k_{v_0}, k_{v_1} \rangle : \begin{cases} k_{v_0} \in K_V & k_{v_0} \text{ is a key referencing a previously defined node} \\ k_{v_1} \in K_V & k_{v_1} \text{ is a key referencing a previously defined node} \\ k_{v_0} \neq k_{v_1} & k_{v_0} \text{ and } k_{v_1} \text{ are not the same} \end{cases}$$

An example of the JSON file format can be found in Listing 5.6.

Listing 5.6. An example graph serialized into the JSON file format

```
{
  graph: {
        attributes: { birthdate: 0, directed: false,...},
        nodes: [  { key: 'key0', param: value, ...}, ...,],
        edges: [ [ 'key0', 'key1' ], ..., [ 'keym', 'keyn' ], ],
  }
}
```

An index-only JSON format is also supported, where nodes are *implicitly* defined via indices stored in the `edges` array. As the document is parsed, the array of nodes V is extended by adding as many new, 'unkeyed' nodes until we reach the maximum index: $|G| \leftarrow MAX(|G|, MAX(i_{v_0}, i_{v_1}))$. This format consists of a single `graph` object abstracting the formula $G \leftarrow (E)$, where `edges` is an unordered array of unique tuples of indices $e \leftarrow \langle i_{v_0}, i_{v_1} \rangle$ which meet the following criteria:

$$e = \langle i_{v_0}, i_{v_1} \rangle : \begin{cases} i_{v_0} \in (0, |V|] & i_{v_0} \text{ is the index of a previously defined node} \\ i_{v_1} \in (0, |V|] & i_{v_1} \text{ is the index of a previously defined node} \\ i_{v_0} \neq i_{v_1} & i_{v_0} \text{ and } i_{v_1} \text{ are not the same} \end{cases}$$

An example of the JSON (indexed) file format can be found in Listing 5.7.

Listing 5.7. An example graph serialized in the JSON (indexed) file format

```
{
  graph: {
    edges: [ [ 0, 1 ], ...,  [ n, m ], ],
  }
}
```

5.9. Layout Module

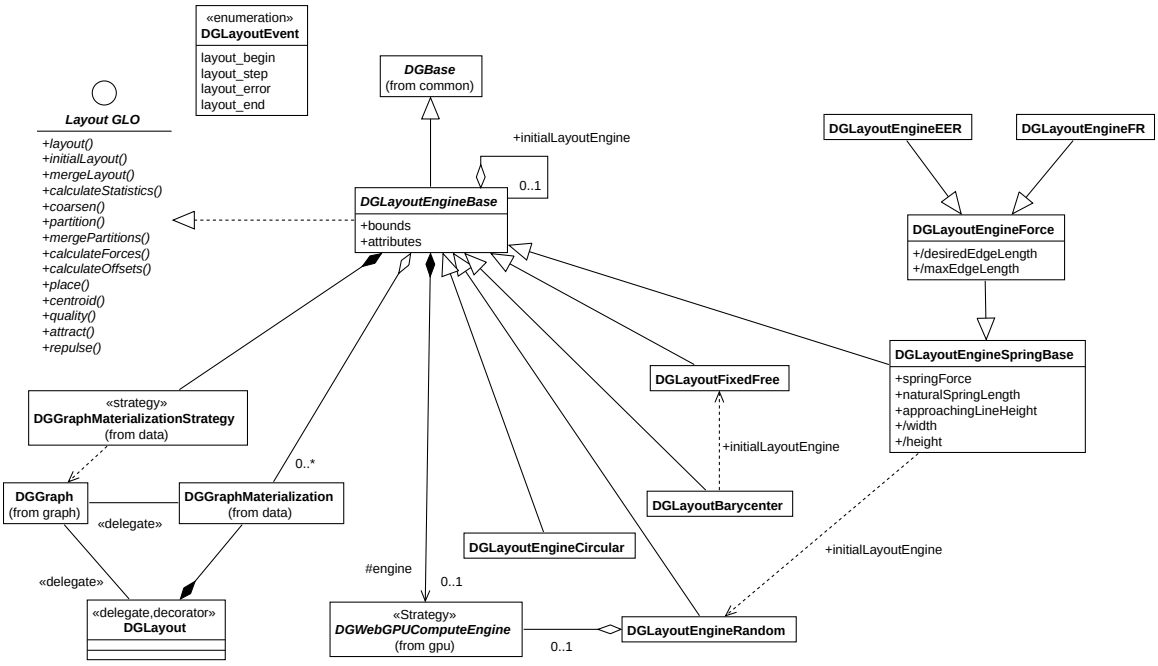


Figure 5.9. A UML class diagram of the Dynamical.JS Layout Module.

The Layout Module consists of a class cluster based on the abstract `DGLayoutEngineBase` class (Listing C.9), which utilizes the composite, pattern, strategy, and chain of responsibility design patterns (Gamma et al., 1995). The `DGLayoutEngineBase` class cluster allows the programmer to compose complex

layouts by chaining predefined GLO methods. `DGLayoutEngineBase` implements both `EventSource` and `DGExtensibleBase` interfaces and is generally implemented as both a default implementation that executes on the CPU, which is then subclassed to override the CPU implementation with methods that execute GLO kernels on a GPU via a concrete `DGComputeEngineBase` subclass (Listing C.11) which performs the actual calculations.

5.9.1 Layout Execution

As described below in Section 3.1, graph layout algorithms progress through a sequence of stages. Each stage corresponds to one or more GLOs or EGOs, each with its own WebGPU kernel. Each algorithm is implemented as a concrete subclass of `DGLayoutEngineBase`, which implements the *strategy* design pattern (Gamma et al., 1995) to control the execution of these kernels. Within the Layout Module, each GLO is a bulk-synchronous "step" that manipulates a materialization, and graph layout algorithms are built from a sequence of these steps. Inter-stage dependencies may exist, but individual operations within a step are processed in parallel. Each GLO corresponds to an overridden `DGLayoutEngineBase` instance method implemented in the appropriate concrete subclass.

As shown in Listing 5.8, each `DGLayoutEngineBase` method defaults to a `no-op` (Null-Operation), i.e., an empty function returning the current layout with no changes unless overridden. The graph layout algorithm is actually performed by calling the `layout()` method on the graph ADT. This method uses layout-specific `DGComputeEngine` subclasses to calculate the final positions of the nodes, generating a `layout_step` event after each run through the loop. This allows the implementation to choose the stages which are appropriate for a particular algorithm, reuse code wherever possible, and skip stages that are not required for the algorithm to

function. In all cases, the layout engine will work only on the graph's *materialization*, and never the graph itself. This facilitates changes to the underlying graph ADT in response to asynchronous events while the rendering task is taking place. Once the layout is complete, the `layout()` method will generate a `layout_complete` event.

Listing 5.8. The `DGLayoutEngineBase` implementation of the LAYOUT GLO.

```
/**
 * Loop through the body of the algorithm until the stop condition is met.
 * @param {DGGraph} graph - The graph to layout
 * @returns {Promise}
 */
async loop( layout, stop = this.stop ) {
  while( !await stop( layout ) )
    await this.calculateOffsets( layout )
      .then( this.place( layout ) )
      .then( () => { this.emit( DGLayoutEvent.LAYOUT_STEP, this ) } )

  return layout;
}

/**
 * Layout the graph asynchronously.
 * @returns {Promise<DGGraph>} - A promise which will resolve when the graph is laid out
    successfully or fails.
 */
async layout( graph = this.graph ) {
  this.emit( DGLayoutEvent.LAYOUT_BEGIN, { graph: graph } );

  const layout = await this.initialLayout( graph );

  this.calculateStatistics( layout )
    .then( layout => this.partition( layout ) )
    .then( layout => this.loop( layout ) )
    .then( layout => this.mergePartitions( layout ) )
    .then( layout => this.finesse( layout ) )
    .catch( ( error ) => {
      console.error( error );
```

```
        this.emit(
          DGLayoutEvent.LAYOUT_ERROR,
          { graph: this.graph, layout: layout, error: error }
        );
      } )
      .finally( () => {
          this.emit( DGLayoutEvent.LAYOUT_END, { graph: this.graph, layout: layout } );
          return layout;
      } );
  }
```

To facilitate the animation of intermediate vertex positions and split the layout tasks among multiple threads, each `layout-step` event notifies the rendering engine that the materialization may be drawn to the screen. These events may be safely ignored if the application does not require them. Dynamical.JS may also facilitate a more traditional animation method between two final layouts by ignoring interim `layout_step` events and interpolating between the values generated between `layout_complete` events.

Within each GLO kernel, the relationship between materialized vertices and compute threads is one-to-one; each thread computes the layout contribution of a single vertex or edge. In WebGPU, adjacent threads do not share a mutable state. Consequently, combining multiple operations on vertex attributes into a single kernel saves significant memory bandwidth that would otherwise be wasted writing intermediate values to memory and reading them back again. We also avoid an unmap-remap cycle that reloads vertex arrays for each changed attribute. After each thread in the same block has computed the fitness of its vertex, the fitness values are summed into the global fitness metric via parallel reduction (Qu et al., 2017) by the QUALITY GLO's kernel (Section 2.9.15).

Each GLO corresponding to a layout stage requires *at most* two CPU-GPU

map-copy-unmap cycles: one to copy the materialization into GPU memory and one to copy it back out. Adroit usage of WebGPU's `GPUBindGroupLayout` objects can reduce this to a lone cycle pair for the entire algorithm. Further, only the output cycle is required if data can be generated solely on the GPU (e.g., the random or circumscribed positions of the RANDOM and POLYGON layouts). Neither iterations through the PLACEMENT GLOs nor drawing GROs (Graph Rendering Operations) require any memory cycles; only a single function call to switch between appropriate `GPUBindGroupLayouts` is needed.

5.10. Drawing Module

The Drawing Module has two primary responsibilities: drawing graph layouts to the UI and responding to user input. GROs are triggered via events from the Layout Module or user events generated by HIDs (Human Interface Devices) such as mice or keyboards. This module also determines the animation framerate and interpolates between source and destination layouts.

5.10.1 Rendering & Animation

Graph layouts are rendered to the target canvas in response to the `layout_step` and `layout_complete` events generated by the Layout Module (Section 5.9). Like the Layout Module, each GRO or GAO (Graph Analysis Operation) is a self-contained rendering step controlled by a `DGWebGPURendererBase` instance. `DGWebGPURendererBase` is a class cluster that utilizes the same design patterns to implement a staged, idiomatic rendering process. Because it inherits from DGComputeEngine, kernel functions and shaders are loaded and executed using identical methods. `DGWebGPURendererBase` is a concrete subclass of `DGComputeEngine` and inherits most of its methods. However, `DGWebGPURendererBase` subclasses utilize a

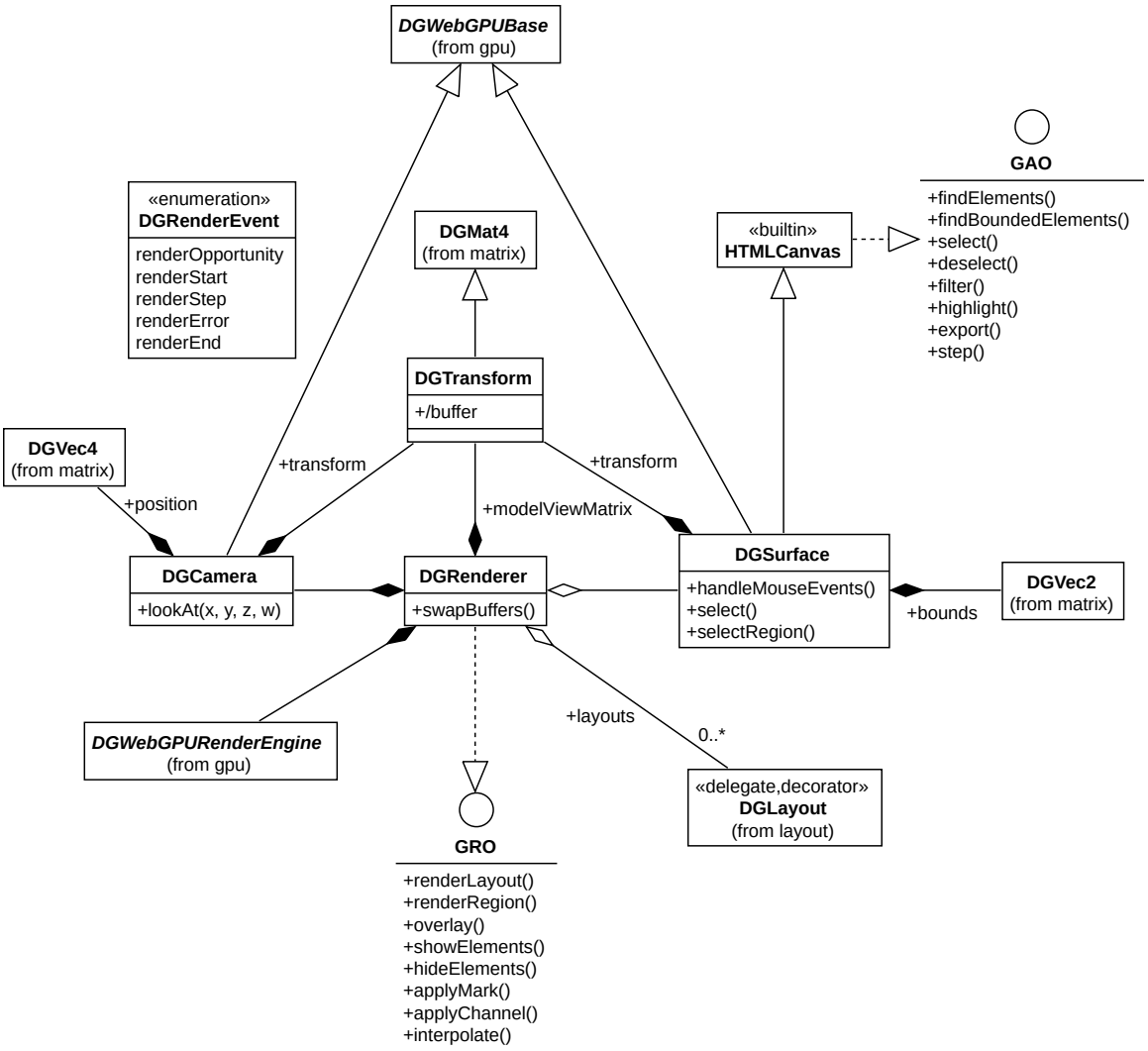


Figure 5.10. A UML class diagram of the Dynamical.JS Drawing Module.

trio of shader programs (Section 4.9.1):

1. A compute shader to determine which materialized graph elements should be rendered to the canvas. This set is updated with the SHOW-ELEMENTS (Section 2.10.6) and HIDE-ELEMENTS (Section 2.10.7) GDOs. The analyst may manually select these elements with the SELECT-ELEMENTS (Section 2.11.3) and DESELECT-ELEMENTS (Section 2.11.3) GAOs. This set is then translated into one or more vertex buffers used by the RENDER (Section 2.10.3) and RENDER-

REGION (Section 2.10.4) GDOs, which are then submitted to the rendering pipeline (Section 4.8). Concrete `DGWebGPURendererBase` subclasses also use *double buffering* to ensure smooth animation. While the rendering pipeline reads one buffer, the compute pipeline (Section 4.9) writes updated vertex data to its twin. After each render cycle is complete, the buffers are swapped, and the cycle repeats. Each buffer pair is defined, controlled, and resized by WebGPU's `GPUBindBufferLayout` object and does not require mapping; all data remains on the GPU and is recycled or purged as needed. Animation is performed by the INTERPOLATE (Section 2.10.10) GRO where the renderer's current vertex buffer and the layout materialization's vertex buffer are the a and b interpolants, respectively.

2. A *vertex shader* that maps vertices into view space using the *model-view transform* matrix (Section 4.8.1). The canvas updates this matrix in response to `resize`, `scroll`, and `zoom` events generated by the UI in response to actions made by the analyst. This matrix can also be manually controlled via the TRANSFORM GRO (Section 2.10.2).

3. A fragment shader that renders geometry defined by the vertex shader into fragments and merges these fragments into the output canvas's framebuffer. Fragment attributes corresponding to visual marks and visual channels, e.g., color or line thickness, are controlled via the APPLY-CHANNEL (Section 2.10.9) and APPLY-MARK (Section 2.10.8) GROs.

Chapter VI.

Conclusion

Employing an operation-centric programming model for graph processing and layout allowed us to design a robust and highly extensible exploratory graph layout framework: Dynamical.JS. The development process uncovered various interesting yet challenging research and development opportunities.

6.1. Summary

Dynamical.JS was devised during an attempt to create an interactive visualization of the worldwide wildlife trafficking networks to identify how patterns of law enforcement actions changed the trade flows over time. The initial work was limited to functional and procedural methods and Javascript's `Canvas2D`. It quickly became apparent that a generalized package, object orientation, an appropriate high-level programming model, and low-level optimizations for parallel graph analytics on GPUs (Graphics Processing Units) were needed. The current design of Dynamical.JS demonstrates that with the proper tools, exploratory graph visualization can become relatively simple and efficient.

Specifically, this work has achieved our high-level goals: the design and implementation of Dynamical.JS meet the key qualities we defined for dynamic graph visualization frameworks based on the twin pillars of abstraction and composition. This Javascript framework is self-contained and takes advantage of the latest GPU technologies for the web while providing a fallback implementation that can be used on any modern Javascript runtime. We decomposed several well-documented graph layout algorithms into a common, staged, and idiomatic layout strategy built of composable EGOs (Exploratory Graph Operations). The stages and EGOs (Exploratory

Graph Operations) described above facilitate the exploration of dynamical graphs with an overview, intermediate, or detail view, focusing on graph structure and temporal changes. We demonstrated the step-wise local refinement graph layouts for static rendering and animation. After each step, Dynamical.JS enables the analyst to visualize intermediate renderings, select a different subset of data, and refine the desired visual representation for their selection. By segregating the graph ADT (Abstract Data Type) from its materialization, we enabled both element selection and multiple representation, alterable at any time via analyst-triggered events or periodic changes in the underlying graph morphology.

During the development of the Dynamical.JS framework, we kept three types of analysts in mind: (1) algorithm designers who wish to invent or modify domain-specific graph layout algorithms, (2) data scientists who wish to utilize GPUs to explore large, multivariate relational networks, and (3) programmers who wish to reproduce the results of this thesis and make improvements to its components. We hope that Dynamical.JS will serve as a standard for exploratory graph visualization on the web.

6.2. Challenges

6.2.1 Multithreading

The Dynamical.JS code executed by the CPU (Central Processing Unit) in Javascript is single-threaded. Extending the asynchronous model to support splitting work among several Web Workers remains unaddressed. Code executed via WebGPU (GPU Computing for the Web) is currently restricted to a single GPU, even on substrates where multiple GPUs are available.

6.2.2 Memory Limits

Due to the nature of Javascript, the largest individual buffer size we can create is 4GiB, so we have limited ourselves to this size. Due to this limitation, Dynamical.JS is restricted to graphs containing 2^{30} vertices and 2^{31} edges. However, graphs of this size are far larger than we can render smoothly.

6.2.3 Limited Optimization

As of this writing, the versions of the WebGPU framework and the WGSL (WebGPU Shading Language) language versions implemented in the various Webkit and non-Webkit browsers are in flux, and the relevant specifications have changed in significant ways. Therefore, full implementation of the Dynamical.JS framework was impossible to complete before the publication of this thesis. However, we include Javascript and WGSL source code compatible as of 1 September 2022 but have removed any code which relies on currently deprecated functionality or syntax. Because the WebGPU specification remains incomplete at the time of this writing, several critical performance metrics were unavailable to us: GPU memory mapping events, performance timestamps, and configurable point sizes remain unsupported. As neither the WebGPU nor the WGSL language specifications are complete, performance measurements were impossible to include; however, performance testing and measurement capabilities are incorporated into the Dynamical.JS design and reference implementation. A benchmarking suite and substrate performance analysis to aid in tuning parameters are reserved for future work.

6.2.4 Timeslicing

Additional functional parameters for timeslicing and data amplification remain unsupported, such as a computational "budget" focus allowing prioritization of structural integrity over animation time-step quantification to help meet environmental constraints (Hadlak et al., 2011). Simonetto et al. (2020) described an event-based model where graph layouts exist in an $n+1$-dimensional spacetime cube, where each graph layout is an n-dimensional timeslice with arbitrarily fine time sampling rates (Simonetto et al., 2020). This would require a stream of data and events to be recorded in a timeline data structure to describe the temporal domain containing the dynamics of the dynamical graph, which is currently unsupported (Hadlak et al., 2011). The FILTER-ELEMENT GLO (Graph Layout Operation) is a crucial analysis task for studying dynamical graphs and is the most complex. Filtering static or dynamic graphs is challenging yet achievable; however, filtering these dynamical graphs by specific morphological changes is a task beyond the current scope of Dynamical.JS.

6.3. Future Work

Moving forward, several aspects of Dynamical.JS need improvement, including architecture, execution model, performance characterization, core graph operators, new graph primitives, and usability changes.

6.3.1 Architectural Changes

Expansion to new graph types, layout algorithms, and complex encodings will make Dynamical.JS a more functional framework for exploratory graph visualization. Doing so will require the addition of several new GDOs (Graph Data Operations) and the incorporation of GPU-enabled versions of graph-theoretical operators like APSP

(All Pairs Shortest Path). The LOAD and STORE GDOs require support for additional file formats and GML (Graph Modeling Language) to support the importation of larger and more complex graph types. Support for other visualization techniques, such as matrix or circular layouts, complex visual marks, small multiples, or spectral energy minimization techniques, requires investigation.

The current reference implementation does not support the export of visualizations to textures or support multiple coordinated views in realtime. This thesis focused on decomposing current graph layout algorithm techniques but did not focus on methods of local *in situ* combinations of multiple layouts nor switching between them interactively (Hadlak et al., 2011). As the analyst's focus may shift during analysis, changing computational techniques with different graph foci may be necessary (Hadlak et al., 2011).

6.3.2 Optimization Changes

Graph-theoretical operations at the GDO level will benefit from adding additional parallelism for both CPU and GPU. Doing so will also expose new opportunities to build memory- and workload-optimizing building blocks that can be plugged into or replace higher-level EGO implementations.

The design space for materialization data structures is vast (Lefohn et al., 2006; Ahn et al., 2014; Idreos et al., 2019, 2018a). Until the WebGPU memory access model is complete, it is impossible to investigate data structures capable of cache-conscious access patterns in Javascript. Further, the current materialization format requires large-scale GPU calculations but walking through a streaming graph or resampling materializations on several scales to speed up filtering or coarsification strategies is unsupported.

Complex graphs often contain rich data on nodes and edges. Because

Javascript lacks direct access to binary data except in specific circumstances, materializations are limited to numerical data; further research is needed to determine optimal methods for incorporating arbitrary data. Currently, Dynamical.JS puts all the information into linear arrays to facilitate sequential scans on the GPU, which is not ideal—adding the capability of loading rich attribute information onto the GPU could enable more complex graph query tasks and allow us to support several data access primitives that require rich information during computation (Wang et al., 2017).

Currently, there are few solutions for efficiently handling general graph mutations on GPUs (Wang et al., 2017). Mutable materializations require continuous processing of graph layout algorithms on dynamical inputs to update the graph ADT incrementally and update materializations given the changes. Dynamical.JS needs to provide either approximated results or expose such capabilities to future programmers (Frishman & Tal, 2008; Qu et al., 2017).

Appendix I.

Glossary

Analyst A human who interacts with an *application* to solve a problem or to gain insight by analyzing its output. ii, 4, 9, 11–13, 16–18, 21, 24–28, 32, 33, 46–49, 57, 83, 85, 86, 123, 135, 143, 144, 146, 149, 151, 153, 154

Application An interactive *program* or *process* that runs on a specific *platform* to produce output useful to the *analyst*. Adobe Photoshop, Microsoft Word, and OmniGraffle are examples of applications. 6, 9, 11, 12, 25, 29, 32, 46, 48, 83, 88, 89, 91, 94, 98, 104, 115, 132, 151, 156

Canvas The 2D (two-dimensional) drawing surface where *graphs* will be rendered. Such canvases may be visible on-screen, off-screen (cached in RAM (Random Access Memory)), or *serialized* to an output file. 33, 46, 90, 91, 143, 144, 157

Compute Core A single GPU, streaming multiprocessor, ALU (Arithmetic Logic Unit), CPU, or co-processor made available by the hardware *substrates*. 9, 28, 29, 99, 104, 105, 156

Dynamic Any object, structure, or concept that changes in a predefined or predictable manner or has a known *retrogressive* evolution. ii, 5, 9, 21, 22, 27, 152

Dynamic (Online) Graph Layout Algorithm A graph layout algorithm which works on *dynamic graphs* whose morphology changes due to exceptional or periodic circumstances (Di Battista et al., 1994). *Dynamic graph layout algorithms* are generally simple extensions of *static graph layout algorithms*. 4, 5, 49, 50, 54

Dynamic Graph A temporally dynamic yet morphologically static $n + 1$-dimensional *graph* $D \leftarrow (G, T)$ which changes due to exceptional circumstances or a *graph* whose morphological history is known or can be *retrogressively* derived. For example, a connectivity diagram of a wireline network or the electrical lines (*edges*) in a modern home where outlets (*nodes*) are rarely added or removed. Because these graphs do not often change (if at all), static *graph drawing algorithms* are generally used without modification or special optimizations but at a high computational cost. When used in offline *graph drawing algorithms*, the *graph* and all of its changes are encoded as a spacetime cube, and the *graph* state G_t at time t is known as a *timeslice* $D_t \leftarrow (G_t, t)$ (Cohen et al., 1992). 8, 21–23, 27, 69, 145, 148, 151, *see* Graph

Dynamical Any object, structure, or concept that evolves *progressively* or changes in an unstructured or unpredictable manner in response to external events. A cloud computing service that reallocates resources due to fluctuations in current client demand is an example of a dynamical *substrate*. 5, 6, 9, 27, 150

Dynamical (Online) Graph Layout Algorithm A graph layout algorithm which works on *dynamical graphs* whose morphology evolves due to unpredictable events such as user interactivity (Di Battista et al., 1994). 50

Dynamical Graph A *graph* which is always in a state of flux, *progressively* evolving due to a stream of events which may add, remove or modify its *node*, *edges* or *attributes* in an unpredictable manner. For example, a visualization of a social network or a COVID-19 contact tracing program where potentially infected patients (*nodes*) and their contacts (*edges*) are added, removed, or filtered continuously, either one at a time or in batches. Due to these constant,

arbitrary changes, static *graph drawing algorithms* must be modified to take the changes into account, both for performance reasons and to preserve the *analyst*'s *mental map* of the *graph*, leaving unchanged subsections of the *graph* intact where possible. ii, 4, 7, 8, 12, 133, 146, 148, 152, 154, *see* Graph

Force-Directed Layout The *graph layout* of a node-link diagrams based on metaphorical "forces" which push and pull *nodes* to their final positions by minimizing a global "energy" function. These layouts are frequently utilized because they naturally lead to symmetrical and aesthetically pleasing layouts by default (Nobre et al., 2019). 27, 58, 60, 62, 63, 71, 75, 130, 134

Framebuffer A dedicated region of VRAM (Video RAM) for storing output pixel data. 87, 88, 90, 91, 93, 96, 97

Framework A collection of software functions, APIs (Application Programming Interfaces), code libraries, or other technologies a given *system* provides to perform related computational tasks. OpenGL (Open Graphics Library), DirectX, WebGPU, and Dynamical.JS are examples of frameworks. x, 2, 6, 9–14, 18, 32, 33, 46, 80, 83, 87, 89, 97, 98, 103–108, 110–113, 115, 145–148, 156, 157

Graph A logical structure which abstracts a set of entities (called *vertices* or *nodes*), and the relationships between these entities (called *edges*): $G \leftarrow (V, E, A)$. Just like relationships between real entities in the world, *edges* between the *nodes* may be directed ($\langle u \leftarrow v \rangle, \langle u \rightarrow v \rangle$), undirected ($\langle u \leftrightarrow v \rangle$), or even self-referential ($\langle u \circlearrowleft u \rangle$). Large graphs are also known as *networks* (Tutte, 1963). ii, xi, xii, xv, 1–6, 8, 10–28, 31–38, 40, 46, 49–52, 54–57, 60–64, 66–68, 74, 75, 86, 90, 98, 104, 107, 113, 116, 117, 119, 120, 122–125, 127–140, 143, 145–157, *see* Multivariate Network

Graph Drawing Algorithm An algorithm that reads as input a *graph layout* $L \leftarrow \textsc{layout}(G)$ of *graph* G and produces as output a drawing (*rendering*) according to a given graphics standard, including any specified *visual marks* and *visual channels*. A *graph drawing algorithm* may pass through any *aesthetic* constraints to the *graph layout algorithm* used to generate its input. 3, 4, 7, 51, 152, 153, 155, *see* Graph Layout Algorithm

Graph Layout A *graph layout* is a graph where the nodes $\vec{\boldsymbol{p}}_v \in P_G$ have been positioned within an arbitrary *layout (coordinate) space*. ii, 5, 16, 17, 19, 21–23, 27, 32, 33, 43, 45, 46, 50, 51, 53, 54, 56, 60, 61, 63, 67, 71, 75, 87, 103, 107, 117, 123, 135, 142, 145, 146, 148, 153, 154

Graph Layout Algorithm An algorithm that reads as input a combinatorial description of a *graph* $G \leftarrow (V, E, A)$, and produces as output a *graph layout* $L = \textsc{layout}(G)$. "Readability" of the resultant *layout* is controlled via optimization goals expressed as parametric *aesthetic* and design constraints imposed by the presentation medium and application domain (Di Battista et al., 1994). 3, 13, 21, 24, 51–53, 56–58, 84–87, 92, 104, 106–109, 116, 119, 127, 129, 130, 134, 139, 145, 146, 149–152, 154, 156

Layout Space The n- or $n+1$-dimensional coordinate space into which *graph layout algorithms* place *graph* elements. 16, 19, 33, 38, 41, 42, 45, 49, 51, 54, 56, 58, 60, 61

Mental Map The morphological structure of a *dynamical graph* as currently represented in an *analyst*'s mind. The *analyst*'s *mental map* remains stable across *affine* transformation operations (rotation, scale, translation), while local and global properties of the *graph* evolve without abrupt transitions during both

constructive and destructive operations (filtering, growth, pruning, simplification) (Misue et al., 1995). ii, 4, 12, 13, 26, 48, 49, 54, 57, 75, 86, 153

Multivariate Network A complex *graph* where both the contained *nodes* and their relational *edges* are arbitrarily complex. This complexity arises due to a variable number of *attributes* each *node* or *edge* may possess. Depending on the operation, *attributes* may strengthen, weaken or negate the relationship between *nodes*, which in turn affect the morphology of the *graph* as determined by a *graph drawing algorithm* (Nobre et al., 2019). ii, 1, *see* Graph

Network . 27, 83, 145, 146, *see* Graph

Node The fundamental graph-theoretical unit from which all *graphs* are formed. ii, x, xiii, 2, 3, 5, 10, 16, 19–27, 32, 34, 38–42, 49–56, 58–69, 71, 74, 75, 86, 89, 90, 107, 108, 122, 123, 125, 128, 129, 132–135, 137, 139, 152–155, 157, 158

Node-Link Diagram The most common graphical representation of *graphs*, where *nodes* are represented as a point or circle *visual marks*, and *edges* are represented as line or curve *visual marks* connecting the *nodes* (Figure 1.1). These diagrams place *nodes* based on intrinsic *attributes* of the entities themselves, constrained by aesthetic requirements imposed by *graph drawing algorithms*, which visually clarify the graph. For example, by limiting the number of *edge crossings* or ensuring a minimum distance between *nodes* or *edges* (Tutte, 1963; Nobre et al., 2019). x, 24, 25, 27, 51, 153, 155, 158

On-Node/On-Edge Encoding Refers to modifying the *visual channel* (size, color, line weight, &c.) of a *node* or an *edge* or embedding complex *visual marks* (bar charts, line charts, &c.) in a *node* or a *edge* in a *node-link diagram* (Nobre et al., 2019; Bertin, 2011). x, 7, 24

Platform A specific collection of *frameworks*, execution environments, or programming languages one or more *applications* require for execution. The Javascript and Python runtimes are examples of platforms. 3, 6, 9, 84, 88, 110, 151

Small Multiple A complex visual mark, usually a complete multivariate visualization in its own right, used to display different slices of a dataset. Small multiples are commonly arranged in a grid and are indexed by category or a label, sequenced over time like the frames of a movie or ordered by some quantitative variable not used in the single image itself (Tufte, 1990). 21, 24, 27

Spacetime Cube An n+1-dimensional *visualization space*, consisting of n (normally Euclidean) spatial dimensions and one time dimension, used to conceptualize the dynamic properties of a dataset. A general heuristic is to decide on a n-dimensional visual representation of the data for a given *timeslice* and then extrude it over time. An entity which moves on a 2D map becomes a static 3D (three-dimensional) trajectory when visualized in a spacetime cube (Bach et al., 2017). 21, 23, 24, 27, 49, 50, 53, 148, 152, 157

Static Graph Layout Algorithm A graph layout algorithm which works on *static graphs* with no temporal attributes, where morphology is not expected to change (Di Battista et al., 1994). 22, 49, 50, 151

Substrate This non-standard term always refers to the collection of CPU, GPU, and ASIC (Application-Specific Integrated Circuit) resources provided by a given hardware configuration. This includes instruction sets, *compute cores*, co-processors, memory configurations, or other control hardware to facilitate computation. Substrates may be *monolithic* (confined to a single host), *dis-*

tributed (shared among multiple hosts), or *virtual* (logical partitions of the above resources which may operate concurrently on the same host). 8–10, 28, 29, 48, 82, 84, 87, 88, 90, 97, 101–104, 106, 110, 120, 122, 151, 152, 157

System Unless qualified, this refers specifically to the OS (Operating System) running on a given substrate. Windows, macOS, and Linux are examples of systems. 6, 9, 12, 18, 82–85, 87, 88, 107, 110, 119, 153

Timeslice An n-dimensional hyperplane within the $n + 1$-dimensional *Spacetime cube*, containing (non-temporal) data at a specific time t (Bach et al., 2017). 21–24, 28, 34, 48–50, 53, 55, 128, 129, 135, 148, 156

Vertex A data structure that describes a point in two or 3D space and is the fundamental unit from which all graphical primitives are derived in a given computer graphics *framework*. This term also refers to a *graph node* that has been materialized for layout and rendering. 10, 34, 58, 64, 71, 91–96, 98–101, 107, 120, 128, 130, 131, 133, 141, 143, 144, 147

View Space The two-dimensional coordinate space defined by the output canvas. 46, 48, 49, 56, 91, 95, 97, 144

Visual Channel A variation or other complication of a *visual mark* which encodes the current value of the represented data variable (Bertin, 2011). For example, the size or area of a *visual mark* may increase as the value increases. Likewise, the circle's color may vary depending on the categorical type. Many channels can be applied to *visual marks*, alone or in combination. For example, area, position, connectivity, grouping, line weight, color, resolution, &c. 47, 51, 74, 123, 144, 154, 155, 158

Visual Mark A shape, glyph, or other graphical element used to represent data variables in a visualization (Bertin, 2011). Simple *node-link diagrams* generally represent *nodes* as circles and *edges* as lines. Marks may be *simple* (representing a single data variable) or *complex* (representing multiple data variables simultaneously). Marks vary based on the current value of the variable at the time the visualization has been drawn. This variation is known as the mark's visual channel. 24, 25, 27, 44, 47, 51, 74, 123, 144, 149, 154–157

Appendix II.

Acronyms

`no-op` Null-Operation. 57, 139

1D one-dimensional. 30

2D two-dimensional. 1, 6, 30, 33, 51, 133, 151, 156, 157

3D three-dimensional. 10, 30, 98, 100, 106, 156, 157

ADT Abstract Data Type. xi, xii, 33, 112, 122–125, 129, 132, 134–136, 139, 140, 146, 150

AI Artificial Intelligence. 2, 3

ALU Arithmetic Logic Unit. 81, 92, 120, 151

AMP Asymmetric MultiProcessing. 84

APFS APple File System. 81

API Application Programming Interface. ii, 9, 15, 88, 89, 106, 108, 109, 153

APSP All Pairs Shortest Path. 104, 148

ASCII American Standard Code for Information Interchange. 136

ASIC Application-Specific Integrated Circuit. 8, 88, 89, 156

ASM Assembly Language. 101, 102, 106

BFS Breadth-First Search. 103

BLAS Basic Linear Algebra Subprograms. 119

CPU Central Processing Unit. x, 8, 13, 15, 29–31, 48, 81, 82, 87, 89, 90, 99, 101–103, 106, 109, 120, 121, 127, 128, 130, 132, 134, 135, 139, 141, 146, 149, 151, 156

CU-BLAS CUDA Basic Linear Algebra Subprograms. 103

CUDA Compute Unified Device Architecture. 89, 104, 106, 107

DOM Document Object Model. 8

DRAM Dynamic RAM. 87

EGO Exploratory Graph Operation. ii, xiv, 7, 13, 24, 25, 33, 78, 93, 135, 139, 145, 149

ES7 ECMAScript 7. 86

FIFO First In, First Out. 84, 92

GAO Graph Analysis Operation. 46, 142, 143

GCC GNU Compiler Collection. 106

GDO Graph Data Operation. xii, 33, 34, 46, 60, 126, 128, 135, 136, 143, 144, 148, 149

GIS Geographic Information Systems. ii

GLO Graph Layout Operation. xii, xv, 16, 38, 39, 51, 52, 128, 131, 136, 139–142, 148

GLSL OpenGL Shading Language. 107, 108

GML Graph Modeling Language. 149

GPGPU General-Purpose Graphics Processing Unit. 101–109

GPU Graphics Processing Unit. x, 3, 5, 8, 13, 15, 29–31, 48, 81, 82, 84, 89, 90, 92, 93, 96–99, 101–108, 112, 115, 119–121, 127–131, 134, 139, 141, 142, 144–151, 156

GRO Graph Rendering Operation. xii, 42, 142, 144

HAL Hardware Abstraction Layer. 102

HID Human Interface Device. 17, 142

HTML HyperText Markup Language. 110

HTML5 HyperText Markup Language, Version 5. 111

IDE Integrated Development Environment. 82, 110

IoT Internet-of-Things. 98, 110

JIT Just-in-Time. 83

JPEG Joint Photographic Experts Group. 47

JSON JavaScript Object Notation. xv, 136–138

LAN Local Area Network. 82

LLVM Low-Level Virtual Machine. 106

MIMD Multiple-Instruction, Multiple Data. 105

OBE Object-Bound Execution. 105

OpenCL Open Compute Language. 106, 107

OpenGL Open Graphics Library. 9, 88, 89, 104, 106, 153

OS Operating System. 9, 157

PDF Portable Document Format. 47, 110

PID Proportional Integral Derivative. 52

PRN Pseudo-Random Number. 119–121

PRNG Pseudo-Random Number Generator. 119, 120, 122

RAM Random Access Memory. 28, 29, 151

ROM Read Only Memory. 87

SIMT Single-Instruction, Multiple Thread. 100

SMP Symmetric MultiProcessing. 84

SOC System-On-a-Chip. 81, 82

SPMD Single-Program, Multiple Data. 104, 105

SRAM Static RAM. 88

SSD Solid State Drive. 81

SSSP Single Source Shortest Path. 104

SVG Scalable Vector Graphics. 8, 47

TFLOPs Tera- Floating Point Operations s^{-1}. 82

UI User Interface. 5, 12, 16, 17, 86, 116, 142, 144

UMA Unified Memory Architecture. 29, 82

UML Unified Modeling Language. x, xi, 2, 15, 112, 113, 117, 118, 121, 122, 124, 125, 127, 138, 143

UTF-8 Unicode Transformation Format, 8-bit. 136

VBL Vertical BLanking interval. 87

VLSI Very Large Silicon Integrated circuit. 3

VRAM Video RAM. 29, 31, 87–89, 153

W3C World Wide Web Consortium. 80, 108

WebGL Web Graphics Library. 84, 107, 108

WebGPU GPU Computing for the Web. ii, 6, 7, 9, 81–84, 89, 108, 109, 115, 116, 122, 131, 139, 141, 142, 144, 146, 147, 149, 153, 229

WGSL WebGPU Shading Language. 6, 81, 83, 108, 109, 119, 122, 147

Appendix III.

Code

C.1. Javascript Code

C.1.1 DGBase

Listing C.1. Javascript code for `DGBase`.

```
/**
 * @file /src/common/base.js
 * @author Rob Dotson
 * @copyright 2020 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the implementation of the abstract DGGraphLayoutBase class.
 *
 * @created 02 July 2020
 *
 * @todo Need to test browser for ES6 compliance (should be all webkit + firefox)
 * @todo Need to fix the code for listening for all events '*'
 */

import {DGAbstractInstantiationError} from './error/error.js';

/**
  * Enum for WebGPU events.
  * @readonly
  * @enum {string}
  */
export const DGBaseEvent = Object.freeze({
  DESTROY: "destroy",
});

/**
 * DGBase is an abstract base class which almost all DG objects inherit. It allows for type
    reflection, cloning and printing.
 * @extends EventTarget
```

```
 */
export default class DGBase extends EventTarget {
  /**
   * Should we display debug information
   * @type {boolean}
   * @todo Remove me.
   */
  _debug = false;

  /**
   * A map of our event listeners.
   * @type {Map}
   */
  _listeners = new Map();

  /**
   * Return the map of event listeners.
   * @type {Map}  - The map of event listeners.
   */
  get listeners() { return this._listeners; }

  /**
   * The default constructor.
   * @constructor
   * @param {boolean} debug - A boolean flag indicating whether or not we need to debug the
     object.
   * @throws DGAbstractInstantiationError
   */
  constructor( debug = false ) {
    super(); // Initialize event target
    this._debug = debug;
    let name = this.constructor.name;
    if( name.endsWith( 'Base' ) ) throw new DGAbstractInstantiationError( name );

    this._listeners.set( "*", [] );
  }

  /**
   * Destroy the object and free any internal resources, notifying any listeners so they may
```

```
   clear any references to this instance. Subclasses *must* call `super.destroy()` before
   removing any used resources.
 */
destroy() {
  // Tell the world I'm going away.
  this.emit( DGBaseEvent.DESTROY );
}

/**
 * Clone an object, making a shallow copy of the object
 * @param {*} obj - The object to clone.
 * @returns {*} - An object of the same type, copied shallowly.
 */
static clone(obj) {
  return Object.assign(Reflect.construct(Reflect.getPrototypeOf(obj).constructor, []), obj);
}

/**
 * Clone a specific instance, uses DGBase.clone()
 * Subclasses should override this method to provide deep copy functionality if so required.
 * @returns {*} An object of the same type, copied shallowly.
 */
clone() {
  return DGBase.clone(this);
}

/**
 * Determine if two objects have the same type.
 * @param {*} lhs - The object to compare.
 * @param {*} rhs - The object to compare against.
 * @returns {boolean} - A boolean indicating whether the two objects are *exactly* the same
   type.
 */
static isa(lhs, rhs) {
  return lhs.constructor.name === rhs.constructor.name;
}

/**
 * Test to determine if the callee is the same type as `obj`. Uses DGBase.isa(obj,this).
```

```
 * @param {*} obj - The object to compare.
 * @returns {boolean}
 */
isa(obj) {
  return DGBase.isa(obj, this);
}

/**
 * Returns the class constructor for the current class
 * @type {Class}
 */
static get myClass() { return this; }

/**
 * Returns the class constructor for the current instance.
 * @type {Class}  - The class constructor.
 */
get myClass() { return DGBase.myClass; }

/**
 * Convert the object to JSON
 * @returns {{object|*}}
 */
toJSON() { return { class: this.constructor.name }; }

/**
 * Get the JSON representation as a parameter.
 * @returns {string}
 * @constructor
 */
get JSON() { return JSON.stringify(this); }

/**
 * Is the object in debug mode?
 * @return {boolean} - A boolean indicating whether the object is in debug mode.
 */
get debug() { return this._debug; }

/**
```

```
 * Should we debug the object?
 * @param {boolean} debug - A boolean indicating whether the object should be put into debug
   mode.
 */
set debug(debug) { this._debug = debug; }

/**
* Does this class have a particular property?
* @return {boolean} - A boolean indicating whether or not the class has a particular property.
* @return {String} - A string containing the key for the property to be queried.
*/
static hasProperty( obj, property  = '' ) {
  //const prototype = Reflect.getPrototypeOf(obj);
  return (
    Object.hasOwnProperty(obj, property) ||
    Object.hasOwn(obj, property) ||
    property in obj
  );
}

/**
* Does this object have a particular property?
* @return {String} - A string containing the key for the property to be queried.
* @return {boolean} - A boolean indicating whether or not the object has a particular
   property.
*/
hasProperty( property = '' ) { return DGBase.hasProperty( this, property ); }

/**
 * Initialize the object
 * @param {Object} options - A dictionary containing the options required to initialize the
   context.
 * @returns {Promise}  - A promise indicating the class has successfully initialized.
 */
async init( options = {} ) { (options);}

/**
 * Determine if two objects are equal
 * @param {any} x - The first object to compare
```

```
 * @param {any} y - The second object to compare
 */
static objectEquals( x, y ) {
  if( typeof x !== 'object' || typeof y !== 'object' ) return x === y;
  if( x === null || y === null ) return x === y;
  if( x.constructor !== y.constructor ) return false;
  if( x instanceof Function ) return x === y;
  if( x instanceof RegExp ) return x === y;
  if( x === y || x.valueOf() === y.valueOf() ) return true;
  if( Array.isArray(x) && Array.isArray(y) && x.length !== y.length ) return false;
  if( x instanceof Date ) return false;
  if( !( x instanceof Object ) ) return false;
  if( !( y instanceof Object ) ) return false;

  const x1 = x;
  const y1 = y;
  const p = Object.keys(x);
  return Object.keys( y ).every( i => p.indexOf( i ) !== -1 ) && p.every( i =>
   DGBase.objectEquals( x1[ i ], y1[ i ] ) );
}

/**
 * Get listeners for a particular event, and lazily add support for new listeners.
 * @param {string} event - A string describing the event to look for.
 * @returns {Array<CallableFunction|EventListenerObject>}
 */
_listenersForEvent( event ) {
  let listeners = this.listeners;
  if( !listeners.has( event ) ) listeners.set( event, [] );

  return listeners.get( event );
}

/**
 * A map containing all of the objects wrapped in event handlers.
 * @type {Map<Object,CallableFunction}
 */
_wrappedObjects = new Map();
```

```
/**
 * Wrap an EventListenerObject to be used as a callback
 * @param {EventListenerObject} listener - The event handler object
 * @todo Do I really need to do this?
 */
_wrapObjectListener( listener ) {
  if( !this._wrappedObjects.has( listener ) ) {
    this._wrappedObjects.set( listener, (e) => listener.handleEvent.call( listener, e ) );
  }

  return this._wrappedObjects.get( listener );
}

/**
 * Add an event listener.
 * @param {string} event - A string defining the event we're emitting.
 * @param {CallableObject|EventListenerObject} listener - The event handler object or
   callback.
 * @param {boolean} once - true if the listener should only handle the event once.
 */
_addListener( event, listener, once = false, options = {} ) {
  let listeners = this._listenersForEvent( event );
  let toAdd = listener;

  // If we only want to call the listener once, we should wrap it.
  if( once ) {
    let onceWrapper = null;
    switch( typeof listener ) {
      case 'object':
        onceWrapper = ( e ) => {
          listener( e );
          this.off( event, onceWrapper );
        }
        break;
      default:
        onceWrapper = ( e ) => {
          listener( e );
          this.off( event, onceWrapper );
        }
```

```
      break;
    }

    toAdd = onceWrapper;
    options[ once ] = once;
  }

  if( !once ) toAdd = listener;

  // Add the listener to the list
  listeners.push( toAdd );

  // Add the event listener
  this.addEventListener( event, toAdd, options );

  // Allow for method chaining
  return this;
}

/**
 * Remove an event listener.
 * @param {string} event - A string defining the event we're emitting.
 * @param {CallableObject|EventListenerObject} listener - The event handler object or
   callback.
 */
_removeListener( event, listener ) {
  // get all listeners for this event
  let listeners = this.listeners.get( event ) || [];

  // Define a filter
  const filter = ( fn ) => fn === listener;

  // By default, we're not filtering
  let filtered = listeners;

  // Do we want to filter by callback?
  if( listener ) {
    switch( typeof listener ) {
      case 'object': listener = this._wrapObjectListener( listener ); break;
```

```
      default:
        filtered = listeners.filter( filter );
    }
  }

  // Remove the (filtered) event listeners
  filtered.forEach( listener => {
    this.removeEventListener( event, listener );
    listeners.splice( listeners.findLastIndex( filter ), 1 );
  } );

  // Allow for method chaining
  return this;
}

/**
 * In preparation for garbage collection, clear all listeners.
 */
clearListeners() {
  const self = this;
  // clear all of the event listeners
  self.listeners.forEach( ( event, listeners ) => listeners.forEach( ( listener ) =>
   self.removeEventListener( event, listener ) ) );
  this._listeners.clear();

  this._wrappedObjects.clear();
}

/**
 * Emit an event to be handled by all listeners.
 * @param {string} event - A string defining the event we're emitting.
 * @param {any} info - Event details to be passed on to the listener.
 * @todo Support default event types as well, so we can simulate events.
 * @todo Should be async?
 */
emit( event, info = {} ) {
  if( event === '*' ) throw new TypeError( "Cannot emit '*' events!" );

  // Emit the event
```

```
  let custom = new CustomEvent( event, { detail:info } );
  this.dispatchEvent( custom );

  // Emit a '*' event, including the proper details
  if( this._listenersForEvent( '*' ).length > 0 ) {
    info[ 'sourceType' ] = event;
    custom = new CustomEvent( '*', { detail:info } );
    this.dispatchEvent( custom )
  }

  return this;
}

/**
 * Add an event handler for when we receive events.
 * @param {string} event - A string defining the event we're listening for.
 * @param {CallableFunction} callback - The callback to run when the event has occurred.
 * @param {boolean} once - true if we only want to handle the event once, false otherwise.
 */
on( event, listener, once = false, options = {} ) {
  switch( typeof listener ) {
    case 'object':
      // Only objects which have the handleEvent property can be passed
      if( !Reflect.has( listener, 'handleEvent' ) )
        throw new TypeError( `Object ${listener.constructor.name} has no handleEvent
   property!` );

      this._addListener( event, this._wrapObjectListener( listener ), once, options );
      break;

    case 'function':
      this._addListener( event, listener, once );
      break;

    default:
      throw new TypeError( `Invalid event listener type: ${typeof listener}!` );
  }

  return this;
```

```
  }

  /**
   * Add an event handler for when we receive events. These events are only handled once, and
     then removed.
   * @param {string} event - A string defining the event we're listening for.
    * @param {CallableFunction} callback - The callback to run when the event has occurred.
   */
  once( event, listener ) {  return this.on( event, listener, true ); }

  /**
   * Disable any event handlers for this event
   * @param {Event} event - The event we want to stop listening for.
   * @param {CallableFunction} callback - A callback we'd like to filter out.
   */
  off( event, listener ) { return this._removeListener( event, listener ); }

  /**
   * A basic event handler.
   * @param {Event} event - The event to handle.
   */
  handleEvent( event ) {
    // Get the key, or the source event, if we're a '*' event.
    const key = ( event.type === '*'  && event.detail.sourceType ) ? event.detail.sourceType :
     event.type;
    try {
      this[ 'on' + key ]( event );
    } catch( e ) {
      console.error( `Received unhandled ${key} event!`, event )
      console.info( e )
    }
  }
}
```

C.1.2 DGTestRig

Listing C.2. Javascript code for `DGTestRig`.

```
/**
 * @file /src/common/test/test.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the GPU test rig for Dynamical Graph.
 *
 * @created 28 July 2022
 */

import DGBase from '../base.js'

export default class DGTestRig {

  static assertThrows( fn, message ) {
    try {
      fn();
      console.error( `Function failed to throw: ${message}!` );
      return false;
    } catch( e ) {
      console.log( `fn threw exception: ${e}` )
      return true;
    }
  }

  static assertTrue( test, message ) {
    if( !(test) ) {
      console.error( `Assertion Failed! ${message}`, test );
      return false;
    }

    return true;
  }

  static assertFalse( test, message ) {
    if( (test) ) {
      console.error( `Assertion Failed! ${message}`, test );
      return false;
```

```
    }

    return true;
  }

  static assertEQ( a, b, message ) {
    if( !DGBase.objectEquals( a, b ) ) {
      console.error( `Assertion Failed! ${message}`, a, b )
      return false
    }

    return true;
  }

  static assertNEQ( a, b, message ) {
    if( DGBase.objectEquals( a, b ) ) {
      console.error( `Assertion Failed! ${message}`, a, b )
      return false
    }

    return true;
  }

  static assertInRange( value, min, max, message ) {
    if( value < min || value > max ) {
      console.error( `Assertion Failed: ${message} ${value} <> (${min}, ${max}!` )
      return false
    }

    return true
  }

  /**
   * Run all functions whose name starts with "test_"
   * @todo Replace with a function generator!
   * @todo Parse the *TestRig name and create a new group, so we can add multiple tests to a
     single module.
   */
  static async run() {
```

```
    const asyncEvery = async ( array, predicate ) => {
      for( let item of array ) {
        if( !await predicate( item ) ) return false;
      }
      return true;
    };

    const needle = /^test_/g;
    const tests = Reflect.ownKeys( this )
      .filter( ( key ) => key.match( needle ) )

    await asyncEvery( tests, async ( test ) => {
        console.group( test )
        console.log( `Running test "${test.replace( needle, '').replace( /_/g, ' ')}"` )
        return this[ test ]()
          .then( ( ok ) => {
            ( ok ) ? console.log( 'Passed!' ): console.error( 'Failed!' );

            return ok;
          } )
          .finally( () => { console.groupEnd(); } );

      } )
      .then( ( ok ) => {
        ( ok ) ? console.log( 'All tests passed!' ): console.error( 'Some tests failed!' );

        return ok;
      } )

  }
}
```

C.1.3 DGGPUEngineTestRig

Listing C.3. Javascript code for `DGGPUEngineTestRig`.

```
/**
 * @file /src/common/gpu/test/gpu.engine.test.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the WebGPU[Compute/Render]Engine test rig for Dynamical
    Graph.
 *
 * @created 28 July 2022
 */

 /* global GPUMapMode */

import DGTestRig from '../../test/test.js';

import DGUtil from '../../util/util.js';
import DGWebGPUComputeEngineValidation from './compute.js';

export default class DGGPUEngineTestRig extends DGTestRig {
  static computeEngine = null;

  static async test_compute_engine() {
    // Build the engine
    return DGWebGPUComputeEngineValidation.build()
      // Execute the shader
      .then( ( engine ) => {
        DGGPUEngineTestRig.computeEngine = engine;
        return engine;
      } )
      // Get the output
      .then( ( engine ) => {
        const buffer = engine.bufferForKey( 'result' );
        return buffer.mapAsync( GPUMapMode.READ, 0, engine.bufferSize );
```

```
      } )
      // Do the test
      .then( () => {
        const engine = this.computeEngine;
        const buffer = engine.bufferForKey( 'result' );
        const expected = new Uint32Array( [ ...DGUtil.iterRange( engine.elementCount, x => x +
    engine.offset ) ] );
        const arrayBuffer = buffer.getMappedRange( 0, engine.bufferSize );
        const actual = new Uint32Array( arrayBuffer );

        return this.assertEQ( expected, actual, "values not equal!" );
      } )
      .catch( ( e ) => {
        console.error( e );

        return false;
      } );
  }
}

document.addEventListener( 'DOMContentLoaded', () => { DGGPUEngineTestRig.run() } );
```

C.1.4 DGAsyncLoop

Listing C.4. Javascript code for the `DGAsyncLoop` Interface.

```
/**
 * @file /src/common/adt/asyncLoop.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the Asynchronous Loop interface for Dynamical Graph.
 *
 * @created 28 July 2022
 */

/**
  * Adds an async version of the 'map' method.
  * @param {AsyncFunction} callback - An asynchronous function callback.
  * @param {Array|Object} thisArg - The array/object to be used as 'this'
  * @returns An array
  */
Array.prototype.asyncMap = async function( callback, thisArg = this ) {
  return Promise.all( thisArg.map( async ( e, i, a ) => await callback( e, i, a ) ) );
}

/**
  * Adds an async version of the 'forEach' method. An alias for asyncMap()
  * @param {AsyncFunction} callback - An asynchronous function callback.
  * @param {Array|Object} thisArg - The array/object to be used as 'this'
  */
Array.prototype.asyncForEach = async function( callback, thisArg = this ) {
  return thisArg.asyncMap( callback, thisArg );
};

/**
  * Adds an async version of the 'filter' method.
  * @param {AsyncFunction} callback - An asynchronous function callback.
  * @param {Array|Object} thisArg - The array/object to be used as 'this'
```

```
  */
Array.prototype.asyncFilter = async function( predicate, thisArg = this ) {
  return thisArg.asyncMap( predicate, thisArg )
    .then( ( result ) => thisArg.filter( ( _, i ) => result[i] ) )
}

/**
  * Adds a parallel async version of the 'reduce' method.
  * @param {AsyncFunction} callback - An asynchronous function callback.
  * @param {Array|Object} thisArg - The array/object to be used as 'this'
  */
Array.prototype.reduce = async function( callback, thisArg = this, init = 0, ) {
  const reducer = async ( p, e, i, a ) => await callback( p, e, i, a ) ? [ ...await p, e ] : p;
  return thisArg.reduce( reducer, init );
}

/**
  * Adds a sequential async version of the 'filter' method.
  * @param {AsyncFunction} callback - An asynchronous function callback.
  * @param {Array|Object} thisArg - The array/object to be used as 'this'
  */
Array.prototype.reduceSequential = async function( callback, init ) {
  const reducer = async ( p, e, i, a ) => [ ...await p, ...await callback( p, e, i, a ) ? [ e ]
    : [] ];
  return this.reduce( reducer, init );
}
```

C.1.5 DGBitVector & DGBitMatrix

Listing C.5. Javascript code for `DGBitVector` & `DGBitMatrix`.

```
/**
 * @file /src/common/adt/bitvector.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the implementation of the bitvector ADT class.
 *
 * @created 17 August 2022
 */

/**
 * An implementation of a bitvector/bitvector ADT.
 */
export class DGBitVector {
  /**
   * Return the number of bits per 'element' of the base array.
   * @type {number}
   */
  static get BITS_PER_ELEMENT() { return 32; }

  /**
   * The default constructor.
   * @constructor
   * @param {number} count - The number of bits in the bitvector.
   */
  constructor( count = 64 ) {
    if( typeof count !== 'number' )
      throw new TypeError( `Attempt to create bitset with invalid type ${typeof count}!` );

    // Sanitize
    this._length = Math.floor( count );
    const elementCount = Math.ceil( this._length / DGBitVector.BITS_PER_ELEMENT );
```

```
  // Create the bit array
  this._bits = new Uint32Array( elementCount );
}

/**
 * The length of the bitvector.
 * @type {number}
 */
_length = 64;

/**
 * Convenience method for sanitizing the index.
 * @param {number} index - The bit index.
 * @returns {number} The sanitized index value.
 * @throws TypeError if index is not a numeric type.
 * @throws RangeError if index is out of bounds.
 */
_checkIndex( index ) {
  if( typeof index !== 'number' )
    throw new TypeError( `invalid index type ${typeof index}!` );

  if( index >= this._length )
    throw new RangeError( `index out of range: ${index} > ${this._length}!` );

  return index;
}

/**
 * Return the length of the bitvector.
 * @type {number} - The length of the bitvector (in bits).
 */
get length() { return this._length; }

/**
 * Get the value of the bit at the specified index.
 * @param {number} index - The bit index.
 */
get( index ) { return this._get( index ); }
```

```
/**
 * A convenience method so the iterator works for subclasses as well.
 * @param {number} index - The index.
 */
_get( index ) {
  index = this._checkIndex( index );

  return ( this._bits[ Math.floor( index / DGBitVector.BITS_PER_ELEMENT ) ] & ( 1 << ( index
   % DGBitVector.BITS_PER_ELEMENT ) ) ) != 0;
}

/**
 * Set the bit at the specified index.
 * @param {number} index - The bit index.
 */
set( index, value = true ) {
  index = this._checkIndex( index );

  // set the bit
  if( value )
    this._bits[ Math.floor( index / DGBitVector.BITS_PER_ELEMENT ) ] |= ( 1 << ( index %
   DGBitVector.BITS_PER_ELEMENT ) );
  else
    this._bits[ Math.floor( index / DGBitVector.BITS_PER_ELEMENT ) ] &= ~( 1 << ( index %
   DGBitVector.BITS_PER_ELEMENT ) );

  // allow method chaining
  return this;
}

/**
 * Clear the bit at the specified index.
 * @param {number} index - The bit index.
 * @throws {RangeError|TypeError}
 */
clear( index ) { return this.set( index, false ); }

/**
 * An iterator for scanning each of the bits.
```

```
   * @type {Iterator}
   */
  [ Symbol.iterator ]() {
    let cursor = 0;
    const vec = this;
    const iterator = {
      next() {
        let done = !( cursor < vec.length );
        return {
          value: done ? undefined : vec._get( cursor++ ),
          done: done
        };
      }
    };

    return iterator;
  }

  /**
   * Print the object to a string, by looping through all of the bits and output.
   */
  toString() { return '0b' + [ ...this ].map( x => x ? 1 : 0 ) .join( '' ); }
}

export default DGBitVector;

/**
 * Class representing a square matrix of bits.
 * @extends DGBitVector
 * @inheritdoc
 */
export class DGBitMatrix extends DGBitVector {
  constructor( rows ) {
    super( rows * rows );

    this._size = { rows: rows, columns: rows };
  }

  /**
```

```
 * The size of the matrix: r,c.
 * @type {Object}
 */
_size = {
  rows: 1,
  columns: 1,
};

/**
 * The size of the matrix: r,c.
 * @type {Object}
 */
get size() { return Object.freeze( this._size ); }

/**
 * Calculate the index offset for given row and column indices.
 * @param {number} r - The row index.
 * @param {number} c - The column index.
 * @throws TypeError if `r` or `c` are not number instances.
 * @throws RangeError if `r` >= the number of rows or `c` >= the number of columns.
 */
_checkIndex( r, c ) {
  if( c === undefined ) return super._checkIndex( r );

  if( typeof r !== 'number' || typeof c !== 'number' )
    throw new TypeError( `Invalid index type (${typeof r}${typeof c},!` );

  if( r >= this._size.rows || c >= this._size.cols )
    throw new RangeError( `Index out of range: (${r},${c})!` );

  return r * this._size.rows + c;
}

/**
 * Get the bit value at the given row and column indices.
 * @param {number} r - The row index.
 * @param {number} c - The column index.
 */
get( r, c ) { return super.get( this._checkIndex( r, c ) ); }
```

```
  /**
   * Set the bit value at the given row and column indices.
   * @param {number} r - The row index.
   * @param {number} c - The column index.
   */
  set( r, c, value = true ) { super.set( this._checkIndex( r, c ), value ); return this; }

  /**
   * Clear the bit value at the given row and column indices.
   * @param {number} r - The row index.
   * @param {number} c - The column index.
   */
  clear( r, c ) { super.clear( this._index( r, c ) ); return this; }

  /**
   * Return a string representation of the bitmatrix.
   * @returns {string}
   */
  toString() {
    const r = this._size.rows;
    const c = this._size.columns;
    return [ ...this ].map( ( x, i ) => {
        const s = ( ( i % c ) == ( c - 1 ) ) // Is this the end of a row?
          ? ( i != ( r * c - 1 ) ) ? '\n' : '' // Is this the last item?
          : '';
        return `${x ? 1 : 0}${s}`
      } )
      .join( '' );
  }
}

/**
 * A class representing an upper or lower triangular bit-matrix.
 * @type {DGTriangularBitMatrix}
 * @extends DGBitMatrix
 * @inheritdoc
 */
export class DGTriangularBitMatrix extends DGBitMatrix {
```

```
constructor( rows, upper = false ) {
  super( ( rows * rows ) / 2 );

  this._size = { rows: rows, columns: rows };
  this._upper = upper;
}

/**
 * Return true, if this is an upper triangular matrix, false otherwise.
 * @type {boolean}
 */
_upper = false;

/**
 * Return true, if this is an upper triangular matrix, false otherwise.
 * @type {boolean}
 */
get upper() { return this._upper }

/**
 * Calculate the number of columns for a given row.
 * @param {number} r - The row index.
 */
_cols( r ) {
  return ( this._upper ) ?
    ( this._size.columns - r ) :
    r + 1
}

/**
 * Calculate the index offset for given row and column indices
 * @param {number} r - The row index.
 * @param {number} c - The column index.
 * @throws {RangeError|TypeError}
 */
_checkIndex( r, c ) {
  if( c === undefined ) return super._checkIndex( r );

  if( typeof r !== 'number' || typeof c !== 'number' )
```

```
    throw new TypeError( `Invalid index type (${typeof r}${typeof c},!` );

  if( r > this._size.rows )
    throw new RangeError( `Index out of range: (${r},${c})!` );

  if( this._upper ) {
    if( c < r )
      throw new RangeError( `Index out of range: (${r},${c})!` );
  }
  else if( c > r )
    throw new RangeError( `Index out of range: (${r},${c})!` );

  const m = this;
  const doit = ( r, c ) => {
    const d = m._cols( r - 1 )
    const e = Math.ceil( ( d * ( d + 1 ) ) / 2 )
    return e + c;
  }

  return ( this._upper ) ?
    doit( ( this._size.rows - 1 ) - r, ( this._size.rows - 1 ) - c ) :
    doit( r, c )
}

/**
 * Return a string representation of the bitmatrix.
 * @returns {string}
 */
toString() {
  const r = this._size.rows;
  const p = new Array( r - 1 )
    .fill( ' ' )
    .join( '' );
  let a = [];
  let i = 0, j = 0;

  // This is an upper-triangular matrix.
  if( this._upper ) {
    for( i = 0; i < r; ++i ) {
```

```
        for( j = i; j < r; ++j ) a.push( this.get( i, j ) ? 1 : 0 );
        if( i < ( r - 1 ) ) {
          a.push( '\n' );
          a.push( p.substring( r - ( i + 2 ) ) );
        }
      }
    }
    // This is a lower-triangular matrix.
    else {
      for( i = 0; i < r; ++i ) {
        for( j = 0; j < this._cols( i ); ++j ) a.push( this.get( i, j ) ? 1 : 0 );
        a.push( p.substring( j - 1 ) );
        if( i < ( r - 1 ) ) a.push( '\n' );
      }
    }

    return a.join( '' );
  }
}
```

C.1.6 DGGraphBase

Listing C.6. Javascript code for `DGGraphBase`.

```
/**
 * @file /src/common/adt/graph/base.js
 * @author Rob Dotson
 * @copyright 2020 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the implementation of the abstract DGGraphBase class.
 *
 * @created 28 July 2020
 */

import DGBase from '../../base.js';
import DGRNG  from '../../math/random/base.js';

/**
 * @class DGGraphBase
 * @desc DGGraphBase is an abstract class interface for creating DGGraph* instances.
 */
export class DGGraphBase extends DGBase {
  /**
   * @constructor
   * @param {string} key - A String containing the DGGraphBase instances's key.
   * @param {object} attributes - A key/value pair containing the DGGraphBase instance's
     attributes.
   * @param {boolean} debug - A boolean flag indicating whether or not to debug the DGGraphBase
     instance.
   */
  constructor( key, attributes, debug = false ) {
    super( debug );

    // Set the key
    if( !key || !( ( typeof key === 'string' ) || ( key instanceof String ) ) || key == "" )
      throw new RangeError( `Cannot initialize ${this.constructor.name} with null 'key' field!`
    );
```

```
  this._key = key;

  // Set the attributes
  this._attributes = attributes || {};

  // Set the time
  this._attributes[ 'birthdate' ] = performance.now();
}

/**
 * Get the specified key from a DGGraphBase subclass instance.
 * @param {DGGraphBase} input - The subclass instance.
 * @todo rename makeKey()
 */
static getKey( input ) {
  switch( typeof input ) {
    case 'string': break;
    case 'object':
      if( input instanceof DGGraphBase ) { input = input.key; break; }
      // fallsthrough
    default:
      throw new TypeError( `Source element has an invalid type: ${input.constructor.name}!` );
  }

  return input;
}

/**
 * Generate a default key for the specified instance.
 * @returns {string}
 */
static makeKey() { return [
            this.prototype.constructor.name.replace(/^DG/g, ''),
            DGRNG.Uint32.toString(16)
          ].join( '-' );
}

/**
```

```
    * A string uniquely identifying the instance.
    * @type {string}
    */
   _key = "";

   /**
    * Return the instance's key.
    * @type {string}  - The key.
    */
   get key() { return this._key; }

   /**
    * An object containing the instances attributes.
    * @type {Object}
    */
   _attributes = {};

   /**
    * Return the instance's attributes.
    * @type {Object}  - The attributes.
    * @todo Move attributes to attributable interface, or remove.
    */
   get attributes() { return this._attributes; }

   /**
    * Return the age of the graph element.
    * @type {Number}  - The birthdate of the materialization in milliseconds.
    */
   get birthdate() { return this._attributes[ 'birthdate' ]; }

   /**
    * Return the age of the graph element.
    * @type {Number}  - The age of the graph element in milliseconds.
    */
   get age() { return performance.now()  - this._attributes[ 'birthdate' ]; }
}

export default DGGraphBase;
```

C.1.7 DGGraph

Listing C.7. Javascript code for `DGGraph`.

```
/**
 * @file /src/common/adt/graph/graph.js
 * @author Rob Dotson
 * @copyright 2020 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the implementation of the concrete DGGraph class.
 *
 * @created 28 July 2020
 */

import DGGraphBase            from './base.js';
import DGQuickBloom           from '../bloom.js';
import DGGraphNode            from './node.js';
import {DGGraphEdge,
    DGGraphDirectedEdge}      from './edge.js';
import DGGraphSet           from '../set.js';
import DGStack              from '../stack.js';
import DGQueue              from '../queue.js';
import {DGUnimplementedFeatureError,
    DGUnsupportedFeatureError}    from '../../error/error.js';
import '../array.js';

/**
 * An object containing "enum" strings for graph events.
 * @type {Object}
 */
export const DGGraphEvent = Object.freeze({
  CLEARED       : 'graph_cleared',
  EDGE_ADDED      : 'graph_edge_added',
  EDGE_DELETED    : 'graph_edge_deleted',
  EDGE_MERGED     : 'graph_edge_merged',
  EDGE_REPLACED   : 'graph_edge_replaced',
  EDGES_CLEARED   : 'graph_edges_cleared',
```

```
  NODE_ADDED      : 'graph_node_added',
  NODE_DELETED    : 'graph_node_deleted',
  NODE_MERGED     : 'graph_node_merged',
  NODE_REPLACED   : 'graph_node_replaced',
  NODES_CLEARED   : 'graph_nodes_cleared',
  MATERIALIZE_BEGIN : 'graph_materialize_begin',
  MATERIALIZE_END   : 'graph_materialize_end',
  MATERIALIZE_ERROR : 'graph_materialize_error',
});

/**
 * A concrete class implementation encapsulating the graph ADT.
 * @extends DGGraphBase
 */
export class DGGraph extends DGGraphBase {
  /**
   * The default constructor.
   * @param {string} key - A string to use to uniquely identify a graph.
   * @param {Object} data - An object containing two arrays: { vertices : <Array|TypedArray>,
     edges : <Array|TypedArray> }
   * @param {Object} attributes - An object containing the attributes to apply to the entire
     graph.
   * @param {boolean} debug - true if we want to debug this object, false otherwise.
   */
  constructor( key = DGGraph.makeKey(), attributes = {}, debug = false ) {
    super( key, attributes, debug );
  }

  /**
   * Convenience getter for graph events.
   * @type {Object}
   */
  static events = Object.freeze( DGGraphEvent );

  /**
   * Clear the graph
   */
  clear() {
    this
```

```
    ._clearEdges()
    ._clearNodes()
    ._clearMaterialization()

    this.emit( DGGraphEvent.CLEARED, this );
  }

  /**
   * The attributes for the graph
   * @type {Object}  - A key-value store containing attributes for this graph.
   */
  get attributes() { return this._attributes; }

  /**
   * Determines if the graph allows self-looping.
   * @type {boolean}  - True, if self-loops are allowed, false otherwise.
   */
  get allowsSelfLoops() { return this._attributes[ 'allowsSelfLoops' ] || true; }

  /**
   * Determines if the graph is directed or not.
   * @type {Boolean}  - True, if the graph is directed, false otherwise.
   */
  get directed() { return this._attributes[ 'directed' ] || false; }

  /**
   * Determines whether or not the graph is a multigraph or not.
   * @type {boolean}  - True, multiple edges between the same nodes is allowed, false otherwise.
   */
  get multiGraph() { return this._attributes[ 'multiGraph' ] || false; }

  /**
   * Return true if the graph is complete, false otherwise.
   * @type {boolean}
   */
  get complete() {
    const e = [ ...this._nodes.values() ]
      .map( n => n.degree( this.directed, this.allowsSelfLoops ) )
      .reduce( ( prev, current ) => prev + current, 0 );
```

```
  const n = this.order;

  return ( e / 2 ) == ( n * ( n - 1 ) / 2 );
}

/**
 * The map of all edges attached to this graph.
 * @type {Map<string,DGGraphEdge>}
 */
_edges = new Map();

/**
 * Returns the size of the graph.
 * @type {number}  - The number of edges defined for this graph.
 */
get size() { return this._edges.size; }

/**
 * The map of all edges attached to this graph.
 * @type {Map<string,DGGraphEdge>}
 */
get edges() { return this._edges; }

/**
 * Return the maximum number of edges that could be defined for this graph.
 * @type {number}  - The maximum number of edges.
 */
get maxEdges() {
  const n = this.order;
  let   m = n - 1 + this.directed + this.allowsSelfLoops;
  return ( n * m ) / 2;
}

/**
 * Some extensible edge operations.
 * @type {Object}
 */
static _edge_ops = Object.freeze( {
  CHECK   : 0,
```

```
  OVERWRITE : 1,
  MERGE   : 2,
} );

/**
 * Some extensible edge operations.
 * @type {Object}
 */
static get EDGE_OP() { return this._edge_ops; }

/**
 * Add an edge with the specified parameters to the graph.
 * @param {DGGraphNode|string} source - The source node.
 * @param {DGGraphNode|string} target - The target node.
 * @param {Object} attributes - Attributes to use when creating the edge.
 * @param {number} op - The operation to perform when adding the edge: 0 = check, 1 =
   overwrite, 2 = merge
 * @throws {DGUnsupportedFeatureError|RangeError|DGUnimplementedFeatureError}
 */
addEdge( source, target, attributes = { directed:this.directed }, op = DGGraph.EDGE_OP.CHECK
   ) {
  // Does the graph allow self loops?
  if( source == target && !this.allowsSelfLoops )
    throw new DGUnsupportedFeatureError( `${this.key} does not allow self loops!` );

  // Cache the nodes so we don't call .has() and .get() repeatedly
  source = this._nodes.get( DGGraph.getKey( source ) );
  target = this._nodes.get( DGGraph.getKey( target ) );

  // Does the source node exist?
  if( !source ) throw new RangeError( `Source node '${source}' doesn't exist!` );

  // Does the destination node exist?
  if( !target ) throw new RangeError( `Destination node '${target}' doesn't exist!` );

  // Do the directed attributes match?
  if( ( attributes.directed || this.directed ) != this.directed )
    throw new RangeError( `Attempt to add ${ attributes.directed ? '' : 'un'}directed edge to
   ${ this.directed ? '' : 'un'}directed graph!` );
```

```
  // Are we trying to add a self-loop to a graph that disallows them?
  if( source === target && !this.allowsSelfLoops )
    throw new RangeError( `Attempt to add self-loop to graph that disallows them!` );

  // Create the edge
  //! @todo - should I also insert the edge with the opposite key for undirected graphs?
  let edge = ( attributes.directed )
    ? new DGGraphDirectedEdge( source, target, attributes )
    : new DGGraphEdge( source, target, attributes );

  // Get the old edge, if it exists
  const old = this._edges.get( edge.key );

  //! @todo we should make it easy for subclasses to check for this edge, and also check the
   inverse, [b-a]
  //! @todo to support multi-graphs, we should check if it exists, and if so, replace the
   current edge with an array.
  switch ( op ) {
    // Check to see if the edge exists.
    case DGGraph.EDGE_OP.CHECK: return this._edgeCheckAndSet( old, edge );

    // Overwrite the existing edge.
    case DGGraph.EDGE_OP.OVERWRITE: return this._edgeOverwrite( old, edge );

    // Merge the edges.
    case DGGraph.EDGE_OP.MERGE: return this._edgeMerge( old, edge );

    // Invalid operation
    default: throw new DGUnsupportedFeatureError( `Invalid edge op: ${op}!` );
  }
}

_edgeCheckAndSet( prev, next ) {
  if( prev )
    throw new RangeError( `Edge with key '${next.key}' exists!` );

  // add the node
  this._edges.set( next.key, next );
```

```
  // notify anyone who cares
  this.emit( DGGraphEvent.EDGE_ADDED, { graph: this, edge: next } );

  // method chaining
  return this;
}

_edgeOverwrite( prev, next ) {
  // set the new node with that key
  this._nodes.set( next.key, next );

  // notify anyone who cares
  if( prev ) this.emit( DGGraphEvent.EDGE_REPLACED, { graph: this, edge: prev } );

  // method chaining
  return this;
}

_edgeMerge( prev, next ) { (prev,next);
  throw new DGUnimplementedFeatureError( 'Edge merging is not yet supported!' );
}

/**
 * Determine whether or not the graph has an edge connecting these two nodes.
 * @param {DGGraphNode|string} source - The source node or its key.
 * @param {DGGraphNode|string} target - The target node or its key.
 * @returns {boolean}  - True if the graph has the specified key, false otherwise.
 */
hasEdge( source, target ) {
  // sanitize
  source = DGGraph.getKey( source );
  target = DGGraph.getKey( target );

  const keys = ( this.directed )
    ? [ DGGraphEdge.makeKey( source, target, true ), undefined ]
    : [ DGGraphEdge.makeKey( source, target, false ), DGGraphEdge.makeKey( target, source,
   false ) ];
```

```
  // If we're not directed, the node could have been stored in either direction.
  return this._edges.has( keys[ 0 ] ) || this._edges.has( keys[ 1 ] );
}

/**
 * Retrieve the edge connecting the specified nodes.
 * @param {DGGraphNode} source - The source node.
 * @param {DGGraphNode} target - The target node.
 * @returns {DGGraphEdge}
 * @throws {RangeError}
 */
getEdge( source, target ) {
  // sanitize
  source = DGGraph.getKey( source );
  target = DGGraph.getKey( target );

  const keys = ( this.directed )
    ? [ DGGraphEdge.makeKey( source, target, true ), undefined ]
    : [ DGGraphEdge.makeKey( source, target, false ), DGGraphEdge.makeKey( target, source,
   false ) ]

  const edge = this.edgeForKey( keys[ 0 ] ) || this.edgeForKey( this._edges.get( keys[ 1 ] )
   );

  if( !edge )
    throw new RangeError( `No edge from ${source} to ${target} exists!` );

  return edge;
}

/**
 * Find the edge with the specified key
 * @param {string} key - A key identifying the edge to locate.
 * @returns {DGGraphEdge|undefined}
 */
edgeForKey( key ) { return this._edges.get( key ); }

/**
 * Clear all of the edges in the graph.
```

```
 */
_clearEdges() {
  this._edges.clear();
  this.emit( DGGraphEvent.EDGES_CLEARED, this );

  return this;
}

/**
 * Remove an edge from the graph instance.
 * @param {DGGraphEdge|string} edge - The edge to remove.
 */
edgeRemove( edge ) {
  switch( typeof edge ) {
    case 'object': break;
    case 'string':
      if( ( edge = this.edgeForKey( edge ) ) ) break;
      // fallthrough
    default:
      throw new TypeError( `Invalid edge or key!` );
  }

  // Clear the edge and remove it.
  this._edges.delete( edge.clear().key );

  // Notify everyone
  this.emit( DGGraphEvent.EDGE_DELETED, edge );
}

/**
 * Filter edges by arbitrary criteria.
 * @param {CallableFunction} filter - The filter to use when selecting edges.
 * @returns {Array<DGGraphEdge>}
 */
edgeFilter( filter ) {
  if( typeof filter !== 'function' )
    throw new TypeError( `Invalid filter!` );

  return [ ...this._edges.values() ].filter( filter );
```

```
}

/**
 * A map of all of the vertices (nodes) contained in the graph.
 * @type {Map<string,DGGraphNode>}
 */
_nodes = new Map();

/**
 * Return the number of vertices in the graph instance.
 * @type {number}  - The number of vertices in the graph instance.
 */
get order() { return this._nodes.size; }

/**
 * Return the set of vertices.
 * @type {Map<string,DGGraphNode>}  - The nodes attached to this graph.
 */
get nodes() { return this._nodes; }

/**
 * Basic node operations to determine what to do when adding nodes.
 * @type {Readonly<Object>}
 */
static _node_ops = Object.freeze( {
  CHECK: 0,
  OVERWRITE: 1,
  MERGE: 2,
} );

/**
 * Basic node operations to determine what to do when adding nodes.
 * @type {Readonly<Object>}
 */
static get NODE_OP() { return this._node_ops; }

/**
 * Add a node to the graph.
 * @param {DGGraphNode} node - The node to add.
```

```
 * @param {number} op - The operation to perform on the node: 0 = check, 1 = overwrite, 2 =
   merge
 */
nodeAdd( node, op = DGGraph.NODE_OP.CHECK ) {
  if( !( node instanceof DGGraphNode ) )
    throw new TypeError( `Attempt to add node of invalid type: '${node.constructor.name}'!` );

  const key = node.key;
  // get the old node if it exists
  const old = this._nodes.get( key );

  switch ( op ) {
    // Check to see if the node already exists
    case DGGraph.NODE_OP.CHECK:
      if( old )
        throw new RangeError( `Node with key '${key}' exists!` );

      // add the node
      this._nodes.set( key, node );
      // notify anyone who cares
      this.emit( DGGraphEvent.NODE_ADDED, { graph: this, node: node } );
      break;

      // Overwrite any node(s) that already exist.
    case DGGraph.NODE_OP.OVERWRITE:
      // set the new node with that key
      this._nodes.set( key, node );
      // notify anyone who cares
      if( old ) this.emit( DGGraphEvent.NODE_REPLACED, { graph:this, previous:old,
   current:node } );
      break;

      // Merge the new and old nodes together.
    case DGGraph.NODE_OP.MERGE:
      throw new DGUnimplementedFeatureError( 'Node merging is not yet supported!' );

      // We shouldn't get here.
    default:
      throw new DGUnsupportedFeatureError( `Invalid node op: ${op}!` );
```

```
  }

  return this;
}

/**
 * Remove a node from the graph instance.
 * @param {DGGraphNode|string} edge - The node to remove.
 */
nodeRemove( node ) {
  node = this.nodeGet( node );

  // Get any edges linked to this node, and remove them.
  this._edges
    .filter( ( edge ) => edge.source === node || edge.target == node )
    .forEach( ( edge ) => this.edgeRemove( edge ) )

  // Remove the node.
  this._edges.delete( node.key );

  // Notify everyone
  this.emit( DGGraphEvent.NODE_DELETED, node );
}

/**
 * Get a node, ensuring that it is a valid node type.
 * @param {DGGraphNode|string} node - The node to retrieve
 */
nodeGet( node ) {
  switch( typeof node ) {
    case 'string': node = this.nodeForKey( node ); break;
    case 'object': node = this.nodeCheck( node ); break;
    default:
      throw new TypeError( `Invalid node or key!` );
  }

  return node;
}
```

```
/**
 * Retrieve a node with the specified key.
 * @param {string} key - The node key.
 */
nodeForKey( key ) {
  let node;

  if( !( node = this._nodes.get( key ) ) )
    throw new RangeError( `Node with key '${key}' doesn't exist!` );

  return node;
}

/**
 * Check to make sure this node is attached to this graph.
 * @param {DGGraphNode} node - The node to check.
 */
nodeCheck( node ) {
  if( node instanceof DGGraphNode ) {
    if( !this._nodes.has( node.key ) )
      throw new RangeError( `Disconnected node '${node.key}'!` );
  } else
    throw new TypeError( `Invalid node object:${node.constructor.name}!` );

  return node;
}

/**
 * Filter nodes by arbitrary criteria.
 * @param {CallableFunction} filter - The filter to use when selecting nodes.
 * @returns {Array<DGGraphNode>}
 */
nodeFilter( filter ) {
  if( typeof filter !== 'function' )
    throw new TypeError( `Invalid filter!` );

  return [ ...this._nodes.values() ].filter( filter );
}
```

```
/**
 * Return an array of nodes attached to the graph with an order > 0.
 * @type{Array<DGGraphNode>}
 */
connectedNodes() {
  const filter = ( node ) => { return node.degree( this.directed, this.allowsSelfLoops ) > 0;
   }
  return this.nodeFilter( filter );
}

/**
 * Return an array of disconnected nodes attached to the graph.
 * @type{Array<DGGraphNode>}
 */
disconnectedNodes() {
  const filter = ( node ) => { return node.degree( this.directed, this.allowsSelfLoops ) ==
   0; }
  return this.nodeFilter( filter );
}

/**
 * Clear all of the nodes from the graph.
 */
_clearNodes() {
  this._nodes.clear();
  this.emit( DGGraphEvent.NODES_CLEARED, this );

  return this;
}

/**
 * Determine whether two nodes are adjacent.
 * @param {DGGraphNode|String} a - The first node, or its key.
 * @param {DGGraphNode|String} b - The second node, or its key.
 * @returns {boolean}  - True if the two nodes are adjacent, false otherwise.
 * @todo optimize.
 */
isAdjacent( a, b ) {
  try {
```

```
    return this.hasEdge( a, b );
  } catch( e ) {
    console.log( e );
    return false;
  }
}

/**
 * Return a set of neighbors for the specified nodes.
 * @param {DGGraphNode} node - The node to query.
 * @returns {DGGraphSet} - The set of neighbors of the specified node.
 */
neighbors( node ) {
  return DGGraphSet.union( node.sources, node.targets );
}

/**
 * Return the graph density.
 * @todo update for multigraphs
 */
get density() {
  const order = this.order;
  const size  = this.size;
  const d   = ( order * ( order - 1 ) );
  return ( order < 2 )
    ? 0
    : ( this.directed )
      ? size / d
      : ( 2 * size ) / d;
}

/**
 * Return an array of nodes ordered by depth-first search.
 * @param {DGGraphNode|string} node - The node from which to start the search.
 * @param {CallableFunction} - A callback function to call on each visited node: ( node,
  index ) => {}
 * @returns {Array<DGGraphNode>}
 */
dfs( node, visitFunc ) {
```

```
  const start = ( node ) ? this.nodeGet( node ) : this._nodes.keys().next().value;
  let visited = new DGGraphSet();
  let stack = DGStack.from( [ start ] );
  let i   = -1; // Index of the node we're visiting.
  const noop  = ( node, index ) => { console.log( `DFS visit: ${index}`, node.key ); }

  // Set the visit function
  visitFunc   = visitFunc || noop;

  // Loop through all nodes connected to the first.
  while( !stack.empty ) {
    // Pop the current node from the stack
    const node = stack.pop();

    // If we've already visited it, we're done
    if( visited.has( node ) ) continue;

    // Mark the current node as visited
    visited.add( node );

    // Perform the visit function on the node.
    visitFunc( node, ++i );

    // For each target node we haven't visited, push it onto the stack.
    node.targets.forEach( t => { if( !visited.has( t ) ) stack.push( t ); } )
  }

  // Return the array in the order they were visited.
  return [ ...visited ];
}

/**
 * Return an array of nodes ordered by breadth-first search.
 * @param {DGGraphNode} start - The node from which to start the search.
 * @param {CallableFunction} - A callback function to call on each visited node: ( node,
   index ) => {}
 * @returns {Array<DGGraphNode>}
 */
bfs( node, visitFunc ) {
```

```
  const start = ( node ) ? this.nodeGet( node ) : this._nodes.keys().next().value;
  let visited = new DGGraphSet();
  let queue   = DGQueue.from( [ start ] );
  let i       = -1; // Index of the node we're visiting.
  const noop    = ( node, index ) => { console.log( `BFS visit: ${index}`, node.key ); }

  // Set the visit function
  visitFunc = visitFunc || noop;

  while( !queue.empty ) {
    // Remove the current node from the queue
    const node = queue.dequeue();

    // If we've already visited this node, we're done
    if( visited.has( node ) ) continue;

    // Mark the current node as visited
    visited.add( node );
    visitFunc( node, ++i );

    // For each target node we haven't visited, push it into the queue.
    node.targets.forEach( t => { if( !visited.has( t ) ) queue.enqueue( t ); })
  }

  // Return the array in the order they were visited.
  return [ ...visited ];
}

_materialization = null;



/**
 * Materialize an array.
 * @param {DGMaterializationStrategy} strategy - The materialization strategy to use.
 * @returns {Promise<boolean>}  - Returns resolving to true if the materialization completed,
   false otherwise.
 */
async materialize( strategy ) {
  // notify the world that we've begun materializing
```

```
    this.emit( DGGraphEvent.MATERIALIZE_BEGIN, this );

    // execute the strategy.
    return strategy.execute( this )
      .then( ( materials ) => { this._materialization = materials; return this; } )
      .then( () => { this.emit( DGGraphEvent.MATERIALIZE_END, this ); return this; } )
      .catch( ( error ) => {
        console.error( error );
        this.emit( DGGraphEvent.MATERIALIZE_ERROR, this );
      } )
      .finally( () => { return this; } )
  }

  /**
   * Clear the materialization if it exists.
   */
  _clearMaterialization() {
    this._materialization.destroy();
    this._materialization = null;
  }
}

export default DGGraph;

/**
 * An implementation of an undirected multigraph ADT.
 * @extends DGGraph
 * @inheritdoc
 */
export class DGMultiGraph extends DGGraph {
  /**
   * The default constructor.
   * @param {string} key - A string to use to uniquely identify a graph.
   * @param {Object} data - An object containing two arrays: { vertices : <Array|TypedArray>,
     edges : <Array|TypedArray> }
   * @param {Object} attributes - An object containing the attributes to apply to the entire
     graph.
   * @param {boolean} debug - true if we want to debug this object, false otherwise.
   */
```

```
  constructor( key = DGGraph.makeKey(), attributes = {}, debug = false ) {
    super( key, attributes, debug );

    this._attributes[ 'multiGraph' ] = true;
  }
}

/**
 * An implementation of a directed graph ADT.
 * @extends DGGraph
 * @inheritdoc
 */
export class DGDirectedGraph extends DGGraph {
  /**
   * The default constructor.
   * @param {string} key - A string to use to uniquely identify a graph.
   * @param {Object} data - An object containing two arrays: { vertices : <Array|TypedArray>,
     edges : <Array|TypedArray> }
   * @param {Object} attributes - An object containing the attributes to apply to the entire
     graph.
   * @param {boolean} debug - true if we want to debug this object, false otherwise.
   */
  constructor( key = DGGraph.makeKey(), attributes = {}, debug = false ) {
    super( key, attributes, debug );

    this._attributes[ 'directed' ] = true;
  }
}

/**
 * An implementation of an directed multigraph ADT.
 * @extends DGDirectedGraph
 * @inheritdoc
 */
export class DGMultiDirectedGraph extends DGDirectedGraph {
  /**
   * The default constructor.
   * @param {string} key - A string to use to uniquely identify a graph.
   * @param {Object} data - An object containing two arrays: { vertices : <Array|TypedArray>,
```

```
     edges : <Array|TypedArray> }
   * @param {Object} attributes - An object containing the attributes to apply to the entire
     graph.
   * @param {boolean} debug - true if we want to debug this object, false otherwise.
   */
  constructor( key = DGGraph.makeKey(), attributes = {}, debug = false ) {
    super( key, attributes, debug );

    this._attributes[ 'multiGraph' ] = true;
  }
}
```

C.1.8 DGQuickBloom

Listing C.8. Javascript code for `DGBloomFilterBase` & `DGQuickBloom`.

```
/**
 * @file /src/common/adt/set.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the implementation of the abstract DGGraphLayoutBase class.
 *
 * @created 17 August 2022
 *
 * @cite https://llimllib.github.io/bloomfilter-tutorial/
 */

import DGUtil   from '../util/util.js';
import DGHash   from '../math/hash.js';
import DGBitVector  from './bitvector.js';

/**
 * An abstract DGBloomFilter class.
 */
  export class DGBloomFilterBase {
  /**
   * The default false positive rate.
   * @type {number}
   */
  static get DEFAULT_FPR() { return .001; }

  /**
   * Create a new bloom filter.
   * @constructor
   * @param {number} n - The maximum capacity of the bloom filter.
   * @param {number} fpr - The false positive rate.
   */
  constructor( n, fpr = DGBloomFilterBase.DEFAULT_FPR ) {
```

```
  // Sanitize
  n = Math.floor( n );
  n = ( n < 1 ) ? 1 : n;

  const bitsPerFilter = DGBloomFilterBase.bitsForFPR( n, fpr );

  this._bitvector = new DGBitVector( bitsPerFilter );
  this._hashCount = DGBloomFilterBase.hashesForFPR( n, fpr, bitsPerFilter );
  this._hashes = [];
}

/**
 * Calculate the number of bits to achieve the specified fpr (False Positive Rate) for a
   given number of values.
 * @param {number} n - The maximum number of values to store in the bloom filter.
 * @param {number} fpr - The desired false positive rate.
 * @returns {number}
 */
static bitsForFPR( n, fpr ) {
  let m = Math.pow( 2.0, Math.log( 2.0 ) );
  let nlp = n * Math.log( fpr );
  m = Math.log( 1.0 / m );
  m = ( ( m * -nlp ) / m );

  return Math.ceil( m );
}

/**
 * Returns the number of hashes required for the specified fpr.
 * @param {number} n - The number of items indexed by the bloom filter.
 * @param {number} fpr - The desired fpr.
 * @param {number} bits - The number of bits in the bitvector.
 * @returns {number}
 */
static hashesForFPR( n, fpr, bits ) { return Math.ceil( ( bits / n ) * Math.log( 2.0 ) ); }

/**
 * Return the "width" of the bloom filter in bits.
 * @type {number}  - The number of bits in the bloom filter.
```

```
   */
  get width() { return this._bitvector.length; }

  /**
   * Check to see if this function may contain the value we're looking for.
   * @param {any} item - The item to check for.
   * @returns {boolean} - True, if the bloom filter might contain the value, false otherwise.
   */
  mayContain( item ) { (item); /* abstract, is always false. */ return false; }

  /**
   * Add the item to the hash function.
   * @param {any} item - The item to add.
   * @returns {DGBloomFilterBase}
   */
  add( item ) { (item); /* abstract, does nothing */ return this; }

  /**
   * Hash the value
   * @param {TypedArray} value - A TypedArray to hash
   */
  hash( value ) { return this._hashes.map( ( fn ) => fn( value ) % this._bitvector.length ); }

  /**
   * Return a string representation of the internal bitvector.
   */
  toString() { return this._bitvector.toString(); }
}

/**
 * A concrete subclass of DGBloomFilterBase which uses the "quick-bloom" algorithm.
 * @type {DGQuickBloom}
 *
 * @cite Adam Kirsch & Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better
    Bloom Filter.
 * @see https://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf
 */
export class DGQuickBloom extends DGBloomFilterBase {
  /**
```

```
 * Create a new bloom filter.
 * @constructor
 * @param {number} n - The number of bits to store in the bloom filter.
 * @param {number} fpr - The false positive rate.
 * @param {Array<CallableFunction> hashes - An array of hashes to use.
 */
constructor( n, fpr = DGBloomFilterBase.DEFAULT_FPR, hashes = [ DGHash.fnv_1, DGHash.murmur ]
   ) {
  super( n, fpr )

  this._hashes = hashes;
}

/**
 * Perform the nth hash.
 * @param {number} n - The current hash round index.
 * @param {number} a - The first hash.
 * @param {number} b - The second hash.
 * @param {number} size - The length of the bitvector.
 */
_hash_n( n, a, b, size ) { return ( a + n * b ) % size; }

/**
 * Gets an iterable range of indices where the hash values could be.
 * @param {number} count - The number of hash rounds to run.
 * @param {Array<u32>} hashes - An array of hashes (for quickbloom, this is 2).
 * @param {number} length - The length of the bit vector
 * @returns {Iterable}
 */
_quickbloom = ( count, hashes, length ) => DGUtil.iterRange( count, ( n ) => this._hash_n( n,
   hashes[ 0 ], hashes[ 1 ], length ) );

/**
 * Add the item to the hash function.
 * @param {any} item - The item to add.
 */
add( item ) {
  // Hash the item
  const hashes = this.hash( item );
```

```
  // How many hash rounds do we need?
  const count = this._hashCount;

  // How many bits are in the bitvector?
  const length = this._bitvector.length;

  // Use the quickbloom algorithm
  for( const index of this._quickbloom( count, hashes, length ) ) this._bitvector.set( index
   );

  // The above is equivalent to:
  // for( let n = 0; n < count; ++n ) {
  //  const index = this._hash_n( n, hashes[ 0 ], hashes[ 1 ], length );
  //  this._bitvector[ index ] = true;
  // }

  // support method chaining.
  return this;
}

/**
 * Check to see if this function may contain the value we're looking for. We actually only
   use this to determine if something *is not* contained within the filter.
 * @param {any} item - The item to check for.
 * @returns {boolean} - True if the bloom filter might contain the value, false otherwise.
 * @todo Optimize this entire function to remove the stack allocation, turning this into a
   single expression.
 */
mayContain( item ) {
  // Hash the item
  const hashes = this.hash( item );
  const count = this._hashCount;
  const length = this._bitvector.length;

  const indexes = this._quickbloom( count, hashes, length );
  const map = [ ...indexes ].map( index => this._bitvector.get( index ) );
  const every = map.every( val => val );
```

```
    return every;
  }
}

export default DGQuickBloom;
```

C.1.9 DGLayoutEngineBase

Listing C.9. Javascript code for `DGLayoutEngineBase`.

```
/**
 * @file /src/layout/base.js
 * @author Rob Dotson
 * @copyright 2020 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the implementation of the abstract DGLayoutEngineBase class.
 *
 * @created 28 July 2020
 *
 * @todo Use static mixins for the registry?
 */

import DGBase from '../common/base.js';
import DGGraph from '../common/adt/graph/graph.js';

/**
 * Enum for layout events.
 * @readonly
 * @enum {string}
 */
export const DGLayoutEvent = Object.freeze({
  LAYOUT_BEGIN: "layout_begin",
  LAYOUT_END:   "layout_end",
  LAYOUT_STEP:  "layout_step",
  LAYOUT_ERROR: "layout_error",
});

/**
 * The abstract class from which all layout engines inherit.
 * @type {DGLayoutEngineBase}
 */
export class DGLayoutEngineBase extends DGBase {
  /**
```

```
 * A registry of valid Random Number Generators.
 * @type {Map<class>}
 */
static #registry = new Map();

/**
 * Return an array of valid shader type names.
 * @type {Array<string>}
 */
static get registeredLayouts() { return Array.from( DGLayoutEngineBase.#registry.keys() ); }

/**
 * Register a layout class with the registry.
 * @param {string} key - A string uniquely identifying a layout.
 * @param {Prototype} value - A layout for a DGLayoutEngineBase subclass.
 */
static _registerLayout( key, value ) { DGLayoutEngineBase.#registry.set( key, value ); }

/**
 * The default constructor.
 * @constructor
 * @param {DGLayoutEngineBase} initialLayoutEngine - The layout engine to use when initially
   laying out the graph.
 * @param {Object} bounds - The boundaries within which the graph will be laid out.
 * @param {boolean} debug - true if we wish to debug the engine, false otherwise.
 *
 * @todo Subclasses must use the same materialization strategy for both this instance and
   initialLayoutEngine.
 */
constructor( graph, initialLayoutEngine, strategy, debug = false ) {
  super( debug )

  // set the materialization strategy
  this._strategy = strategy;

  // sanitize the input
  this._initialLayoutEngine = initialLayoutEngine;

  // set the bounds
```

```
  this._bounds = bounds;
}

/**
 * The layout attributes
 * @type {Object|Map}
 */
_attributes = {};

/**
 * Return the canvas boundaries.
 * @type {Object}  - The canvas boundaries.
 */
get bounds() { return this._attributes.bounds || { w: 256, h: 256 }; }

/**
 * Set the canvas boundaries.
 * @param {Object} bounds - The canvas boundaries
 * @todo Subclasses must sanitize this.
 */
set bounds( bounds ) { this._attributes[ 'bounds' ] = bounds; }

/**
 * The graph to layout.
 * @type {DGGraph}
 */
 _graph = null;

 /**
  * The graph to layout.
  * @type {DGGraph}
  */
 get graph() { return this._graph; }

/**
 * The materialization strategy to use to materialize the graph.
 * @type {DGMaterializationStrategyBase}
 * @todo Subclasses for individual layouts should define the fields used in a materialization.
 */
```

```
 _strategy = null;

 /**
  * Get the materialization strategy.
  * @type {DGLayoutStrategyBase}
  */
  get materializationStrategy() { return this._strategy; }

  /**
   * Set the materialization strategy.
   * @param {DGLayoutStrategyBase} strategy - The materialization strategy to use to
   materialize the graph.
   */
  set materializationStrategy( strategy ) { this._strategy = strategy; }

/**
 * The graph materialization.
 * @type {Array<DGGraphMaterializationBase>}
 */
_materials = null;

/**
 * Return the materialization if it exists, otherwise lazily materialize the graph.
 */
get materials() { return this._materials || this.materialize(); }

/**
 * The initial layout engine to use when laying out this graph.
 * @type {DGLayoutEngineBase}
 */
_initialLayoutEngine = null;

/**
 * Return the initial layout engine.
 * @type {DGLayoutEngineBase}  - The initial layout engine.
 */
get initialLayoutEngine() { return this._initialLayoutEngine; }

/**
```

```
 * Set the initial layout engine.
 * @param {DGLayoutEngineBase} engine - The initial layout engine.
 */
set initialLayoutEngine( engine ) { this._initialLayoutEngine = engine; }

/** BEGIN DGGLOInterface **/

/**
 * Materialize the graph
 */
async materialize() {
  this._materials = await this._strategy.materialize( this.graph );

  return this;
}

/**
 * Merge the passed in layouts with this instance, and return a new layout.
 * @param {Array<DGGraphLayout>|DGGraphLayout} layouts - One or more layouts to merge into
   another.
 */
async mergeLayouts( layouts ) { return this.clone(); }

/**
 * Merge another materialization with this one.
 * @param {DGGraphMaterializationBase} materials - The materials to merge with the current
   instance's.
 */
 async mergeMaterials( materials ) {
   await this._strategy.merge( materials, this.materials )
   return this;
 }

/**
 * Use the initial layout engine to layout the graph.
 * @param {DGGraph} graph - The graph to be laid out.
 * @returns {Promise<DGGraphLayout>}  - Returns a promise which will resolve when the initial
   layout is complete.
 * @todo Subclasses must merge this materialization with the current materialization.
```

```
 */
async initialLayout( graph = this.graph ) {
  return this.initialLayoutEngine.layout( graph );
}

/**
 * Calculate graph statistics.
 * @returns {Promise<DGGraphLayout>}  - A promise which will resolve when the graph's
   statistics have been calculated.
 */
async calculateStatistics( layout ) { return layout; }

/**
 * Calculate graph offsets.
 * @returns {Promise<DGGraphLayout>}  - A promise which will resolve when the graph's offsets
   have been calculated.
 */
async calculateOffsets( layout ) { return layout; }

/**
 * Partition the layout materials, and/or slice into buffer-sized components.
 * @param {DGGraphLayout} layout - The graph layout.
 * @type {AsyncCallableFunction} scatter - A function to partition the layout's materials.
 * @returns {Promise<DGGraphLayout>}  - A promise which will resolve when the materials have
   been scattered.
 */
async partition( layout, scatter = this.scatter ) { return layout._materialsArray = scatter(
   layout.materials ), layout; }

/**
 * Merge the layout materials.
 * @param {DGGraphLayout} layout - The graph layout.
 * @type {AsyncCallableFunction} gather - A function to merge the materials.
 * @returns {Promise<DGGraphLayout>}  - A promise which will resolve when the graph has been
   sliced.
 */
async mergePartitions( layout, gather = this.gather ) { return layout._materials = gather(
   materialsArray ), layout; }
```

```
/**
 * Place a graph's nodes according to the instance's algorithm.
 * @param {DGGraph} graph - The graph.
 * @returns {Promise<DGGraphLayout>}  - A promise which will resolve when the graph has been
   laid out.
 */
async place( layout ) { return layout; }

/**
 * Finesse the graph's nodes into their final positions.
 * @param {DGGraph} graph - The graph.
 * @returns {Promise<DGGraphLayout>}  - A promise which will resolve when the graph has been
   finessed.
 */
async finesse( layout ) { return layout; }

/**
 * Determine if we've reached the terminal condition.
 * @type {boolean}  - True, if we've reached the terminal condition, false otherwise.
 */
get converged( layout ) { return true; }

/**
 * The default stop condition for the loop.
 * @type {AsyncCallableFunction}  - A function to determine whether to exit the loop.
 */
get stop() { return async ( layout ) => { return this.converged( layout ); } }

/**
 * The default partition function.
 * @type {AsyncCallableFunction} - A function to partition the materials.
 */
get scatter() { return async ( materials ) => { return [ materials ] }; }

/**
 * The default merge partition function.
 * @type {AsyncCallableFunction} - A function which merges materials together.
 */
get gather() { return async ( materialsArray ) => { return materialsArray[ 0 ] }; }
```

```
/**
 * The default fitness condition for the layout.
 * @type {AsyncCallableFunction}  - A function which returns the fitness metric for the
   layout.
 */
get fitness() { return async ( layout ) => { return true; } }

/**
 * Loop through the body of the algorithm until the stop condition is met.
 * @param {DGGraph} graph - The graph to layout.
 * @param {Object} context - The event context.
 * @returns {Promise}
 */
async loop( layout, context, stop = this.stop ) {
  while( !await stop( layout ) )
    await this.calculateOffsets( layout )
      .then( this.place( layout ) )
      .then( () => {
        this.emit( DGLayoutEvent.LAYOUT_STEP, context );
      } )

  return layout;
}

/**
 * Layout the graph asynchronously.
 * @returns {Promise<DGGraph>}  - A promise which will resolve when the graph is laid out
   successfully or fails.
 */
async layout( graph = this.graph ) {
  const layout = await this.initialLayout( graph );

  let context = { graph: graph, layout: layout, error: null };
  this.emit( DGLayoutEvent.LAYOUT_BEGIN, context );

  return this.calculateStatistics( layout )
    .then( layout => this.partition( layout ) )
    .then( layout => this.loop( layout, context ) )
```

```
      .then( layout => this.mergePartitions( layout ) )
      .then( layout => this.finesse( layout ) )
      .catch( ( error ) => {
        console.error( error );
        context.error = error;
        this.emit( DGLayoutEvent.LAYOUT_ERROR, context );
      } )
      .finally( () => {
        this.emit( DGLayoutEvent.LAYOUT_END, context );
        return layout;
      } );
  }

  /**
   * Event handler for when graphs change.
   * @param {Event} event - The change event for the graph.
   * @returns {Promise}  - Promise which resolves when the event has been handled.
   */
  async onchange( event ) { ( event ); }
}

export default DGLayoutEngineBase;
```

C.2.1 DGWebGPUBase

All objects requiring access to the WebGPU (GPU Computing for the Web) subsystem are subclasses of the abstract `DGWebGPUBase` class.

Listing C.10. Javascript code for `DGWebGPUBase`.

```
/**
 * @file /src/common/gpu/base.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the abstract base class for all WebGPU-enabled objects.
 *
 * @created 28 July 2022
 *
 * @todo Need to add support for adaptor and device limits.
 */

import DGBase from '../base.js';
import DGGPU from './webgpu.js';
import { DGUnsupportedFeatureError } from '../error/error.js';

/**
 * Enum for WebGPU events.
 * @readonly
 * @enum {string}
 */
export const DGGPUEvent = Object.freeze({
  DESTROY: "destroy",
});

export default class DGWebGPUBase extends DGBase {
  /**
   * The GPU adapter
```

```
 * @type {GPUAdapter}
 */
_adapter;

/**
 * Return the instance's GPUAdapter
 * @type {GPUAdapter}  - The adapter.
 */
get adapter() { return this._parent ? this._parent.adapter : this._adapter; }

/**
 * The GPU Device
 * @type {GPUDevice}
 */
_device;

/**
 * Return the instance's GPUDevice
 * @type {GPUDevice} - The device.
 */
get device() { return this._parent ? this._parent.device : this._device; }

/**
 * The GPU Queue
 * @type {GPUQueue}
 */
_queue;

/**
 * Return the instance's GPUQueue
 * @type {GPUQueue}  - The queue.
 */
get queue() { return this._parent ? this._parent.device.queue : this.device.queue; }

/**
 * The DGWebGPU* instance this object has inherited its features from.
 * @type {DGWebGPUBase} - The parent object
 */
_parent;
```

```
/**
 * Return the instance's parent object.
 * @type {DGWebGPUBase}  - The parent object.
 */
get parent() { return this._parent; }

/**
 * A generic label to be applied to this object.
 * @type {string}
 */
_label = '';

/**
 * Return the label string.
 * @type {string}  - The label string.
 */
get label() { return this._label; }

/**
 * Set the label string
 * @param {string} label - The label string.
 */
set label( label ) { this._label = label; }

/**
 * Construct the DGWebGPUBase object
 * @constructor
 * @param {debug} debug - A boolean indicating if we want debug messages displayed.
 */
constructor( debug = false ) {
  super( debug )
  if( !DGGPU.isSupported )
    throw new DGUnsupportedFeatureError( "WebGPU is not supported!" );
}

/**
 * Initialize the context
 * @method
```

```
 * @async
 * @param {object} options - A list of adapter options. `adaptorOptions` should be a
   GPURequestAdapterOptions value.
 * @throws {DGUnimplementedFeatureError} - If WebGPU is not supported on a browser, this is
   thrown.
 * @throws {InternalError} - Thrown if the WebGPU adapter or device cannot be requested.
 * @todo Rewrite to use promise cascade and properly catch exceptions.
 */
async init( options = {} ) {
  // Do we wish to inherit?
  const inherit = options.inherit || false;
  if( inherit ) {
    this._parent = options.inherit || null;

    // Register for destroy events sent by my parent.
    this._parent.once( 'destroy', this );
  }

  // initialize as normal
  else {
    return super
      .init( options )
      .then( () => {
        // GPURequestAdapterOptions
        const adapterOptions = Reflect.has( options, 'adapterOptions' ) ?
   options.adapterOptions : DGGPU.defaultAdapterOptions;
        // Request the adapter
        return DGGPU.requestAdapter( adapterOptions );
      } )
      .then( async ( adapter ) => {
        // Set the adapter
        this._adapter = adapter;

        // Get the adapter info
        this._adapter[ 'info' ] = await adapter.requestAdapterInfo();

        const features = ["depth-clip-control",
          "depth32float-stencil8",
          "texture-compression-bc",
```

```
          "texture-compression-etc2",
          "texture-compression-astc",
          "timestamp-query",
          "indirect-first-instance",
          "shader-f16",
          "bgra8unorm-storage",
          "rg11b10ufloat-renderable"];
        let supportedFeatures = [];
        features.forEach( ( f, i, a ) => { (i,a);
          if( adapter.features.has( f ) ) supportedFeatures.push( f );
        })


        // GPUDeviceDescriptor
        const requestedFeatures = DGGPU.deviceFeatureNames;

        // Request the device
        return this._adapter.requestDevice( { requestedFeatures } );
      } )
      .then( ( device ) => {
        // Set the device
        this._device = device;

        // Set the device queue
        this._queue = device.queue;

        // Finally return the device
        return device;
      } )
      .catch( error => console.error( error ) )
      .finally( () => { return this; } );
  }

  return this;
}

/**
 * Event handler for 'destroy' events sent by my parents.
 */
```

```
ondestroy() { this.destroy(); }

/**
 * Destroy any GPU resources I may have available.
 */
destroy() {
  // Call super destroy
  super.destroy();

  // Destroy my buffers
  this._destroyBuffers();

  // Destroy my device
  this._destroyDevice();
}

/**
 * Destroy my device.
 */
_destroyDevice() {
  // Don't destroy devices I inherit!
  if( !this._parent ) {
    this.device.destroy();
    this._device = null;
  }
}

/**
 * Destroy any buffers I may have.
 */
_destroyBuffers() {
  this._buffers.forEach( ( buffer ) => buffer.destroy() );
  this._buffers.clear();
  this._buffers = null;
}

/**
 * Build a DGWebGPUSubclass instance and initialize it. Since nearly all GPU* objects require
   asynchronous operation, this is asyncronous.
```

```
   * @todo Find a better way to do this
   * @param {object} options - An object containing objects required to configure this object
   * @param {boolean} debug - A flag indicating whether or not this object should be defined
   * @returns {Promise<DGWebGPUBase>}  - A promise indicating that this object was created
     successfully.
   */
  static async build( options, debug = false ) {
    // the async builder method
    let builder = async () => {
      try {
        return new this( debug );
      }
      catch ( e ) {
        console.error( e )
        return Promise.reject( e )
      }
    }

    return builder( debug )
      .then( ( obj ) => {
        return obj.init( options )
      } )
      .catch( error => {
        console.error( error )

        return null;
      } )
      .finally( () => {} )
  }
}

export { DGWebGPUBase }
```

C.2.2 DGComputeEngine

Listing C.11. Javascript code for `DGComputeEngine`.

```
/**
 * @file /src/common/gpu/compute.js
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains an abstract class for defining WebGPU compute engines.
 *
 * @created 28 July 2020
 */

import '../adt/map.js';
import DGWebGPUBase from './base.js';
import DGWebGPUShaderBase from './shader/base.js';
import DGWebGPUComputeShader from './shader/compute.js';

/* global GPUShaderStage, GPUBuffer */

/**
 * A Compute Engine class.
 * @type {DGWebGPUComputeEngine}
 * @extends DGWebGPUBase
 *
 * @todo Do we need to check valid values of buffer binding types, or leave that to the WebGPU
    subsystem?
 * @todo We need to add binding layout indices to a stored value, so we can get the indices
    back out! I know we also don't wish to foreach, on the array of layout descriptors!
 */

export default class DGWebGPUComputeEngine extends DGWebGPUBase {

  /**
    * The constructor.
    * @constructor
```

```
  * @param {boolean} debug - A flag indicating whether or not we wish to debug.
  */
constructor( debug = false ) { super( debug ) }

/**
 * Initialize the compute engine
 * @param {Object} options - A dictionary containing the options required to initialize the
   context.
 * @returns {Promise}  - A promise indicating the class has successfully initialized.
 */
async init( options = {} ) {
  return super
  .init( options )
  .then( () => { return this; } )
  .catch( error => {
    console.error( error );
    return null;
  } )
  .finally( () => {
    if( this.debug ) console.log( this );
  } );
}

/**
 * The default label for compute engines.
 * @type {string}
 */
static get label() { return 'compute'; }

/**
 * The instance's label.
 * @type {string}
 */
_label = null;

/**
 * Return the instance's label.
 * @type {string}  - The label.
 */
```

```
get label() { return ( this._label ) ? this._label : DGWebGPUComputeEngine.label; }

/**
 * Set the instance's label.
 * @param {string} label - The label.
 */
set label( label ) { this._label = label; }

/**
 * Generate a label for the instance's subcomponent. The three inputs are joined thus: [
   label, key, type ] => 'label-key-type'
 * @param {string} label - The base label.
 * @param {string} key - The key sub-label.
 * @param {string} type - The type sub-label.
 */
_makeLabel( label, key, type ) {
  const array = [];

  if( label && typeof label == 'string' && label  != '' ) array.push( label );
  if( key   && typeof key == 'string' && key    != '' ) array.push( key );
  if( type  && typeof type  == 'string' && type   != '' ) array.push( type );

  return array.join( '-' );
}

/**
 * An array of three integers which indicates the x, y and z values of a compute workgroup.
 * @type {Array<Int,3>}
 * @todo We need to interrogate the adapter to determine what the max workgroup sizes are.
 */
_workgroupSize = [ 64, 1, 1 ];

/**
 * Get the compute engine's workgroup size
 * @type {Array<Int,3>}  - An array of three integers which indicates the x, y and z values
   of a compute workgroup.
 */
get workgroupSize() { return this._workgroupSize; }
```

```
/**
 * Sets the compute engine's workgroup size
 * @param {Array<Int,3>} wgs - An array of three integers which indicates the x, y, and z
   values of a compute workgroup.
 * @todo We need to check against the valid ranges.
 */
set workgroupSize( wgs = [ 64, 1, 1 ] ) {
  if( !( wgs instanceof Array ) || wgs.size != 3 )
    throw new TypeError( "Workgroup size must be an array of three integer values: [ x, y, z
   ]!" );

  this._workgroupSize = wgs;
}

/**
 * A map of bind group layout descriptors used to build the bind groups.
 * @type {Map<string,Array<GPUBindGroupLayoutDescriptor>>}
 */
_bindGroupLayoutEntries = new Map();

/**
 * Get an array of bind group layouts for the specified key.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @returns {Array<GPUBindGroupLayoutDescriptor>}
 */
bindGroupLayoutEntriesForKey( bindGroupKey ) {
  // Check inputs
  if( !bindGroupKey || typeof bindGroupKey !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof bindGroupKey}!` );

  // If we don't yet have an array of bind group layouts for this key, add one.
  if( !this._bindGroupLayoutEntries.has( bindGroupKey ) ) this._bindGroupLayoutEntries.set(
   bindGroupKey, [] );

  // Return the array of bind group layouts entries.
  return this._bindGroupLayoutEntries.get( bindGroupKey );
}

/**
```

```
 * Get a bind group layout descriptor for a particular bind group.
 * @param {GPUBindGroupLayoutDesc} bindGroupKey - A string identifying the bind group.
 */
bindGroupLayoutDescriptorForKey( bindGroupKey ) {
  // get the entries
  const entries = this.bindGroupLayoutEntriesForKey( bindGroupKey );

  // if we don't have any, we can't create a bind group!
  if( entries.length == 0 )
    throw new RangeError( `No bind group layout entries for key '${bindGroupKey}'!` );

  // index the entries
  entries.forEach( ( e, i ) => e.binding = i );

  return {
    label : bindGroupKey,
    entries : entries,
  };
}

/**
 * Add a bind group layout entry.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @param {GPUBindGroupLayoutEntry} entry - A layout to add to the bind group.
 */
addBindGroupLayoutEntryForKey( bindGroupKey, entry ) {
  if( entry === undefined || entry === null )
    throw new TypeError( 'Attempt to add null layout to bind group!' );

  // Get the layout array
  let entries = this.bindGroupLayoutEntriesForKey( bindGroupKey );

  // Add the layout entry
  entries.push( entry );

  // Support method chaining
  return this;
}
```

```
/**
 * Add a GPUBufferBindingLayout to the bind group layouts for a key.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @param {string} label - A string containing the buffer label we want to attach to the
   layout.
 * @param {GPUShaderStageFlags} visibility - A bitset of the members of GPUShaderStage.
 * @param {GPUBufferBindingType} type - Indicates the type required for buffers bound to this
   bindings. Valid values: "uniform", "storage", "read-only-storage"
 * @param {boolean} hasDynamicOffset - Indicates whether this binding requires a dynamic
   offset.
 * @param {GPUSize64} minBindingSize - Indicates the minimum buffer binding size.
 * @see https://gpuweb.github.io/gpuweb/#dictdef-gpubufferbindinglayout
 * @todo Make sure we're supporting both the adapter and device supported limits.
 * @see https://gpuweb.github.io/gpuweb/#supported-limits
 */
addBufferBindingLayoutEntryForKey( bindGroupKey, label ="", visibility =
   GPUShaderStage.COMPUTE, type = 'uniform', hasDynamicOffset = false, minBindingSize = 0 ) {
  const valid_types = [ "uniform", "storage", "read-only-storage" ];
  if( !valid_types.includes( type ) )
    throw RangeError( `Invalid GPUBufferBindingType: '${type}'! Valid types =
   [${valid_types.join(", ")}]` );

  // Create a layout
  const layout = {
    visibility  : visibility,
    buffer    : {
      label       : label,
      type        : type,
      hasDynamicOffset  : hasDynamicOffset,
      minBindingSize    : minBindingSize,
    },
  };

  // Add the binding layout to the array
  this.addBindGroupLayoutEntryForKey( bindGroupKey, layout );

  // Support method chaining
  return this;
}
```

```
/**
 * Build and return a binding buffer layout for the specified key.
 * @param {string} bindGroupKey - A string instance identifying the bind group.
 * @param {string} bufferKey - A string instance identifying the buffer.
 */
bufferEntryForBindGroupWithKey( bindGroupKey, bufferKey ) {
  const buffer  = this.bufferForKey( bufferKey );
  const index   = this.bufferBindingIndexForKey( bindGroupKey, bufferKey );
  const entry   = { binding : index, resource : { buffer : buffer } };

  return entry;
}

/**
 * Get the buffer binding index for this buffer.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @param {string} bufferLabel - A string identifying the buffer we're trying to interrogate.
 */
bufferBindingIndexForKey( bindGroupKey, bufferLabel ) {
  // get the layout descriptor
  const layout = this.bindGroupLayoutDescriptorForKey( bindGroupKey );
  const entries = layout.entries;

  const index = entries
    .findIndex( ( entry ) => {
    return entry
      .hasOwnProperty( 'buffer' )
      && entry.buffer.hasOwnProperty( 'label' )
      && entry.buffer.label == bufferLabel;
  });

  if( index < 0 )
    throw new RangeError( `No buffer binding for buffer labeled '${bufferLabel}' in layout
   '${bindGroupKey}' exists!` );

  return index;
}
```

```
/**
 * Add a GPUSamplerBindingLayout to the bind group layouts for a key.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @param {string} label - A string containing the label we want to attach to the layout.
 * @param {GPUShaderStageFlags} visibility - A bitset of the members of GPUShaderStage.
 * @param {GPUSamplerBindingType} type - Indicates the required type of a sampler bound to
   this bindings. Valid values: "filtering", "non-filtering", "comparison"
 * @todo Move to DGWebGPURenderEngine
 */
addSamplerBindingLayoutEntryForKey( bindGroupKey, label = "", visibility =
   GPUShaderStage.COMPUTE, type = "filtering" ) {
  const valid_types = [ "filtering", "non-filtering", "comparison" ];
  if( !valid_types.includes( type ) )
    throw RangeError( `Invalid GPUSamplerBindingType: ${type}` );

  // Create a layout
  const layout = {
    label   : label,
    visibility  : visibility,
    sampler   : {
      type  : type,
    },
  };

  // Add the binding layout to the array
  this.addBindGroupLayoutEntryForKey( bindGroupKey, layout );

  // Support method chaining
  return this;
}

/**
 * Add a GPUTextureBindingLayout to the bind group layouts for a key.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @param {string} label - A string containing the label we want to attach to the layout.
 * @param {GPUShaderStageFlags} visibility - A bitset of the members of GPUShaderStage.
 * @param {GPUTextureSampleType} sampleType - Indicates the type required for texture views
   bound to this binding.
 * @param {GPUTextureViewDimension} viewDimension - Indicates the required dimension for
```

```
   texture views bound to this binding.
 * @param {boolean} multisampled - Indicates whether or not texture views bound to this
   binding must be multisampled.
 */
addTextureBindingLayoutEntryForKey( bindGroupKey, label = "", visibility =
   GPUShaderStage.COMPUTE, sampleType = "float", viewDimension = "2d", multisampled = false )
   {
  const valid_types = [ "float", "unfilterable-float", "depth", "sint", "uint" ];
  if( !valid_types.includes( sampleType ) )
    throw RangeError( `Invalid GPUTextureSampleType: ${sampleType}` );

  const valid_dims = [ "1d", "2d", "2d-array", "cube", "cube-array", "3d" ];
  if( !valid_dims.includes( viewDimension ) )
    throw RangeError( `Invalid GPUTextureViewDimension: ${viewDimension}` );

  // Create a layout
  const layout = {
    label   : label,
    visibility  : visibility,
    sampler   : {
      sampleType    : sampleType,
      viewDimension : viewDimension,
      multisampled  : multisampled,
    },
  };

  // Add the binding layout to the array
  this.addBindGroupLayoutEntryForKey( bindGroupKey, layout );

  // Support method chaining
  return this;
}

/**
 * Add a GPUStorageTextrueBindingLayout entry to the bind group layout entries for a key.
 * @param {string} bindGroupKey - A string uniquely describing the bind group layout.
 * @param {string} label - A string containing the label we want to attach to the layout.
 * @param {GPUShaderStageFlags} visibility - A bitset of the members of GPUShaderStage.
 * @param {GPUStorageTextureAccess} access - Indicates whether texture views bound to this
```

```
    binding will be bound for read-only or write-only access.
   * @param {GPUTextureFormat} format - The required format of texture views bound to this
    binding.
   * @param {GPUTextureViewDimension} viewDimension - Indicates the required dimension for
    texture views bound to this binding.
   */
  addStorageTextureBindingLayoutForKey( bindGroupKey, label = "", visibility =
    GPUShaderStage.COMPUTE, access = "write-only", format, viewDimension ="2d" ) {
    const valid_access_types = [ "access" ];
    if( !valid_access_types.includes( access ) )
      throw RangeError( `Invalid GPUStorageTextureAccess: ${access}` );

    const valid_dims = [ "1d", "2d", "2d-array", "cube", "cube-array", "3d" ];
    if( !valid_dims.includes( viewDimension ) )
      throw RangeError( `Invalid GPUTextureViewDimension: ${viewDimension}` );

    // Don't check valid texture formats, let's see what happens when we send invalid ones in.
     I think we can remove all of this checking code and leave it to the gpu.

    // Create a layout
    const layout = {
      label   : label,
      visibility  : visibility,
      sampler   : {
        access      : access,
        format      : format,
        viewDimension : viewDimension,
      },
    };

    // Add the binding layout to the array
    this.addBindGroupLayoutEntryForKey( bindGroupKey, layout );

    // Support method chaining
    return this;
  }

  /**
   * A cached map containing the bind group layouts for this object.
```

```
 * @type {Map<String,GPUBindGroupLayout>}
 */
_bindGroupLayouts = new Map();

/**
 * Create and return a bind group layout based on the bind group layout descriptors for this
   key.
 * @param {string} bindGroupKey - A key uniquely describing a bind group layout.
 * @returns {GPUBindGroupLayout}  - The bind group layout
 * @throws TypeError
 */
bindGroupLayoutForKey( bindGroupKey ) {
  if( !bindGroupKey || typeof bindGroupKey !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  // convenience
  let layouts = this._bindGroupLayouts;

  // Lazily create the bind group layout if it doesn't yet exist.
  if( !layouts.has( bindGroupKey ) ) {
    // Get the layout descriptor
    const desc = this.bindGroupLayoutDescriptorForKey( bindGroupKey );

    // Create the layout
    const layout = this.device.createBindGroupLayout( desc );

    // Set the key
    layouts.set( bindGroupKey, layout );
  }

  // Return the bind group layout
  return layouts.get( bindGroupKey );
}

/**
 * A map of bindgroups for a specified key.
 * @type {Map<string,GPUBindGroup>}
 */
_bindGroups = new Map();
```

```
/**
 * Retrieve the bind group for a specified key
 * @param {string} key - A string which uniquely identifies a bind group.
 * @todo Remember to complete me before we get to this point.
 * @todo We need to be able to get indices for these! How do we do that?
 */
bindGroupForKey( bindGroupKey ) {
  if( !bindGroupKey || typeof bindGroupKey !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof bindGroupKey}!` );

  // If the array containing buffers for the key doesn't exist, add one.
  if( !this._bindGroups.has( bindGroupKey ) )
    throw new RangeError( `No bind group for key '${bindGroupKey}' exists!` );

  // Need to fill this out
  return this._bindGroups.get( bindGroupKey );
}

/**
 * Add a bind group.
 * @param {string} bindGroupKey - A string which uniquely identifies a bind group.
 * @param {GPUBindGroupDescriptor} desc - The bind group descriptor object.
 */
_addBindGroup( bindGroupKey, desc ) {
  const bindGroup = this.device.createBindGroup( desc );

  // Should we use a map instead of an object?
  this._bindGroups.set( bindGroupKey, bindGroup );
}

/**
 * Add a bind group with the specified layout and entries.
 * @param {string} bindGroupKey - A string which uniquely identifies a bind group.
 * @param {GPUBindGroupLayoutDescriptor} layout - The bind group layout descriptor.
 * @param {Array<GPUBindGroupLayoutEntryDesc>} entries - An array of bind group layout
   entries.
 */
setBindGroupForKey( bindGroupKey, layout, entries ) {
```

```
  if( !bindGroupKey || typeof bindGroupKey !== 'string' )
  throw new TypeError( `Invalid key type: ${typeof bindGroupKey}!` );

  const desc = {
    layout  : layout,
    entries : entries,
  };

  this._addBindGroup( bindGroupKey, desc );
}

deleteBindGroup( key ) { (key);
  // @todo fill me in if needed
}

/**
 * A map of buffers to be used by the compute engine.
 * @type {Map<string,GPUBuffer>}
 */
_buffers = new Map();

/**
 * Return a GPUBuffer instance with the specified key.
 * @param {string} key - A string uniquely describing the array of buffers.
 */
bufferForKey( key ) {
  if( !key || typeof key !== 'string' )
  throw new TypeError( `Invalid key type: ${typeof key}!` );

  // If the array containing buffers for the key doesn't exist, add one.
  if( !this._buffers.has( key ) )
    throw new RangeError( `No buffer for key '${key}' exists!` );

  // Need to fill this out
  return this._buffers.get( key );
}

/**
 * Add a buffer to the compute engine.
```

```
 * @param {string} key - A key for referring to the buffer later.
 * @param {GPUBufferDescriptor|GPUBuffer} desc - A GPUBufferDescriptor object, or an already
   created gpu buffer.
 * @todo Add support for event handling.
 */
_addBuffer( key, desc ) {
  const buffer = ( desc instanceof GPUBuffer ) ? desc : this.device.createBuffer( desc );

  // Should we use a map instead of an object?
  this._buffers.set( key, buffer );
}

/**
 * Add a buffer to the compute engine.
 * @param {string} key - A unique string for identifying the buffer.
 * @param {string} label - A string labelling the buffer.
 * @param {GPUSize64} size - The size of the buffer in bytes.
 * @param {GPUBufferUsageFlags} usage - The allowed usages for the buffer.
 * @param {boolean} mappedAtCreation - If true creates the buffer in an already mapped state.
 * @see https://gpuweb.github.io/gpuweb/#gpubuffer
 */
addBufferForKey( key,  label = this.label, size = 0, usage = 0, mappedAtCreation = false ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key: ${typeof key}!` );

  // If the array containing buffers for the key doesn't exist, add one.
  if( !this._buffers.has( key ) )
    throw new RangeError( `No buffer for key '${key}' exists!` );

  // Check size
  if( !size || size < 1 )
    throw new RangeError( `GPUBuffer must have size > 0: ${size}!` );

  // Generate a label
  label = this._makeLabel( label, key, 'buffer' );

  // Create the buffer descriptor.
  const desc = { label : label, size : size, usage : usage, mappedAtCreation:
   mappedAtCreation };
```

```
  // Add the buffer.
  this._addBuffer( key, desc );

  // Allow method chaining.
  return this;
}

deleteBufferForKey( key ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key: ${typeof key}!` );

  // If there is a buffer for this key, remove it and delete it.
  if( this._buffers.has( key ) ) {
    let buffer = this._buffers.get( key );
    this._buffers.delete( key );
    buffer.destroy()
  }
}

resizeBufferForKey( key, size ) {
  // Get the old buffer
  const oldBuffer = this.bufferForKey( key );

  // if the old buffer size is the same as the new size, do nothing.
  if( oldBuffer.size == size ) return oldBuffer;

  // create a new buffer
  const newBuffer = this._device.createBuffer( { label : oldBuffer.label, size : size, usage
   : oldBuffer.usage, mappedAtCreation: false } );

  // set the new buffer
  this._buffers.set( key, newBuffer );

  // destroy the old buffer
  oldBuffer.destroy();

  // return the new buffer
  return newBuffer;
```

```
}

/**
 * Add a buffer to the compute engine
 * @param {string} key - A key for referring to the buffer later.
 * @param {GPUBufferDescriptor} options - A GPUBufferDescriptor object
 *
 * @see https://webgpu.rocks/reference/dictionary/gpubufferdescriptor/#idl-gpubufferdescriptor
 */
addBuffer(
  key,
  label = this.label,
  size = 0,
  usage = 0,
  mappedAtCreation = true
) {
  // Make sure key is a string, and is not empty.
  if( key === undefined || typeof key !== 'string' || key === '' )
    throw new RangeError( `GPUBuffer key invalid '${key}'!` );

  // Validate the values
  if( this._buffers.has( key ) )
    throw new RangeError( `GPUBuffer with key '${key}' exists!` );

  // Generate a label
  label = this._makeLabel( label, key, 'buffer' );

  // Check size
  if( !size || size < 1 )
    throw new RangeError( `GPUBuffer must have size > 0: ${size}!` );

  // Add the buffer
  this._addBuffer( key, { label: label, size: size, usage: usage, mappedAtCreation:
   mappedAtCreation } );

  // Allow for method chaining
  return this;
}
```

```
/**
 * A map containing the shaders required by the compute engine.
 * @type {Map<String,DGWebGPUShaderBase>}
 */
_shaders = new Map();

/**
 * Return an array of shaders for use by the compute engine.
 * @param {string} key - A key for referring to the buffer later.
 */
shaderForKey( key ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  // If the array containing buffers for the key doesn't exist, add one.
  if( !this._shaders.has( key ) )
    throw new RangeError( `No shader for key '${key}' exists!` );

  return this._shaders.get( key );
}

setShaderForKey( key, shader, override = false ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  if( this._shaders.has( key ) && override )
    throw new RangeError( `Shader for key '${key}' exists!` );

  this._shaders.setOrDeleteIfNull( key, shader );

  return this;
}

deleteShaderForKey( key ) { return this.setShaderForKey( key, null, true ); }

/**
 * Build a DGWebGPUShader from source.
 * @param {string} type - The shader type: 'compute', 'fragment', 'vertex'
 * @param {GPUShaderOptionsDescriptor} shaderOptions - A dictionary containing shader options.
```

```
 * @returns {Promise<DGWebGPUShaderBase>}  - Returns a promise of a DGWebGPUShader subclass.
 */
async buildShaderFromSource( type = DGWebGPUComputeShader.type, shaderOptions ) {
  const options = {
    inherit     : this,
    shaderOptions : shaderOptions,
  };

  return DGWebGPUShaderBase.build( type, options, this.debug )
}

/**
 * Load WGSL shader code from a url.
 * @param {string} url - The url where the shader source can be found.
 * @param {string} type - The type of the shader we want to create.
 * @param {object} keys - Keys to be used by the evaluator to spread and replace in the
   source.
 * @param {CallableFunction} evaluator - An evaluator function to use to use to replace
   variables in the source.
 * @returns {Promise<DGWebGPUComputeEngine>}  - Returns a promise which allows method
   chaining.
 */
async loadShaderFromURL( url = null, type = DGWebGPUComputeShader.type, keys = {}, evaluator
   = null ) {
  if( !url || typeof url !== 'string' )
    throw new TypeError( `URL of source must be a string!` )

  return fetch( url )
  .then( ( response ) => response.text() )
  .then( ( src ) => {
    const options = {
      label: this._makeLabel( this.label, type, 'shader' ),
      code: ( evaluator ) ? evaluator( keys, src ) : src,
    };

    return this.buildShaderFromSource( type, options )
  })
  .catch( ( e ) => console.error( e ) )
}
```

```
/**
 * The pipeline layout descriptors.
 * @type {Map<string,Array<GPUBindGroupLayoutDescriptor>>}
 */
_pipelineLayoutDescriptors = new Map();

/**
 * Add a pipleine layout descriptor.
 * @param {string} key - A string indicating the pipeline layout we want.
 * @param {Array<string>} bindGroupKeys - An array of bindGroupKeys.
 */
addPipelineLayoutDescriptorForKey( key, bindGroupKeys = [] ) {
  if( !bindGroupKeys || !( bindGroupKeys instanceof Array ) )
    throw new TypeError( `Invalid bind group keys type: ${typeof bindGroupKeys}!` );

  if( bindGroupKeys.length < 1 )
    throw new RangeError( `No keys submitted!` );

  const entries = bindGroupKeys.map( key => this.bindGroupLayoutForKey( key ) );
  const desc = {
    label       : key,
    bindGroupLayouts  : entries,
  };

  return this.setPipelineLayoutDescriptorForKey( key, desc );
}

/**
 * Return a pipeline layout descriptor identified by key.
 * @param {string} key - The pipeline layout descriptor.
 */
pipelineLayoutDescriptorForKey( key ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  // Do we have one?
  if( !this._pipelineLayoutDescriptors.has( key ) )
    throw new RangeError( `No pipeline layout descriptor for key '${key}' exists!` );
```

```
  return this._pipelineLayoutDescriptors.get( key );
}

/**
 * Set a pipeline layout descriptor.
 * @param {key} key - A string instance identifying a pipeline layout descriptor.
 * @param {GPUPipelineDescriptor|null} desc - The pipeline descriptor to set, or null, if we
   wish to delete.
 * @param {boolean} override - true if we wish to overwrite any existing values, false
   otherwise.
 */
setPipelineLayoutDescriptorForKey( key, desc, override = false ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  let descriptors = this._pipelineLayoutDescriptors;

  if( descriptors.has( key ) && override == false )
    throw new RangeError( `Pipeline layout descriptor for key '${key}' exists!` );

  descriptors.setOrDeleteIfNull( key, desc );

  return this;
}

/**
 * Delete a pipeline layout descriptor.
 * @param {string} key - A string instance identifying a pipeline layout descriptor.
 * @calls setPipelineLayoutDescriptorForKey
 */
deletePipelineDescriptorForKey( key ) { return this.setPipelineDescriptorForKey( key, null,
   true ); }

/**
 * A map of pipeline layouts.
 * @type {Map<string,Array<GPUPipelineLayout>>}
 */
_pipelineLayouts = new Map();
```

```
/**
 * Returns a default GPUPipelineLayout
 * @type {GPUPipelineLayout}
 */
static get DEFAULT_PIPELINE_LAYOUT() { return "auto"; }

/**
 * Retrieves a pipeline layout for a given key.
 * @param {string} key - A string instance identifying the pipeline layout.
 */
pipelineLayoutForKey( key ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  let layouts = this._pipelineLayouts;

  // If we don't already have these layouts, then try to build one.
  if( !layouts.has( key ) ) {
    const desc    = this.pipelineLayoutDescriptorForKey( key );
    const layout  = this.device.createPipelineLayout( desc );

    layouts.set( key, layout );
  }

  // return the output of the pipeline layouts
  return layouts.get( key );
}

/**
 * Sets the default pipeline layout for the specified key.
 * @param {string} key - A string instance identifying the pipeline layout.
 */
setAutoPipelineLayoutForKey( key ) { return this.addPipelineLayoutForKey( key,
   DGWebGPUComputeEngine.DEFAULT_PIPELINE_LAYOUT ); }

setPipelineLayoutForKey( key, layout, override = false ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );
```

```
  // If we already have a layout for this key, then add a new one
  if( this._pipelineLayouts.has( key ) && override == false )
    throw new RangeError( `Pipeline layout for key '${key}' already exists!` );

  this._pipelineLayouts.setOrDeleteIfNull( key, layout );

  return this;
}

/**
 * Deletes a pipeline layout identified by key.
 * @param {string} key - A string instance identifying the pipeline layout.
 */
deletePipelineLayoutForKey( key ) { return this.setPipelineLayoutForKey( key, null, true ); }

/**
 * Build a compute pipeline.
 * @param {DGWebGPUComputeShader} shader - The shader to use in the pipeline. Defaults to the
   shader already installed.
 * @param {GPUPipelineLayoutDescriptor|string="auto"} layout - The layout to use when
   building the compute pipeline. Defaults to "auto"
 * @param {string} entryPoint - The entry point in the shader
 */
buildComputePipeline( shader = this.computeShader, layout =
   DGWebGPUComputeEngine.DEFAULT_PIPELINE_LAYOUT, entryPoint = 'main' ) {
  const desc = {
    layout  : layout,
    compute : {
      module    : shader.module,
      entryPoint  : entryPoint,
    },
  };

  return  this.device.createComputePipeline( desc );
}

/**
 * A map containing pipelines identified by key.
```

```
 * @type {Map<string,GPUComputePipeline|GPURenderPipeline>}
 */
_pipelines = new Map();

/**
 * Retrieves a pipeline for a specified key.
 * @param {string} key - A string instance identifying the pipeline.
 * @throws {TypeError|RangeError}
 */
pipelineForKey( key ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  if( !this._pipelines.has( key ) )
    throw new RangeError( `No pipeline for key '${key}' exists!` );

  return this._pipelines.get( key );
}

/**
 * Add pipeline identified by key to the map.
 * @param {string} key - A string instance identifying the pipeline.
 * @param {GPUComputePipeline|GPURenderPipeline} pipeline - The pipeline to add.
 * @param {boolean} override - true if we wish to overwrite or delete an existing pipeline,
   false otherwise.
 * @throws{TypeError|RangeError}
 */
setPipelineForKey( key, pipeline, override = false ) {
  if( !key || typeof key !== 'string' )
    throw new TypeError( `Invalid key type: ${typeof key}!` );

  if( this._pipelines.has( key ) && override == false )
    throw new RangeError( `Pipeline for key '${key}' already exists!` );

  this._pipelines.setOrDeleteIfNull( key, pipeline );

  return this;
}
```

```
/**
 * Delete a pipeline identified by key.
 * @param {key} key - A string instance identifying the pipeline.
 */
deletePipelineForKey( key ) {
  return this.setPipelineForKey( key, null, true );
}

/**
 * A convenience method for retrieving a compute pipeline identified by the key 'compute'
 * @type {GPUComputePipeline}  - The pipeline
 */
get computePipeline() { return this.pipelineForKey( 'compute' ); }

/**
 * A convenience method for setting the default compute pipeline,
 * @param {GPUComputePipeline} pipeline - The compute pipeline.
 */
set computePipeline( pipeline ) { this.setPipelineForKey( 'compute', pipeline ); }

/**
 * Return a compute pipeline that does nothing.
 */
noOpComputePipeline() {
  const shader = { module : this.device.createShaderModule( { code:
   DGWebGPUComputeShader.DEFAULT_SHADER_SOURCE, } ) };
  return this.buildComputePipeline( shader );
}

/**
 * The currently active command encoder.
 * @type {GPUCommandEncoder}
 */
_commandEncoder = null;

/**
 * Retrieve the current command encoder, or lazily create one.
 * @type {GPUCommandEncoder}  - The currently active command encoder.
 */
```

```
get commandEncoder() {
  // Lazily create the command encoder if it doesn't exist.
  if( this._commandEncoder === null )
    this._commandEncoder = this.device.createCommandEncoder();

  return this._commandEncoder;
}

/**
 * The currently active pass encoder.
 * @type {GPUComputePassEncoder|GPURenderPassEncoder}
 */
_pass = null;

/**
 * Retrieve the current pass encoder.
 * @type {GPUComputePassEncoder|GPURenderPassEncoder|null}  - The pass encoder, or null if no
   pass is active.
 */
get pass() { return this._pass; }

/**
 * Sets the current pass encoder.
 * @param {GPUComputePassEncoder|GPURenderPassEncoder|null} encoder - The pass encoder to set.
 */
set pass( encoder ) { this._pass = encoder; }

/**
 * Submit work to the device queue.
 */
submitQueue() {
  if( this.debug ) console.log( 'submit queue' );
  this.device.queue.submit( [ this.commandEncoder.finish() ] );
  // at this point, the command encoder is dead, so zero it out so we can create a new one.
  this._pass = this._commandEncoder = null;

  // In case we want notification when this has completed.
  return this.device.queue.onSubmittedWorkDone();
}
```

```
/**
 * Perform work required by the compute engine.
 * @discussion Subclasses should fill this in with appropriate functionality.
 * @param {GPUComputePassEncoder|GPURenderPassEncoder} pass - The pass encoder upon which to
   issue commands.
 */
doPass( pass = this.pass ) { (pass); }

/**
 * Begin the render/compute pass.
 */
passBegin() {
  if( this.debug ) console.log( 'pass begin' );
  this.pass = this.commandEncoder.beginComputePass();

  return this;
}

/**
 * End the render/compute pass.
 */
passEnd() {
  if( this.debug ) console.log( 'pass end' );
  this.pass.end();

  return this;
}

/**
 * Read from any buffers written to during the pass.
 * @discussion Subclasses should override this method with appropriate commands.
 */
passReadBuffers() {}

/**
 * Write to any buffers to be read during the pass.
 * @discussion Subclasses should override this method with appropriate commands.
 */
```

```
  passWriteBuffers() {}

  /**
   * Execute a compute pass.
   * @todo should this be async? Reading the buffers may be required, and we need to prevent
     locking up the buffers.
   */
  async execute() {
    // map the write buffers
    return this.mapWriteBuffers()
      .then( () => { return this.passWriteBuffers(); } )
      .then( () => { return this.passBegin(); } )
      .then( () => { return this.doPass(); } )
      .then( () => { return this.passEnd(); } )
      .then( () => { return this.passReadBuffers(); } )
      .then( () => { return this.submitQueue(); } )
      .catch( ( error ) => { console.error( error ); } )
      .finally( () => {
        if( this.debug ) console.log( this );

        return this;
      } )
  }
}
```

C.3.1 XOR128

Listing C.12. WGSL code for the `XOR128` pseudo-random number generator.

```
/**
 * @file /src/common/gpu/shader/wgsl/xor128.wgsl
 * @author Rob Dotson
 * @copyright 2022 Rob Dotson. All Rights Reserved.
 *
 * @project dynamical.js
 * @description This file contains the WGSL Implementation of the XOR128 algorithm.
 *
 * @created 12 Aug 2021
 * @cite https://www.jstatsoft.org/article/view/v008i14
 * @cite https://github.com/vadimdi/QCDGPU/blob/83eb474a221563bce50682b5c5de8ad7125c8682/Docs/
    hpc-ua-2013_paper.pdf
 *
 * @todo We need to store local state here, it's pretty important not to accidentally pollute
    the random number state.
 * @todo We need to figure out why we can't define any compile-time constants. I understand
    constexpr() are not supported,
 * but it doesn't make sense that we can't even create a constant that is fixed.
 */

@group(0) @binding(0) var<storage,read_write>   state   : array<vec4<u32>>;
@group(0) @binding(1) var<storage,read_write>   output  : array<u32>;
@group(0) @binding(2) var<storage,read_write>   outputf : array<f32>; // hrm, this seems like an
     error

fn xor128_step( id: u32 ) -> u32 {
        let t : u32     = state[id].x ^ ( ( state[id].x ) << 11u );

        state[id].x             = state[id].y;
        state[id].y             = state[id].z;
        state[id].z             = state[id].w;
        state[id].w             = ( state[id].w ^ ( state[id].w >> 19u ) ) ^ ( t ^ ( t >> 8u ) )
```

```
    ;

        return state[id].w;
}

fn xor128_step_float( id: u32 ) -> f32 {
        let XOR128_m_FP         = 4294967296.0f;
        let XOR128_K            = 2.3283064365386962890625E-10f; // 1/2^32
        let XOR128_MIN_FLT      = 1.0f / 4294967296.0f;
        let XOR128_MAX_FLT      = 4294967295.0f / 4294967296.0f;

        var x = xor128_step( id ); // this is error prone, we need to use the function below,
    once we figure out how best to do this.
        while( x < 1 || x > 4294967295 ) {
                x = xor128_step( id );
        }

        return f32( x ) / XOR128_m_FP;
}

fn u32_to_f32( a : u32, b : u32, min : f32, max : f32, k : f32 ) -> f32 {
        return ( f32( a ) + k * f32( b ) - ( min + k * max ) ) / ( ( max - min ) * ( 1.0 - k ) )
    ;
}

@compute
@workgroup_size(${workgroupSize[0]}u, ${workgroupSize[1]}u, ${workgroupSize[1]}u)
fn main( @builtin( global_invocation_id ) global_invocation_id : vec3<u32> ) {
        let GID_SIZE    = ( ${workgroupSize[0]}u * ${workgroupSize[1]}u * ${workgroupSize[2]}u )
    ;
        let GID                 = global_invocation_id.x + global_invocation_id.y * ${
    workgroupSize[0]}u + global_invocation_id.z * ${workgroupSize[0]}u * ${workgroupSize[1]}u;

        let total = arrayLength( &output );             // get the length of the data we need to
     collect
        let index = global_invocation_id.x;             // get the current invocation id
        if( index >= total ) { return; }                // don't overwrite the last few values.

        output[ index ] = xor128_step( GID );           // write the output
```

```
}

@compute
@workgroup_size(${workgroupSize[0]}u, ${workgroupSize[1]}u, ${workgroupSize[1]}u)
fn main_float( @builtin( global_invocation_id ) global_invocation_id : vec3<u32> ) {
        let GID_SIZE    = ( ${workgroupSize[0]}u * ${workgroupSize[1]}u * ${workgroupSize[2]}u )
    ;
        let GID                 = global_invocation_id.x + global_invocation_id.y * ${
    workgroupSize[0]}u + global_invocation_id.z * ${workgroupSize[0]}u * ${workgroupSize[1]}u;

        let total = arrayLength( &output );             // get the length of the data we need to
     collect
        let index = global_invocation_id.x;             // get the current invocation id
        if( index >= total ) { return; }                // don't overwrite the last few values.

        outputf[ index ] = xor128_step_float( GID );    // write the output
}
```

Ahn, J., Plaisant, C., & Shneiderman, B. (2014). A task taxonomy for network evolution analysis. *IEEE transactions on visualization and computer graphics*, *20*(3), 365–376. https://doi.org/10.1109/TVCG.2013.238

Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). *Real-Time Rendering* (3rd ed.). A K Peters, Ltd.

Apple Computer, Inc. (1984). *Inside Macintosh* (Vol. I). Addison-Wesley.

Apple Computer, Inc. (1985). *Inside Macintosh* (Vol. II). Addison-Wesley.

Apple Inc. (2021). *WebKit* [Framework]. https://webkit.org

Apple Inc. (2022a). *Metal* [Framework]. https://developer.apple.com/metal/

Apple Inc. (2022b). *Safari* (Version 16.0)[Computer Software]. https://www.apple.com/safari/

Appleby, A. (2016). *MurmurHash* [Algorithm]. https://github.com/aappleby/smhasher

Archambault, D., Purchase, H., & Pinaud, B. (2011). Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *IEEE Transactions on Visualization and Computer Graphics*, *17*(4), 539–552. https://doi.org/10.1109/TVCG.2010.78

Archambault, D. & Purchase, H. C. (2013). Mental map preservation helps user orientation in dynamic graphs. In W. Didimo & M. Patrignani (Eds.), *Graph Drawing* (pp. 475–486). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-36763-2_42

Argyriou, E., Bekos, M., & Symvonis, A. (2012). Maximizing the Total Resolution of Graphs. *The Computer Journal*, *56*(7), 887–900. https://doi.org/10.1093/comjnl/bxs088

Bach, B., Dragicevic, P., Archambault, D., Hurter, C., & Carpendale, S. (2017). A descriptive framework for temporal data visualizations based on generalized space-time cubes. *Computer Graphics Forum*, *36*(6), 36–61. https://doi.org/10.1111/cgf.12804

Bailey, M. & Cunningham, S. (2009). *Graphics Shaders: theory and practice*. A K Peters, Ltd.

Baldwin, D. & Rost, R. (2019a). *The OpenGL ES® Shading Language* (Version 3.20.6)[Standard]. https://registry.khronos.org/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.html. Accessed: 2021-04-11

Baldwin, D. & Rost, R. (2019b). *The OpenGL ES® Shading Language* (Version 3.20.6)[Standard]. https://registry.khronos.org/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.html

Beck, F., Burch, M., & Diehl, S. (2009). Towards an aesthetic dimensions framework for dynamic graph visualisations. In *2009 13th International Conference Information Visualisation* (pp. 592–597).: IEEE. https://doi.org/10.1109/IV.2009.42

Beck, F., Burch, M., & Diehl, S. (2013). Matching application requirements with dynamic graph visualization profiles. In *2013 17th International Conference on Information Visualisation* (pp. 11–18). https://doi.org/10.1109/IV.2013.2

Beck, F., Burch, M., Vehlow, C., Diehl, S., & Weiskopf, D. (2012). Rapid serial visual presentation in dynamic graph visualization. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 185–192).: IEEE. https://doi.org/10.1109/VLHCC.2012.6344514

Bertin, J. (2011). *Semiology of Graphics: Diagrams Networks Maps*. Esri Press.

Bleiweiss, A. (2008). Gpu accelerated pathfinding. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08 (pp. 65–74). Goslar, DEU: Eurographics Association.

Bostok, M. (2021). *D3* (Version 7)[Computer Software]. https://d3js.org

Buck, I. & Purcell, T. (2004). A toolkit for computation on gpus. In R. Fernando (Ed.), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (pp. 621–636). Addison-Wesley.

Cebenoyan, C. (2004). Graphics pipeline performance. In R. Fernando (Ed.), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (pp. 473–486). Addison-Wesley.

Chuang, J. & Ahuja, N. (1998). An analytically tractable potential field model of free space and its application in obstacle avoidance. *IEEE transactions on systems, man and cybernetics. Part B, Cybernetics*, *28*(5), 729–736. https://doi.org/10.1109/3477.718522

Cohen, R. F., Di Battista, G., Tamassia, R., Tollis, I. G., & Bertolazzi, P. (1992). A framework for dynamic graph drawing. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, SCG '92 (pp. 261–270). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/142675.142728

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (2nd ed., Vol. 1). MIT Press.

Couturier, R. (2014). Pseudorandom number generator on gpu. In *Designing Scientific Applications on GPUs* (pp. 463–474). Chapman and Hall/CRC. https://doi.org/10.1201/b16051-29

Dakkak, A., Pearson, C., & Hwu, W. (2016). Webgpu: A scalable online development platform for gpu programming courses. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 942–949). https://doi.org/10.1109/IPDPSW.2016.63

Dalton, S., Olson, L., & Bell, N. (2015). Optimizing sparse matrix-matrix multiplication for the gpu. *ACM transactions on mathematical software*, *41*(4), 1–20. https://doi.org/10.1145/2699470

Demchik, V. (2014). Pseudorandom numbers generation for monte carlo simulations on gpus: Opencl approach. In V. Kindratenko (Ed.), *Numerical Computations with GPUs* (pp. 245–271). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-06548-9_12

Di Battista, G., Eades, P., Tamassia, R., & Tollis, I. G. (1994). Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry : Theory and Applications*, *4*(5), 235–282. https://doi.org/10.1016/0925-7721(94)00014-X

Di Battista, G., Eades, P., Tamassia, R., & Tollis, I. G. (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

Du, X., Wei, Y., & Wu, L. (2017). A multi-constraint layout algorithm for dynamic network visualization. In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)* (pp. 832–836). https://doi.org/10.1109/ICBDA.2017.8078754

Duţu, A., Sinclair, M. D., Beckmann, B. M., Wood, D. A., & Chow, M. (2020). Independent forward progress of work-groups. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20 (pp. 1022–1035).: IEEE Press. https://doi.org/10.1109/ISCA45697.2020.00087

Dunne, C. & Shneiderman, B. (2013). Motif simplification: Improving network visualization readability with fan, connector, and clique glyphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/2470654.2466444

Eades, P. (1984). A heuristic for graph drawing. *Congressus Numerantium*, *42*, 149–160.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., & Worley, S. (2003). *Texturing & Modeling: A Procedural Approach* (3rd ed.). Morgan Kaufmann Publishers.

European Association for Standardizing Information and Communication Systems (2015). *ECMAScript 2015* (6th ed.). European Association for Standardizing Information and Communication Systems. https://tc39.es/ecma262/

European Association for Standardizing Information and Communication Systems (2020). *ECMA-262: ECMAScript Language Specification* (11th ed.). European Association for Standardizing Information and Communication Systems. https://tc39.es/ecma262/

European Association for Standardizing Information and Communication Systems (2021). *ECMA-262: ECMAScript Language Specification. Jobs and Host Operations to Enqueue Jobs* (11th ed.). European Association for Standardizing Information and Communication Systems. https://tc39.es/ecma262/#sec-jobs

Firebase (2022). *Superstatic* (Version 8.0.0)[Computer Software]. https://www.npmjs.com/package/superstatic

Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1996). *Computer Graphics: Principles and Practice* (2nd ed.). Addison-Wesley.

Formann, M., Hagerup, T., Haralambides, J., Kaufmann, M., Leighton, F. T., Symvonis, A., Welzl, E., & Woeginger, G. (1993). Drawing graphs in the plane with high resolution. *SIAM Journal on Computing*, *22*(5), 1035–1052. https://doi.org/10.1137/0222063

Fowler, G., Noll, L. C., Cisco Systems, Vo, K.-P., Google Inc., Eastlake, D., Huawei Technologies, Hansen, T., & AT&T Laboratories (2019). *The FNV Non-Cryptographic Hash Algorithm* [Algorithm]. https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17.html

Frishman, Y. & Tal, A. (2007). Multi-level graph layout on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, *13*(6), 1310–1319. https://doi.org/10.1109/TVCG.2007.70580

Frishman, Y. & Tal, A. (2008). Online dynamic graph drawing. *IEEE Transactions on Visualization and Computer Graphics*, *14*(4), 727–740. https://doi.org/10.1109/TVCG.2008.11

Fruchterman, T. M. J. & Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and Experience*, *21*(11), 1129–1164. https://doi.org/10.1002/spe.4380211102

Galán, S. F. & Mengshoel, O. J. (2018). Neighborhood beautification: Graph layout through message passing. *Journal of Visual Languages & Computing*, *44*, 72–88. https://doi.org/10.1016/j.jvlc.2017.11.008

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Godiyal, A., Hoberock, J., Garland, M., & Hart, J. C. (2009). Rapid multipole graph drawing on the gpu. In I. G. Tollis & M. Patrignani (Eds.), *Graph Drawing* (pp. 90–101). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-00219-9_10

Google Inc. (2021). *Chrome* (Version 90.0.4430.93)[Computer Software]. https://www.google.com/chrome/

Google Inc. (2022). *Chrome Canary* (Version 106.0.5239.0)[Computer Software]. https://www.google.com/chrome/canary/

Hadany, R. & Harel, D. (2001). A multi-scale algorithm for drawing graphs nicely. *Discrete Applied Mathematics*, *113*(1), 3–21. https://doi.org/10.1016/S0166-218X(00)00389-9

Hadlak, S., Schulz, H., & Schumann, H. (2011). In situ exploration of large dynamic networks. *IEEE Transactions on Visualization and Computer Graphics*, *17*(12), 2334–2343. https://doi.org/10.1109/TVCG.2011.213

Hagberg, A., Schult, D., & Swart, P. (2022). *NetworkX Reference, Release 2.5*. https://networkx.org/documentation/stable/_downloads/networkx_reference.pdf

Harish, P. & Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. In S. Aluru, M. Parashar, R. Badrinath, & V. K. Prasanna (Eds.), *High Performance Computing – HiPC 2007* (pp. 197–208). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-77220-0_21

Idreos, S., Zoumpatianos, K., Athanassoulis, M., Dayan, N., Hentschel, B., Kester, M. S., Guo, D., Maas, L., Qin, W., Wasay, A., & Sun, Y. (2018a). The periodic table of data structures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, *41*(3), 64–75. https://stratos.seas.harvard.edu/files/stratos/files/periodictabledatastructures.pdf

Idreos, S., Zoumpatianos, K., Chatterjee, S., Qin, W., Wasay, A., Hentschel, B., Kester, M., Dayan, N., Guo, D., Kang, M., & Sun, Y. (2019). Learning data structure alchemy. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, *42*(2), 46–57. https://stratos.seas.harvard.edu/files/stratos/files/learningdatastructurealchemy.pdf

Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M. S., & Guo, D. (2018b). The data calculator: Data structure design and cost synthesis from first principles, and learned cost models. *ACM SIGMOD International Conference on Management of Data*, (pp. 535–550). https://doi.org/10.1145/3183713.3199671

Ioannidis, Y. E. & Wong, E. (1987a). Query optimization by simulated annealing. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, Vol. 16 of *SIGMOD '87* (pp. 9–22). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/38713.38722

Ioannidis, Y. E. & Wong, E. (1987b). Query optimization by simulated annealing. *SIGMOD Rec.*, *16*(3), 9–22. https://doi.org/10.1145/38714.38722

Jacomy, A. & Plique, G. (2022). *Sigma.js* [Computer Software]. https://www.sigmajs.org

Jargstorff, F. (2004). A framework for image processing. In R. Fernando (Ed.), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (pp. 445–467). Addison-Wesley.

Jeowicz, T., Kudelka, M., Plato, J., & Snael, V. (2013). Visualization of large graphs using gpu computing. In *2013 5th International Conference on Intelligent Networking and Collaborative Systems* (pp. 662–667).: IEEE Computer Society. https://doi.org/10.1109/INCoS.2013.126

Kahn, A. B. (1962). Topological sorting of large networks. *Commun. ACM*, *5*(11), 558–562. https://doi.org/10.1145/368996.369025

Kamada, T. & Kawai, S. (1989). An algorithm for drawing general undirected graphs. *Information processing letters*, *31*(1), 7–15. https://doi.org/10.1016/0020-0190(89)90102-6

Kelefouras, V., Kritikakou, A., Mporas, I., & Kolonias, V. (2016). A high-performance matrix-matrix multiplication methodology for cpu and gpu architectures. *The Journal of supercomputing*, *72*(3), 804–844. https://doi.org/10.1007/s11227-015-1613-7

Kester, M. S., Athanassoulis, M., & Idreos, S. (2017). Access path selection in main-memory optimized data systems: Should i scan or should i probe? *ACM SIGMOD International Conference on Management of Data*, (pp. 715–730). https://doi.org/10.1145/3035918.3064049

Khronos® Group (2014). *WebGL Specification* [Standard]. https://www.khronos.org/registry/webgl/specs/1.0

Khronos® Group (2017). *WebGL 2 Specification* [Standard]. https://www.khronos.org/registry/webgl/specs/2.0

Khronos® Group (2022a). *OpenCL™* (Version 3.0)[Specification]. https://www.khronos.org/opencl

Khronos® Group (2022b). *Vulkan*. Khronos® Group. https://www.vulkan.org/

Kirsch, A. & Mitzenmacher, M. (2006). Less hashing, same performance: Building a better bloom filter. In Y. Azar & T. Erlebach (Eds.), *Algorithms – ESA 2006* (pp. 456–467). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/11841036_42

Kobourov, S. G. (2012). Spring embedders and force directed graph drawing algorithms. https://doi.org/10.48550/arXiv.1201.3011

Kobourov, S. G. (2013). Force-directed drawing algorithms. In Tamassia & Rosen (2013), 12 12, (pp. 363–408).

Koren, Y. (2005). Drawing graphs by eigenvectors: theory and practice. *Computers & Mathematics with Applications*, *49*(11-12), 1867–1888. https://doi.org/10.1016/j.camwa.2004.08.015

Kreylos, O. & Hamann, B. (2001). On simulated annealing and the construction of linear spline approximations for scattered data. *IEEE Transactions on Visualization and Computer Graphics*, *7*(01), 17–31. https://doi.org/10.1109/2945.910818

Krüger, J. & Westermann, R. (2003). Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, *22*(3), 908–916. https://doi.org/10.1145/882262.882363

Landau, H. J. (1967). Necessary density conditions for sampling and interpolation of certain entire functions. *Acta Mathematica*, *117*(none), 37–52. https://doi.org/10.1007/BF02395039

Lee, B., Plaisant, C., Parr, C., Fekete, J., & Henry, N. (2006). Task taxonomy for graph visualization. In *AVI: Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization; 23-23 May 2006*, BELIV '06 (pp. 1–5).: ACM. https://doi.org/10.1145/1168149.1168168

Lefohn, A. E., Sengupta, S., Kniss, J., Strzodka, R., & Owens, J. D. (2006). Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, *25*(1), 60–99. https://doi.org/10.1145/1122501.1122505

Li, B., Wei, J., Sun, J., Annavaram, M., & Kim, N. (2019). An efficient gpu cache architecture for applications with irregular memory access patterns. *ACM transactions on architecture and code optimization*, *16*(3), 1–24. https://doi.org/10.1145/3322127

Li, J., Ranka, S., & Sahni, S. (2013). Gpu matrix multiplication. In S. Rajasekaran, L. Fiondella, M. Ahmed, & R. A. Ammar (Eds.), *Multicore Computing: Algorithms, Architectures, and Applications*. New York: Chapman and Hall/CRC, 1st edition. https://doi.org/10.1007/978-1-4613-9692-5_3

Lin, C. & Yen, H. (2005). A new force-directed graph drawing method based on edge-edge repulsion. In *Ninth International Conference on Information Visualisation (IV'05)* (pp. 329–334).: IEEE Computer Society. https://doi.org/10.1109/IV.2005.10

Lin, D. & Huang, T. (2021). Efficient gpu computation using task graph parallelism. In *Euro-Par 2021: Parallel Processing*, Lecture Notes in Computer Science (pp. 435–450). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-85665-6_27

Lu, J. & Si, Y. (2020). Clustering-based force-directed algorithms for 3d graph visualization. *The Journal of supercomputing*, *76*(12), 9654–9715. https://doi.org/10.1007/s11227-020-03226-w

Marsaglia, G. (2003). Xorshift rngs. *Journal of statistical software*, *8*(14), 1–6. https://doi.org/10.18637/jss.v008.i14

Marshall, S. (2004). Converting production renderman shaders to real-time. In R. Fernando (Ed.), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (pp. 551–566). Addison-Wesley.

Mathews, M. (2011). *JSDoc* (Version 3)[Computer Software]. https://jsdoc.app

Mi, P., Sun, M., Masiane, M., Cao, Y., & North, C. (2016). Interactive graph layout of a million nodes. *Informatics*, *3*(4). https://doi.org/10.3390/informatics3040023

Microsoft Inc. (2021). *Edge* [Computer Software]. https://www.microsoft.com/en-us/edge

Misue, K., Eades, P., Lai, W., & Sugiyama, K. (1995). Layout adjustment and the mental map. *Journal of visual languages and computing*, *6*(2), 183–210. https://doi.org/10.1006/jvlc.1995.1010

Mockus, A. (2019). Insights from open source software supply chains (keynote). In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019 (pp.3̃). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3338906.3342813

Mozilla Corp. (2021). *Firefox* (Version 88.0)[Computer Software]. https://www.mozilla.org/en-US/firefox

Mozilla Developer Network (2022). *The Event Loop*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

N, R. J. & Murali, S. (2019). *Memory sharing via unified memory architecture*. https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=20190730&DB=EPODOC&CC=CN&NR=110069421A

Namata, G., Staats, B., Getoor, L., & Shneiderman, B. (2007). A dual-view approach to interactive network visualization. *CIKM*. https://doi.org/10.1145/1321440.1321580

NetworkX Developers (2020). *NetworkX: Network Analysis in Python*. https://networkx.org

Nobre, C., Meyer, M., Streit, M., & Lex, A. (2019). The state of the art in visualizing multivariate networks. *Computer Graphics Forum*, *38*(3), 807–832. https://doi.org/10.1111/cgf.13728

Nobre, C., Wootton, D., Harrison, L., & Lex, A. (2020). Evaluating multivariate network visualization techniques using a validated design and crowdsourcing approach. *CHI*. https://doi.org/10.1145/3313831.3376381

NVIDIA Corp. (2022a). *cuBLAS* [Computer Software]. https://developer.nvidia.com/cublas

NVIDIA Corp. (2022b). *CUDA* (Version 11.8)[Framework]. https://www.nvidia.com/en-gb/geforce/technologies/cuda/

Nyquist, H. (1928). Certain topics in telegraph transmission theory. *Transactions of the American Institute of Electrical Engineers*, *47*(2), 617–644. https://doi.org/10.1109/T-AIEE.1928.5055024

OpenJS Foundation (2022). *Node.js* [Computer Software]. https://nodejs.org/en

Panic (2021). *Nova* (Version 10.0)[Computer Software]. https://www.nova.app

Patterson, D. A. & Hennessy, J. L. (2017). *Computer Organization and Design: The Hardware/Software Interface* (Arm ed.). Morgan Kaufmann.

Plique, G. (2022). *Graphology* [Computer Software]. https://doi.org/10.5281/zenodo.5681257

Qu, J., Liu, X., Sun, M., & Qi, F. (2017). Gpu-based parallel particle swarm optimization methods for graph drawing. *Discrete Dynamics in Nature and Society*, *2017*, 2013673. https://doi.org/10.1155/2017/2013673

Ramakrishnan, R. & Gehrke, J. (2000). *Database Management Systems* (2nd ed.). McGraw-Hill, Inc.

Salmon, J., Moraes, M., Dror, R., & Shaw, D. (2011). Parallel random numbers: as easy as 1, 2, 3. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC '11 (pp. 1–12).: ACM. https://doi.org/10.1145/2063384.2063405

Segal, M. & Akeley, K. (2022). *The OpenGL® Graphics System* (Version 4.6)[Specification]. https://www.khronos.org/opengl/

Sheng, S., Wu, C., Dong, X., & Chen, S. (2019). Research on dynamic graph layout by parallel computing and markov process. In *IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)* (pp. 1092–1098).: IEEE Computer Society. https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00156

Silva, C., Chiang, Y., Correa, W., El-Sana, J., & Lindstrom, P. (2013). *Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics, LLNL Technical Report UCRL-JC-150434-REV-1.* Technical report, Lawrence Livermore National Library.

Simonetto, P., Archambault, D., & Kobourov, S. (2020). Event-based dynamic graph visualisation. *IEEE Transactions on Visualization and Computer Graphics*, *26*(7), 2373–2386. https://doi.org/10.1109/TVCG.2018.2886901

Singh, I., Shriraman, A., Fung, W. W. L., O'Connor, M., & Aamodt, T. M. (2014). Cache coherence for gpu architectures. *IEEE Micro*, *34*(3), 69–79. https://doi.org/10.1109/MM.2014.4

Smith, A. R. (1984). Plants, fractals, and formal languages. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84 (pp. 1–10). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/800031.808571

Sorensen, T. & Donaldson, A. F. (2016). Exposing errors related to weak memory in gpu applications. *SIGPLAN Not.*, *51*(6), 100–113. https://doi.org/10.1145/2980983.2908114

Sorensen, T., Donaldson, A. F., Batty, M., Gopalakrishnan, G., & Rakamarić, Z. (2016). Portable inter-workgroup barrier synchronisation for gpus. *SIGPLAN notices*, *51*(10), 39–58. https://doi.org/10.1145/3022671.2984032

Sorensen, T., Pai, S., & Donaldson, A. F. (2019). One size doesn't fit all: Quantifying performance portability of graph applications on gpus. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, International Symposium on Workload Characterization Proceedings (pp. 155–166). New York: IEEE. https://doi.org/10.1109/IISWC47752.2019.9042139

Sorensen, T., Salvador, L. F., Raval, H., Evrard, H., Wickerson, J., Martonosi, M., & Donaldson, A. F. (2021). Specifying and testing gpu workgroup progress models. *Proceedings of ACM on Programming Languages*, *5*(OOPSLA), 1–30. https://doi.org/10.1145/3485508

Stolper, C. D., Foerster, F., Kahng, M., Lin, Z., Goel, A., Stasko, J., & Chau, D. H. (2014). Glos: Graph-level operations for exploratory network visualization. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14 (pp. 1375–1380). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2559206.2581239

Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE transactions on systems, man, and cybernetics*, *11*(2), 109–125. https://doi.org/10.1109/TSMC.1981.4308636

Tamassia, R. & Rosen, K. H. (2013). *Handbook of Graph Drawing and Visualization*. CRC Press LLC.

Tan, J., Yan, K., Song, S. L., & Fu, X. (2020). Energy-efficient gpu l2 cache design using instruction-level data locality similarity. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. https://doi.org/10.1145/3408060

The Mathworks, Inc. (2022). *MATLAB* (Version R2022b)[Computer Software].

Tufte, E. R. (1990). *Envisioning information*. Graphics Press.

Tutte, W. T. (1963). How to draw a graph. *Proceedings of the London Mathematical Society*, *s3-13*(1), 743–767. https://doi.org/10.1112/plms/s3-13.1.743

Udupa, A., Govindarajan, R., & Thazhuthaveetil, M. (2009). Software pipelined execution of stream programs on gpus. *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, *33*(1), 98–105. https://doi.org/10.1109/CGO.2009.20

Valiant, L. G. (1981). Universality considerations in vlsi circuits. *IEEE transactions on computers*, *C-30*(2), 135–140. https://doi.org/10.1109/TC.1981.6312176

Walshaw, C. (2003). A multilevel algorithm for force-directed graph-drawing. *Journal of graph algorithms and applications*, *7*(3), 253–285. https://doi.org/10.7155/jgaa.00070

Wang, Y., Pan, Y., Davidson, A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A. T., & Owens, J. D. (2017). Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing*, *4*(1), 1–49. https://doi.org/10.1145/3108140

Ward, S. A. & Halstead Jr., R. H. (1999). *Computation Structures*. The MIT Press.

Ware, C. (2013). *Information Visualization: Perception for Design* (3rd ed.). Elsevier/MK.

Web Hypertext Application Technology Working Group (2021a). *HTML: Event Loops*. WHATWG. https://html.spec.whatwg.org/multipage/webappapis.html#event-loops

Web Hypertext Application Technology Working Group (2021b). *HTML: Processing Model*. WHATWG. https://html.spec.whatwg.org/multipage/webappapis.html#processing-model

Web Hypertext Application Technology Working Group (2021c). *HyperText Markup Language* (Version 5)[Standard]. https://html.spec.whatwg.org

Wei, Y., Du, X., Xu, D., & Wang, X. (2018). A speedup spatial rearrangement algorithm for dynamic network visualization. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)* (pp. 749–753).: IEEE. https://doi.org/10.1109/DSC.2018.00120

World Wide Web Consortium (2022a). *GPU for the Web Community Group*. https://www.w3.org/community/gpu

World Wide Web Consortium (2022b). *WebGPU*. https://www.w3.org/TR/webgpu

World Wide Web Consortium (2022c). *WebGPU Shading Language*. https://www.w3.org/TR/WGSL

World Wide Web Consortium (2022d). *World Wide Web Consortium (W3C)*. https://w3c.org

Xiao, Y., Feng, R., Han, Z., & Leung, C. (2015). Gpu accelerated self-organizing map for high dimensional data. *Neural processing letters*, *41*(3), 341–355. https://doi.org/10.1007/s11063-014-9383-4

Xu, T., Yang, J., & Gou, G. (2018). A force-directed algorithm for drawing directed graphs symmetrically. *Mathematical problems in engineering*, *2018*, 1–24. https://doi.org/10.1155/2018/6208509

Zhong, J. & He, B. (2014). Medusa: Simplified graph processing on gpus. *IEEE transactions on parallel and distributed systems*, *25*(6), 1543–1552. https://doi.org/10.1109/TPDS.2013.111