FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Attribute Grammars for Incremental Evaluation of Scene Graph Semantics

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Harald Steinlechner
Matrikelnummer 0825851

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr. techn. Werner Purgathofer
Mitwirkung: Dipl.-Ing. Dr. techn. Robert F. Tobler

Wien, TT.MM.JJJJ

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Attribute Grammars for Incremental Evaluation of Scene Graph Semantics

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Harald Steinlechner
Registration Number 0825851

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr. techn. Werner Purgathofer
Assistance: Dipl.-Ing. Dr. techn. Robert F. Tobler

Vienna, TT.MM.JJJJ

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Harald Steinlechner
Hofmannsthalgasse 24/14/22, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I want to thank *Robert F. Tobler* and *Stefan Maierhofer* for giving me the chance to work on the *AARDVARK* rendering engine since its early days, as well as for mentoring me the last couple of years.

Special thanks to *Georg Haaser* for algorithms, proofs and other support which was a crucial point in finishing this work. I want to thank *Michael Wörister* for his seminal diploma thesis and work in incremental scene graph evaluation.

Also, I want to thank the nice ladies from the diploma students room for cuddling and drinking tea several times per day — this would not have been possible without you.

Thanks to my dearest reviewers, especially *Stefan Maierhofer*, *Christian Luksch*, *Josef Eisl*.

Finally, I want to thank my parents *Ida* and *Gerhard* for all the mental and financial support, which made my studies possible.

Abstract

Three dimensional scenes are typically structured in a hierarchical way. This leads to scene graphs as a common central structure for the representation of virtual content. Although the concept of scene graphs is versatile and powerful, it has severe drawbacks. Firstly, due to its hierarchical nature the communication between related nodes is cumbersome. Secondly, changes in the virtual content make it necessary to traverse the whole scene graph at each frame. Although caching mechanisms for scene graphs have been proposed, none of them work for dynamic scenes with arbitrary changes. Thirdly, state-of-the-art scene graph systems are limited in terms of extensibility. Extending framework code with new node types usually requires the users to modify traversals and the implementations of other nodes. In this work, we use *attribute grammars* as underlying mechanism for specifying scene graph semantics. We propose an embedded domain specific language for specifying scene graph semantics, which elegantly solves the extensibility problem. Attribute grammars provide well-founded mechanisms for communication between related nodes and significantly reduce glue code for composing scene graph semantics. The declarative nature of attribute grammars allows for side-effect free formulation of scene graph semantics, which gives raise for incremental evaluation.

In contrast to previous work we use an expressive computation model for attribute grammar evaluation which handles fully dynamic scene graph semantics while allowing for efficient *incremental evaluation*.

We introduce all necessary mechanisms for integrating incremental scene graph semantics with traditional rendering backends. In our evaluation we show reduced development effort for standard scene graph nodes. In addition to optimality proofs we show that our system is indeed incremental and outperforms traditional scene graph systems in dynamic scenes.

Kurzfassung

In CAD-Software und Videospielen finden häufig sogenannte *Szenegraphen* Anwendung, um die Relationen der Daten in einer Szene zu beschreiben. Durch die hierarchische Anordnung der Knoten eines solchen Szenegraphen können Attribute und deren Gültigkeitsbereiche, wie beispielsweise affine Transformationen oder Texturinformationen, einfach und konsistent modelliert werden.

Szenegraphsysteme erlauben zwar durch ihre hohe Ausdrucksstärke die Modellierung von dynamischen Sachverhalten, wie etwa Animation und Simulation, die Programmierung davon gestaltet sich jedoch häufig als schwierig und ineffizient.

In dieser Arbeit zeigen wir Probleme von bestehenden Systemen auf, synthetisieren Design Ziele und entwickeln ein Szenegraph Konzept, welches diese Probleme auf neuartige Art und Weise löst.

Das hohe Abstraktionsniveau unseres Systems erlaubt eine effizientere Programmierung von interaktiven Rendering Applikationen.

Die verbesserte Programmierbarkeit wurde anhand eines Anwendungsfall demonstriert. Dazu wurde der Programmieraufwand in unserem System mit dem von alternativen Implementierungstechniken verglichen.

Anhand von synthetischen Testszenen zeigen wir, dass unser vorgestelltes System trotz des hohen Abstraktionsniveaus, auch in dynamischen Szenen bessere Performance erreicht als bestehende Systeme.

Contents

CHAPTER 1

Introduction

Software development of interactive three dimensional graphics applications is tedious, time consuming and complex due to the high number of concerns regarding interaction logic and low level graphics API interaction. Although abstractions like *RenderMan* [52] or OpenGL [60] help developing pure rendering applications, middleware for interactive applications still suffers from a discrepancy between programmer convenience and performance. In order to fulfill performance requirements, programming at low abstraction level is still predominant in high-performance rendering engines.

Three dimensional scenes are usually hierarchically structured which leads to scene graphs as central data structure for describing virtual content. Many toolkits, such as Inventor [45], OpenSceneGraph [8] and SceniX [30], employ this concept. In this section we review scene graphs as a central data structure for rendering applications. Subsequently, we discuss shortcomings of current solutions based on scene graphs, which eventually gives motivation for this thesis. Finally, we give a short overview of the approach taken in this thesis.

1.1 Scene Graphs

A scene graph can be characterized as a directed acyclic graph (DAG). Leaf nodes typically represent geometric entities. Inner nodes basically change the appearance of nodes within its subgraph. Transformation nodes for example change spatial attributes of child nodes i.e. transform contained geometries. From an abstract point of view each node type has some defined meaning respective to rendering. Surface nodes for example change the appearance of contained nodes, while camera nodes specify viewpoint and projection.

In this work we call the meaning of node types *semantics*, defined by *semantic functions*. Informally, semantic functions describe how to compute attributes for each node type in the graph. Given semantic functions for all node types we can compute the meaning of a scene graph i.e.

the rendering result for a given scene graph. The concept is depicted in Fig. 1.1.

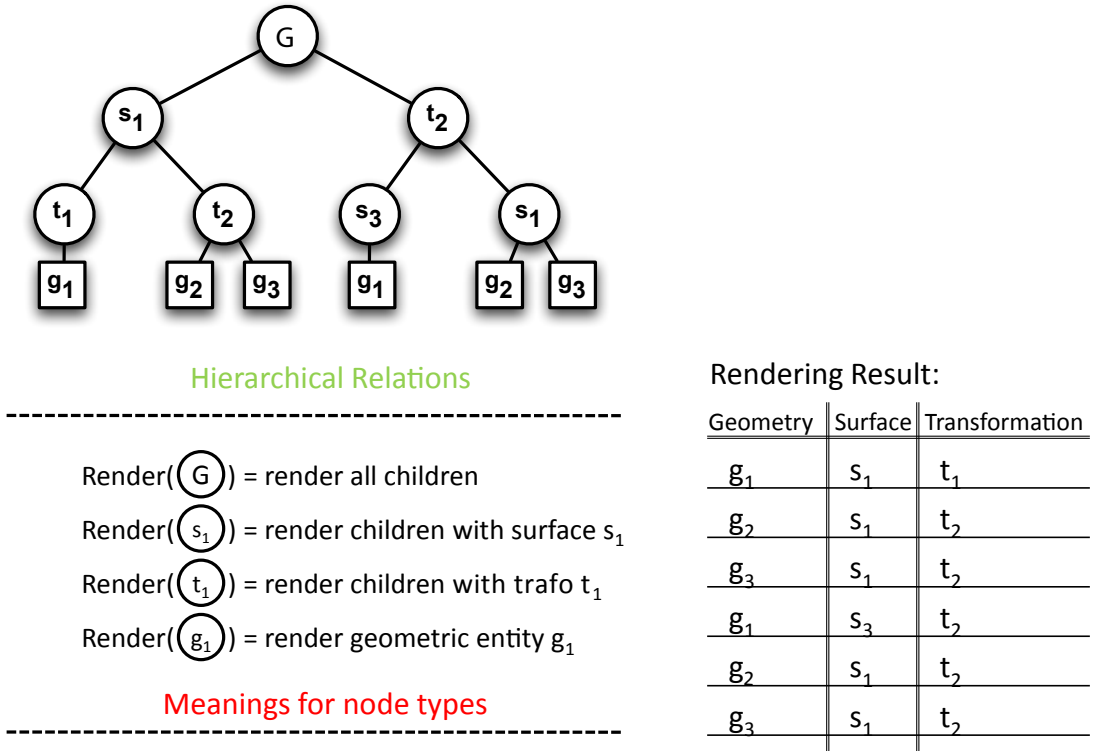


Figure 1.1: Abstract description of scene graphs. Separated from its actual representation semantic functions describe the meaning of each node type. Given a scene graph and semantic functions for each node type we compute the aggregate semantics, which corresponds to the rendering result of the scene graph.

Traditional scene graph systems typically traverse the scene graph in depth-first manner in order to produce the rendering result. In this implementation model each node type executes its semantics by directly modifying the rendering state. Leaf nodes in turn issue graphics commands to the graphics API.

1.2 Motivation

Although scene graphs seem natural for modeling graphical content, state-of-the-art systems have several drawbacks:

1. Due to its hierarchical nature **communication** between related nodes becomes difficult because child nodes are inherently dynamic and cannot be determined statically. There-

fore, instead of calling methods of other nodes directly, values need to be transferred via environment dictionaries or other shared data structures.

2. Scene graph systems often suffer from limited extensibility — various files need to be modified in order to add new node or operation types to the system. As we will see in section 3 this **extensibility** is not easy to accomplish with standard object-oriented programming.

3. Since scene graphs are dynamic in general, proper **memory management** becomes a burden for the application programmer because each eventually traversed scene graph needs to be properly disposed if it becomes inactive.

4. To be useful, nodes typically implement custom semantics and perform **arbitrary effects** within the scene graph. Such effects include removal of sub-nodes or modification of fields within other nodes. Note that a disciplined data flow and communication concept (1) would reduce the need for arbitrary effects. For complex interaction logics however, rendering semantics still need to perform arbitrary side-effects in general (e.g. a node changes appearance parameters of some geometry node).

5. Scene graph traversal is inherently **inefficient** for static scenes: Although nothing changes, all nodes need to be traversed because changes could possible occur due to (4).

   To overcome this efficiency problem scene graph caching was introduced in previous work (e.g. Durbin et al. [15] and recently Wörister [61]). In order to handle updates, usually modeled with side-effects (4), Wörister for example, captures dependencies in the scene graph using a dependency graph. The proposed computation model, based on dependency graphs works well for value changes within the scene graph (for example moving objects), but is not efficient for structural changes like additions or deletions. In this case cacheable units, called rendering caches are invalidated and need to be recomputed from scratch.

   As noted by Wörister et al. [62] the computation model needs to be extended and generalized in order to support proper incremental updates for structural changes.

The motivation for this thesis is to eliminate these drawbacks by utilizing attribute grammars as a declarative way for specifying data and its hierarchical relation. As we shall see, attribute grammars offer well-founded mechanisms, effectively solving (1), (2), (3) and (4).

Finally, in order to solve the efficiency problem, evaluation of scene graph semantics needs to be incremental. For this purpose we generalize the computation model as used by Wörister et al. [62] to work with structural updates in an efficient manner. We show, that our computation model integrates well with attribute grammars ultimately solving (5) while maintaining the clean separation of syntax and semantics provided by attribute grammars.

1.3 Aim of this Work

The proposed system allows for incremental evaluation of scene graph semantics in fully dynamic scenes. Although incremental evaluation has been applied to scene graph evaluation

(e.g. [15, 61, 62]), previous approaches could not handle arbitrary changes in a uniform and clean manner.

In order to allow for arbitrary dynamic scene content, we use a very disciplined approach for authoring scene graph semantics by utilizing attribute grammars, a concept known from compiler theory. This formulation has several benefits including simpler scene graph nodes as well as better performance compared to traditional scene graph systems. A detailed description of the goals and the scope of the thesis is given in section 3.7.

1.4 Methodological Approach

In order to validate our approach, we have implemented a prototype including implementations for most important scene graph nodes. This includes a computation model for incremental evaluation of scene graph semantics as well as a domain specific language for attribute grammar authoring.

We show decreased implementation effort for standard scene graph nodes. To this end we compare our implementation and structure of typical scene graph nodes with traditional scene graph systems.

Additionally, we analyze the incremental complexity of our update algorithms and show performance characteristics.

1.5 Structure of this Work

This thesis is structured as follows:

- Chapter 2 gives an overview of three related areas: *Scene Graph Systems*, *Incremental Computation*, *Attribute Grammars and Systems*.
- Chapter 3 analyzes state-of-the-art scene graph systems and identifies shortcomings and limitations.
  - *Our Approach* gives a short overview of the approach taken in this thesis.
  - Rendering engines are complex software systems. *Scope of this work* defines how the proposed system relates to other modules of rendering engines.
- Chapter 4 summarizes concepts used in this thesis:
  - *Lazy Incremental Scene Graph Caching* describes the computation model used in previous work for incremental scene graph evaluation.
  - In *Limitations of Lazy Incremental Computation* we give an overview on *lazy incremental caching* and discuss its limitations.
  - *Adaptive Functional Programming* introduces an alternative computation model, which handles all types of changes in a uniform manner.

  - In *Problem solving with Attribute Grammars* we give a short introduction to attribute grammars. More specifially, we show how to use attribute grammars to solve simple programming tasks involving recursive data structures.

- Chapter 5 is structured as follows:
  - In *Design Goals* we define design goals for our system.
  - *Key Design Decisions* explores the design space for scene graph system implementations and synthesizes some design decisions in order to achieve the design goals defined previously.
  - *An EDSL for Attribute Grammars* introduces an implementation technique for attribute grammars integrated in our implementation language F#.
  - *Attribute Evaluation* describes an implementation technique of attribute grammars.
  - *Adaptive Functional Programming* introduces a domain specific language which we use to model incremental computations.
  - *Rendering Engine Integration* shows how to integrate incremental evaluation of scene graph semantics with rendering backends, as found in typical rendering engines.
  - In *Scene Graphs as Attribute Grammars - Nodes and Semantics* we present a simple scene graph system based on attribute grammars.
- Chapter 6 investigates the performance of our system for synthetic scenes modeled with incremental attribute grammars.
- Chapter 7 brings the thesis to its conclusion and points out future work.

CHAPTER 2

Related Work

This thesis is related to visual computing, software engineering as well as programming languages. First of all we will give an overview on design principles and state-of-the-art *scene graph systems* (2.1). Next, since this thesis is strongly related to topics from the computer languages community as well we will give an overview on *attribute grammars* (2.2) and *incremental computation* (2.3).

2.1 Scene Graph Systems

Software development of interactive three dimensional graphics applications is tedious, time consuming, and complex due to the high number of concerns regarding interaction logic and graphics API interaction. Early abstraction mechanisms, such as *OpenGL* [60] and *RenderMan* [52] mainly focus on rendering. Especially integration with interactive concerns can be cumbersome and limited in functionality. Researchers proposed various techniques for improving programmability and expressiveness of graphics applications. In the following we will give an overview on middleware systems providing rich abstraction mechanisms for interactive rendering applications.

Strauss and Carey introduced *Inventor* [45], an object-oriented toolkit for graphics applications which was specifically designed with interactive applications in mind. A scene database stores the scene as a directed acyclic graph where nodes represent graphical entities.

Crucially, Inventor provides extensibility — application programmers can extend the system with new node types as well as new rendering methods i.e. traversals.

Inventor uses so called traversal *actions* for specifying rendering semantics. Actions may also be used to perform other specific operations such as computing bounding boxes or picking. For user interaction *Inventor* uses events, which are deeply integrated in the system — events basically traverse the scene graph just like other actions. Scene nodes in turn may handle events themselves or pass them to its subgraph. Interactive scene nodes, depending on user events may

manipulate other parts of the scene. For example a keyboard event may trigger a trackball node to become active, eventually taking over control of the scene.

OpenInventor [59] extends *Inventor* with support for animation. In addition to events and manipulation nodes as used by *Inventor*, *OpenInventor* uses so called *engines* to model dynamic behaviors. Engines essentially compute some output depending on various dynamic input fields. For example consider a scene graph describing the geometry of a windmill. Transformation nodes may now be connected to an engine which computes rotation angles, effectively moving the windmill depending on external properties like wind force.

Similarly to Inventor, *IRIS Performer* [41] stores scene nodes in a scene database, represented as a graph. In contrast to previous approaches, *IRIS Performer* focuses on optimization and multiprocessing. Due to lack of parallelization and optimization, early scene graph systems suffer from unsatisfactory performance because traversal and application code fails to exploit parallelism provided by graphics hardware. In order to overcome that issue, *IRIS Performer* splits up independent traversals into separate processes which leads to better hardware utilization. Rendering processes divide into culling, rendering and picking. The level of parallelism is limited to independent traversals, thus the system works best in scenarios with multiple render tasks which perform culling and rendering in parallel via a multi-process architecture. Furthermore *IRIS Performer* reduces graphics mode changes in the graphics subsystem by managing graphics state explicitly.

OpenSG [35, 36, 56] has a strong focus on performance as well. Although *IRIS Performer* provides parallel render tasks, there is no model for concurrency at the data level. While multiple render tasks operate on shared data structures, the burden of synchronization is mostly left to the user. Especially with rendering tasks running at different refresh rates this problem becomes troublesome when keeping shared data consistent. *OpenSG* replicates data used by multiple threads automatically. Each traversal therefore works on a private copy of the data. Eventually synchronization makes replicas consistent again. Efficiency is maintained by the assumption that most parts of the scene are static and therefore synchronization is rare.

Additionally OpenSG performs scene graph rewriting in order to speed up traversals by only executing traversal actions for nodes significant to the specific traversal.

Similarly to *IRIS Performer*, *OpenSceneGraph* [8] separates culling from rendering and has support for parallel execution of independent tasks. As other previously mentioned approaches, *OpenSceneGraph* has a static view of scene graphs, i.e. content generation is orthogonal to the scene graph system. In order to maintain efficiency while allowing external in place modification of scene graphs, scene graph parts may be marked as *static*, which allows for common optimizations like state sorting and graph rewriting.

NVIDIA *SceniX* [30] is an object-oriented approach to scene graphs with support for custom traversals, i.e. operations on scene graphs. The framework provides a rich set of optimizations, mostly realized with specialized traversals. *Combine-Geometry* traversals for example can be

used to group multiple geometric entities into more efficient, packed and pre-transformed geometries. Other optimizations include state sorting and redundancy removal. Notably, the system supports concurrent traversals. All synchronization is on an per-object basis and mostly up to the user. Rendering states are encapsulated within state sets, which need to be accessed in mutual exclusion using locks, maintained by the state set. Although the design is flexible and extensible, there is no dedicated mechanism for data-flow within scene graph nodes.

Ideally scene graphs model merely scene entities and their hierarchical relationship. Operations, i.e. semantics are provided by traversal objects while the data model remains free from state and rendering-specific behavior. Previously mentioned approaches however inherently mix rendering state and high-level node description. In SceniX for example, application programmers directly modify the rendering states living in scene graph nodes.

In rendering applications with complex interaction logic, this design principle has severe drawbacks. First of all, applications need to store explicit references into scene graph data structures in order to modify rendering states. Thus, application programmers need a way to organize these references (see Fig. 2.1). Not surprisingly their structure is similar to the scene graph structure itself, which leads to duplication. Additionally, with concurrent traversals scene graph nodes and their content need to be synchronized.

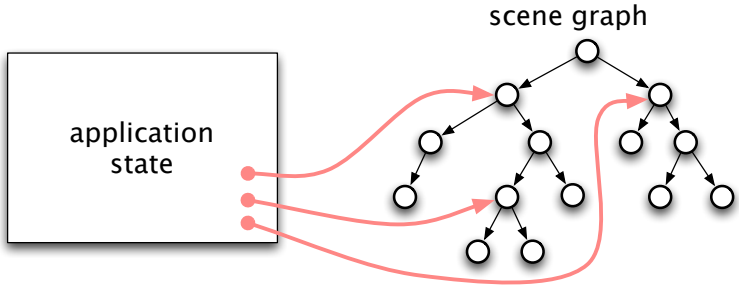


Figure 2.1: The application maintains references to scene graph nodes for external modification and application logic. Application programmers need to organize these references which leads to duplicated effort (taken from Tobler [51]).

Alternatively, it is possible to maintain application state in scene graph nodes directly (see Fig. 2.2). This design principle avoids duplication effort as mentioned before but suffers from another problem when it comes to complex application logic: Since scene graph nodes can be seen as a model for different aspects (e.g. animation, interaction, simulation) represented by traversals, application state is multifaceted as well. With complex application logic and rendering state scattered across multiple scene graph nodes maintenance becomes difficult.

In his work, Tobler [51] takes a fundamentally different approach. Instead of working with a single representation for high level semantics and rendering, Tobler separates between two kinds of scene graphs:

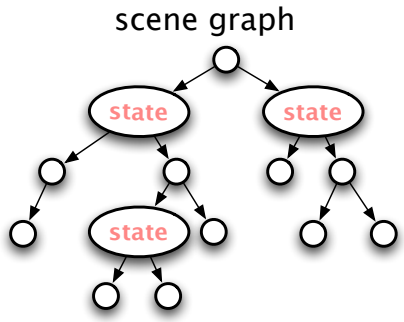


Figure 2.2: State stored directly in the scene graph (taken from Tobler [51]).

The semantic scene graph embodies the scene graph as the user modeled it.

The rendering scene graph manages graphics API state, and issues draw calls.

Similarly to on-demand compilation in just-in-time compilers, semantic nodes are expanded into rendering scene graph nodes. When traversing semantic scene graph nodes, translation rules modify the output scene graph or create new content depending on application state stored within the translation rule. Tobler compares this architecture to the well-known Model-View-Controller (MVC) design pattern. The semantic scene graph corresponds to the model, stateful translation rules to the controller and the rendering scene graph can be seen as the view component (see Fig. 2.3). With application logic living in its own environment, state management becomes much cleaner. Additionally on-the-fly expansion semantics allows for dynamic scenes, while maintaining separation of data and operations.

Although on-the-fly translation is powerful, the approach suffers from slow traversals due to the extra level of indirection. As noted by Wörister et al. [62] scene graph rendering quickly becomes CPU bound due to high traversal cost.

In their work Wörister et al. utilize incremental computation to speed up scene graph evaluation. In order to track dependencies, a dependency graph is maintained. Traditionally application state is modeled using mutable state and traversals perform arbitrary side-effects on state while evaluating semantics of nodes. Since arbitrary state cannot be captured by the dependency system, their system does not work efficiently for fully dynamic scenes.

2.2 Attribute Grammars and Systems

In this section we give a short informal overview of attribute grammars. Next we describe state-of-the art attribute grammar systems in the context of functional as well as object-oriented programming. Although powerful, attribute grammars have limited expressiveness. In the subsequent section we give an overview of extensions to attribute grammars, making attribute grammars suitable for a wider range of applications.

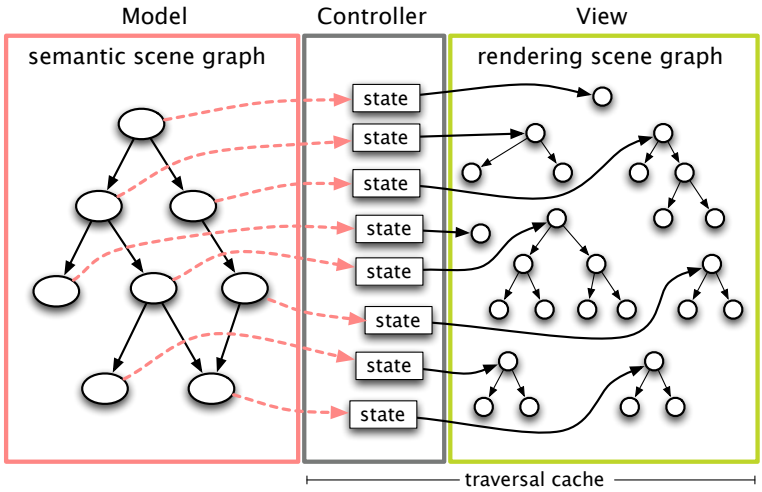


Figure 2.3: Stateful rule objects translate semantic scene graph nodes into rendering scene nodes. The architecture can be seen as the well-known Model-View-Controller design pattern (taken from Tobler [51]).

2.2.1 Attribute Grammars

The *attribute grammar* (AG) formalism was originally introduced by Knuth [26]. Knuth used the concept to assign 'meaning' to strings of some language. While *context-free grammars* offer a formalism for defining the syntax of programming languages, attribute grammars can be used to define their semantics.

Attribute grammars have a long history in compiler technology in the field of compiler compilers (e.g. [23, 39]) and are common for specifying static semantics of programming languages i.e. implementing static analyzers (e.g. [14]).

Informally an attribute grammar consists of a context-free grammar with one extension: Symbols of the grammar are equipped with attributes. Additionally, each production is equipped with attribute equations for each attribute defined in the grammar. So for productions of the form $p : X_0 \rightarrow X_1 ... X_n$ where X_i denotes an occurrence of a grammar symbol, each non-terminal symbol is associated with a set of attributes. Each production p is associated with a set of attribute equations. Attribute equations define attribute values in terms of other attribute values or constants. We call these equations *semantic functions*. Attributes can be of two types:

Synthesized attributes are attributes defined by attribute occurrences on the right-hand side of productions i.e. information flows upwards in the deviation derivation tree.

Inherited attributes are attributes defined by attribute occurrences on the left-hand side of production i.e. information flows downwards in the deviation derivation tree.

Fig. 2.4 defines syntax and semantics of a simple expression based language. The language contains expressions, which are either a numeric *literal*, an *addition* or an *if expression*. In the

⟨*expr*⟩ ::= ⟨*integer*⟩
  | ⟨*expr*⟩ '+' ⟨*expr*⟩
  | 'if' ⟨*expr*⟩ 'then' ⟨*expr*⟩ 'else' ⟨*expr*⟩

(a) *Concrete syntax.*

```
expr → integer { expr.value = parseInteger(integer)    }
expr → expr0 + expr1 { expr.value = expr0.value + expr1.value }
expr → if expr0
         then expr1
         else expr2    { expr.value = if expr.value = 1
                                      then expr1.value
                                      else expr2.value }
```

(b) Attribute grammar of the language. Each production of the grammar is associated with a *semantic function* for evaluating expressions. Semantic functions are enclosed by curly braces. Each semantic function computes the numeric result for the *synthesized* attribute `value`.

Figure 2.4: Syntax and Semantics of a simple expression based language. Concrete syntax is defined by a grammar in *EBNF* (top). The attribute grammar (bottom) defines abstract syntax as well as semantics for evaluating expressions.

attribute grammar we annotate the abstract syntax with appropriate semantics.

As noted by Swierstra [48], attribute grammars can be used for a wide range of programming tasks operating on hierarchical data. Attribute grammar systems take care of attribute evaluation order by performing dependency analysis. Thus programmers can focus on semantics instead of taking care of traversal objects or visitors.

2.2.2 Attribute Grammar systems

The synthesizer generator [38], a tool for automatically generating editors from language descriptions first used attribute grammars for incremental static analysis [38]. In this context Demers et al. developed foundations for incremental evaluation [13, 37]. Reps et al. used static analysis of attribute grammars to determine optimal execution plans for attributes [37].

The Utrecht University Attribute Grammar Compiler (UUAG) [47] is an attribute grammar system in the setting of purely functional programming.

The system compiles an attribute grammar description, consisting of syntax and semantics into catamorphisms[1] implemented in the purely functional programming language Haskell. In contrast to attribute evaluation systems which perform static analysis in order to generate execution plans (e.g. [37]), UUAG works dynamically and evaluates attributes in an on-demand

[1] A *catamorphism* is a generalization of folds for lists [27]

manner. Surprisingly, lazy evaluation makes this dynamic approach efficient. As observed by Augusteijn [5] the dynamic attribute evaluation coupled with lazy evaluation is typically just as fast as static evaluation plans.

Additionally, Haskell's type system guarantees semantic functions to be pure. This allows for folding multiple traversals into single ones, since traversals are guaranteed not to interfere with each other. With this optimization, the famous *RepMin*[2] problem for example can be solved with one single traversal.

The JastAdd system [17] extends Java with support for Rewritable Circular Reference Attributed Grammars (ReCRAGs) [16]. Again, an attribute grammar definition is compiled to a target language – multiple aspects are 'weaved' into Java classes containing attribute evaluation code.

Both approaches, JustAdd and UUAG compile the attribute grammar specification at once. It is therefore not possible to extend a compiled program with syntax and semantics without recompiling the complete grammar, which requires full source code and breaks extensibility. Recent work shows how to extend the UUAG system with aspects [53]. The approach allows for separate compilation of different aspects contributing syntax and semantics.

Although the approach allows for extensibility while maintaining separate compilation, still type-checking is performed after code-generation, which leads to cumbersome type errors in automatically generated code. Slone et al. [42] proposed an embedded domain specific language for attribute grammars in the object-oriented programming language Scala. Instead of compiling attribute grammar specifications to some general purpose language, attribute grammars can be defined directly in the host language Scala. The approach however fails to provide static type safety. Instead the system uses runtime checks to ensure proper typing.

Viera et al. [54] show an embedding in Haskell. By utilizing type-level programming the approach is type safe i.e. no runtime typecasts are necessary.

2.2.3 Extensions to attribute grammars

Attribute grammars as introduced by Knuth are powerful, but sometimes limited in their expressiveness. One limitation of canonical attribute grammars becomes apparent if attributes depend on other properties far away in the tree. In this case all required attributes need to be passed as synthesized attribute to some common super node. Subsequently, they need to be inhereted down the path to the actual client code.

Various approaches extend attribute grammars to support non-local attributes [19, 20, 55]. *Door attribute grammars* use special nonterminals, called *doors*, which allow dereferencing of remote attributes, i.e. attributes far away from the current nonterminal [19]. *Reference attribute grammars* allow direct references to other syntax nodes, which can in turn be used to access remote attributes [20].

[2] Given some integer input list, compute its minimum element and replace each element with its minimum. As shown by Bird [7], the problem can be solved using one single traversal by using circular programs.

Canonical attribute grammars work for tree-like structures. This is no problem for semantic analyzers operating on parse trees. In applications like control flow analysis, structures tend to be graphs instead of trees. This limitation can be solved by reference attribute grammars as well. In our work, sharing of common scene graphs is crucial for performance. Instead of working with reference attribute grammars we expand DAGs into trees in an on-the-fly manner. Thus, attributes of shared nodes are in fact replicated, but internally point to shared structures (e.g. vertex arrays).

As shown by Soderberg et al. [43] it is possible to extend attribute grammar evaluation to work with dynamic dependencies. In our work, we take a different approach by strictly separating incremental evaluation from attribute evaluation, thus canonical attribute grammars suffice for our purpose while providing dynamic semantics.

2.3 Incremental Computation

In the subsequent sections we handle both, incremental attribute grammar evaluation as well as general purpose incremental computation.

Let us first define the term *incremental computation*. In their survey on incremental computation Ramalingam and Reps give a concise definition:

> The abstract problem of incremental computation can be phrased as follows: The goal is to compute a function f on the user's 'input' data x — where x is often some data structure, such as a tree, graph, or matrix and to keep the output $f(x)$ updated as the input undergoes changes. [34, p.1]

Historically incremental computations first appeared in the context of syntax directed editors. A syntax-directed editor is a tool for interactive program development. Syntactic analysis, i.e. the construction of a context-free derivation tree as well as semantic analysis i.e. the evaluation of semantic functions defined by an appropriate attribute grammar should be fast in order to keep the system interactive. Attribute grammars, as underlying formalism for syntax directed editors was first introduced by Demers et al. [13]. The main advantage of using attribute grammars instead of procedural code is the applicative style of expressing semantics [13]. Semantic attributes propagate through the tree in the formalism and need not be specified imperatively. Due to its structure and functional style, attribute grammars offer opportunities for incremental evaluation.

Reps et al. [37] introduced an update propagation algorithm for attribute grammars which they show to be optimal i.e. the algorithm runs in $\mathcal{O}(|\mathrm{AFFECTED}|)$ where $\mathrm{AFFECTED}$ denotes the number of changed attributes in the complete tree.

Later the algorithm was generalized by Hover [21] outside the domain of attribute grammars. Both approaches are based on the central concept of a static dependency graph which can either be synthesized out of attribute grammars or constructed by hand. Static dependency graphs may

be modified structurally in an explicit modification step. During change propagation however, the dependency graph remains static. Given a set of changes of the graph, change propagation ensures a consistent state for all attributes.

Although Reps's algorithm is theoretically optimal with respect to re-computation of attributes after modification, it does not perform well for non-strict semantic functions. Consider a semantic function which conditionally evaluates an expensive attribute depending on some other attribute. In this case Reps's algorithm computes the expensive attribute independently of the result of the condition, i.e. computing the expensive attribute is necessary.

Hudson [22] introduces a demand-driven algorithm for incremental attribute evaluation. In this way the wasted work of computing values that are never actually used is avoided automatically. With its lazy update scheme the algorithm loses optimality but performs better in the presence of non-strict semantic functions.

Other approaches to incremental computation are *function caching* and *partial evaluation*. Function caching tries to reuse previously computed sub solutions whenever appropriate. As a consequence, the complete computation needs to be reevaluated from scratch in order to find parts which can be reused. Dependency graph based solutions on the other hand try to reuse the same computation and its intermediate results directly. With function caching however, sub computations can be used in other contexts which is not possible for dependency based approaches. Another problem with function caching is *equality*. In order to cache function calls, its arguments need to be compared to previous arguments in order to reuse previous results. Additionally it is not clear when to evict old results from the cache. *Partial evaluation* (e.g. [46]), another technique for incremental computation does not require equality and does not require cache invalidation. The approach however is not general purpose and limited in its applicability. A comprehensive summary on incremental computation techniques can be found in [34].

In their seminal work Acar et al. [2] introduce *adaptive functional programming*, a framework for incremental computation. In contrast to previous approaches, the concept allows for incremental evaluation of arbitrary functional programs. Unlike function caching, *adaptive functional programming* uses a dependency graph in order to track dependencies precisely. In contrast to previous approaches based on dependency graphs (e.g. Reps et al. [37]), Acar et al. maintain the dependency graph implicitly and transparent to the programmer. In their system, the program automatically constructs the dependency graph when running the program for the first time. Modifications of variables directly modify the dependency graph, thus graph modification is no longer sperated from change propagation.

Although *Adaptive Functional Programming* tends to provide good incremental performance for some programs, it fails to provide incremental performance for others. Later, in his PhD thesis Acar extends his approach with *selective memoization* and introduces *Memoized dependency graphs* which provide incremental performance for *trace-stable* programs [1]. In their work, Acar et al. give a full implementation of adaptive functional programming implemented in ML, i.e. no compiler support is necessary. However, his library dictates programmers to write adaptive programs in destination passing style. In order to provide a simpler user interface Carlson

introduced a monadic interface on top of Acar's adaptive primitives [10].

Later, Acar et al. introduced *Delta-ML*, an extension to *Standard ML* with built-in support for adaptive computation [4].

CHAPTER 3

Limitations of current approaches and goals of this work

State-of-the-art scene graph systems have shortcomings in both, *design* and *performance*. In this section we analyze the design space, show limitations of current approaches and how resulting problems interact with each other.

3.1 Traditional Scene Graph Rendering

In traditional scene graph systems (e.g. [41, 45, 59]) scene graph nodes and their relation are stored in a scene database. For efficiency reasons, nodes may relate to common sub-nodes, resulting in directed acyclic graphs (DAG).

Leaf nodes typically represent geometric entities containing vertex data like positions and texture coordinates. Internal nodes describe a number of different *attributes* like spatial transformation or surface properties. Attributes are inherited along the edges of the graph until they reach leaf nodes — a central mechanism for communication between scene graph nodes. Leaf nodes in turn use these attributes as arguments for draw calls. Inner nodes publish their attributes by modifying a mutable traversal state which is threaded through the traversal function (which we call *RenderTraversal*). A scene graph can be rendered by traversing it in a depth-first manner.

In flexible scene graph systems with dynamic scene graphs (e.g. Tobler's *semantic scene graph* [51]) nodes may additionally generate dynamic content or modify subgraphs. *Switch nodes* for example return different subgraphs depending on some predicate. In order to support dynamic scene graphs properly, it is necessary to compute aggregate values for subgraphs. Level of detail nodes for example need to access information (e.g. bounding boxes) of subgraphs in order to guide level of detail decisions. The need for computing aggregate values motivates a generalization of the traversal concept. Instead of issuing draw calls to the graphics device, traversals can be used to compute aggregate values for scene graphs. User defined traversals are

central to SceniX [30] and OpenSceneGraph [8]. Both systems use traversals for graph rewriting optimizations like geometry batching. Since scene graph systems should be highly reusable in many problem domains, we require scene graph systems to be extensible respective to node types as well as traversals i.e. application developers need a way to specify custom scene graph nodes and associated operations.

3.2 Problems of current Approaches

Next we will review fundamental design flaws in current scene graph systems.

3.2.1 Extensibility

In order to support extensibility, a main concern for scene graph systems, we require:

Data-Extensibility For application programmers it must be easy to add *additional node types* fitting domain specific needs. As an example consider a new camera node for stereo rendering.

Operation-Extensibility Additionally it shall be easy to add *additional traversals* in order to implement dynamic behavior of new node types. As an example consider a special traversal which extracts metadata from subgraphs which serves as basis for level of detail decisions.

Interestingly these requirements are very similar to the *Expression Problem* [57], a well known benchmark for expressiveness in language design (see appendix A). A valid solution for the expression problems allows for data-extensibility as well as operation-extensibility while maintaining separate compilation and type safety.

In static programming languages it is hard to provide this kind of extensibility. Surprisingly object-oriented programming per se does not provide a valid solution for that problem. Consider a type hierarchy: adding additional operations (methods) breaks extensibility because all node types need to be adapted in order to implement the additional method required by their common supertype[1].

From a theoretical point of view this functionality can be elegantly achieved by object-oriented double dispatch as provided by CLOS [44]. Unfortunately double dispatch, or multi-methods in general are not supported by current object-oriented mainstream languages.

Inventor [45], implemented in C++ uses a two dimensional matrix which stores a traversal function for each node/traversal combination[2]. Although flexible, the approach does not provide full type safety. Traversal results need to be coerced unsafely in order to be used in strongly typed node implementations.

[1] see appendix A for in depth explanation of non-solutions to the *expression problem*.

[2] Multi methods provide this functionality directly.

Instead of resorting to dynamic typing, OpenSceneGraph [8] uses the well-known visitor pattern [18] to model scene graph traversals.

The visitor pattern can be seen as an approximation to double dispatch in mainstream languages. New traversals can be implemented by simply adding another class, implementing the visitor interface. For new node types however, extensibility is violated — in order to add another new node type one needs to modify the visitor interface and all its implementations [11]. Note that with subtyping, default implementations for node types may be provided, which significantly reduces code duplication in visitor implementations [29]. Still, in general the visitor pattern does not provide proper extensibility i.e. introducing new node types is hard, involving adaptation in many actually unrelated source files.

In his scene graph system, Tobler [51] introduces a clean separation of *Semantics from Rendering* which allows for a clever implementation basically solving the expression problem except for semi-safe casts in the core implementation. Tobler identifies the interleaving of high level semantics and low level rendering setup as the main root of scene graph complexity. Therefore he distinguishes between a semantic description of the scene graph and its actual representation for rendering.

In his design, the user models the scene graph in the semantic scene graph. Subsequently the traversal algorithm generates the rendering scene graph from the semantic scene graph by expanding semantic nodes in an on-demand manner. The semantics for this expansion is defined by a user defined mapping for each semantic scene graph node. In his work the translation is defined by a dictionary mapping from node type to rule object creator, whereby rules essentially provides stateful expansion semantics. Note that the translation is no plain one-to-one mapping. Rules may generate semantic nodes again, eventually emitting the final rendering scene graph. Rule objects generated during traversal are stored in the so-called traversal cache of the rendering traversal.

In order to support other operations than rendering (e.g. `ComputeBoundingBox()`), semantic scene graph rule objects may implement additional interfaces. Each custom traversal is associated with such an interface. With this extension, the traversal algorithm either calls the interface method directly, which means that the semantic node provides a specific semantic function for that traversal or calls the generic expansion function defined within the rule object. For example in case of the `RenderingTraversal` the associated interface `Renderable` has one single method, called `Render`. Applied to our rendering traversal, nodes are expanded until a renderable node is reached.

To summarize the traversal algorithm can be described like:

1. When visiting node n, consult the traversal cache to find an appropriate rule instance r for node instance n. If no such rule exists, create a rule for n's type as specified by the semantic map.

2. Given a rule object r for node n and traversal t, check whether t's traversal interface is implemented by r. If so, call the interface method directly. Otherwise expand the node and go to (1).

Extensibility is solved by the fact, that semantic scene graph nodes are decoupled from concrete semantic functions. Whenever the rule defines no specific semantic function for a

given traversal, the traversal automatically continues in the generic traversal function. This way additional operations can be introduced by the implementation of a new traversal with some associated traversal interface. Rules affected from this traversal naturally need to implement that interface, other nodes remain unaffected.

With generics as provided by mainstream languages such as Java and C#, Tobler's concept can be implemented in a type safe manner, except for some semi-safe casts in the abstract traversal algorithm itself. However, the user-site API can be implemented in a type-safe manner. Details on the implementation can be found in Tobler's original paper [51].

To the authors knowledge Tobler's approach is the only one solving the expression problem in the context of scene graph systems.

3.3 Data Flow and Communication

Typically, scene graph systems make it easy to compute some value for a given scene graph. Usually this is achieved by traversal functions or traversal objects. Traversals simply collect data from leaf nodes and combine the resulting values into a single value, representing the outcome of the traversal for some given scene graph. Beginning from leaf nodes (e.g. geometry nodes), information basically flows upwards to the root.

Unfortunately, scene graph systems fail to provide a disciplined way for transferring values downwards, i.e. from the nodes to their children. Although traversal state objects, stored within the traversal object can be used to encapsulate state this approach has severe limitations:

- Traversal states break *extensibility*. Different kinds of traversals need different values, stored in the traversal state. In this case clients extending the scene graph system with additional traversals (operations) need to have access to the traversal state class definition, which in turn leads to recompilation, violating the separate compilation requirement of the expression problem. This limitation can be tackled by using an untyped dictionary as environment. This approach however violates type safety. Additionally traversal state objects tend to become a sink for many unrelated values, which leads to naming and organization issues. Note that Tobler's [51] approach in fact maintains extensibility for specific traversals, since each rule object provides traversal specific entry-points.

- Traversal states are problemantic in respect to encapsulation, since all attributes need to be visible for all traversals. Yet, it is possible to maintain encapsulation by using private getter functions — still, traversal states need to be handled of carefully.

- The only efficient possibility to transfer values between different traversals is via side-effects, i.e. one traversal updates a mutable cell, another one consumes the value of the cell.

- It is very hard to track dependencies in a system with unrestricted side-effecting access to traversal state.

- Executing traversals in parallel becomes tricky since traversal states cannot be implemented efficiently as persistent data structures[3].

To the authors knowledge no scene graph system provides a clean, comprehensible way for providing values to other nodes within the scene graph.

3.4 Performance

Tobler's *semantic scene graph* provides a clean separation of semantics and rendering. This abstraction however, comes with significant performance cost. As found by Wörister et al. [62] traversal overhead is relatively high in Tobler's system.

Constant node overhead aside, still, scene graph traversals in general are not efficient in interactive applications if most of the scene remains static between frames. Furthermore the structure of the input scene graph directly affects performance since different kinds of graphics API state changes have different cost. Surface switches for example are more expensive than switching vertex arrays for different draw calls. Strauss [45] gives an overview on common scene graph optimizations. All operations directly modify the scene graph in order to reduce scene graph size or modify structure for faster traversal or less graphics API interaction. All optimizations basically re-organize the input scene graph — which defeats the purpose of having a clean semantic scene graph.

Recently, Wörister [61, 62] introduced a caching system for scene graphs, capable of eliminating traversal overhead for semi-static scenes. Instead of repeatedly traversing static parts of the scene, rendering caches are placed within the scene graph. Each cache stores a pre-optimized array of draw commands representing the outcome of rendering traversals on the cached sub scene graph. In order to cope with dynamism like user input Wörister maintains a dependency graph which allows for fast in place updates of rendering caches. The algorithm however soley works for non-structural updates within the scene graph. Changes like additions and removals cannot be handled efficiently. In these cases rendering caches need to be rebuilt from scratch. As noted by Wörister et al. [62] proper support for structural updates requires a more disciplined approach for data flow and communication within nodes. Additionally, with static dependency graphs as used by Wöerister et al. it is hard to capture the dynamic semantics for scene graph traversals.

3.5 Summary of Problems and their Interaction

Traditional scene graph systems suffer from three fundamental problems:

No declarative model for communication Current scene graph systems ([8, 30, 35, 36, 51, 56, 59]) provide no declarative way for describing data dependencies between scene graph

[3]Here we mean *persistent* in the context of functional programming. Persistent data structures always preserve previous versions when modified. In purely functional programming, all data structures are persistent automatically.

nodes. Instead imperative traversals are used to query data, and state is used for propagating values downwards. For both mechanisms, **side-effects** are inevitable. Additionally arbitrary side-effects make reasoning, debugging and software maintenance difficult.

Expressive nodes Scene graph nodes may generate new nodes or modify other nodes (e.g. *semantic scene graph* [51]); other scene graph systems which allow for no dynamic content generation within the system use other mechanisms like manipulators to model dynamism [59]. Both variants essentially mutate scene graph nodes and properties in place. Again, **side-effects** appear in an uncontrolled manner. Additionally, scene graphs may become inactive due to modification. This complicates **memory management** due to dangling scene graphs. Firstly, such inactive parts need to be detected. Secondly, all associated rendering resources need to be freed appropriately.

Violation of Extensibility For programmers it shall be easy to add additional scene graph nodes and extend existing scene graph nodes with functionality. Tobler [51] takes an important step towards extensible scene graph systems. His approach however introduced additional overhead for scene graph traversal, which in turn exacerbates performance issues immanent to traditional scene graph rendering.

Due to expressiveness and the lack of declarative communication models side-effects become inevitable. This in turn makes optimizations, as introduced by Wörister [61, 62] impracticable — in order to support fully dynamic semantics in scene graph a more disciplined way for communication is indispensable. An overview of problems and interactions thereof is given in Figure 3.1.

3.6 Our Approach

Computing values for tree like data structures is a common exercise in many fields of computer science. In compiler construction for example most compiler transformations operate on abstract syntax trees. Since scene graphs and abstract syntax trees share a similar structure we looked at solutions in the languages community. Compilers can be structured into three phases:

1. Parsing
2. Semantic analysis
3. Optimizations and code-generation or interpretation

Scene graphs are typically generated by the application developer directly, or synthesized from other scene descriptions like VRML [9] or COLLADA [6]. This step of preprocessing can be seen as parsing, i.e. the construction of a tree representation. Subsequently, scene graph optimizations perform analysis (e.g. identifying identical geometry leafs) in order to possibly rewrite and optimize the scene graph. This phase is very similar to *semantic analysis* in compilers. In a final phase the scene graph is traversed in depth-first manner, eventually issuing draw calls for leaf nodes. Again, this is similar to interpreting abstract syntax trees, or generating faster code

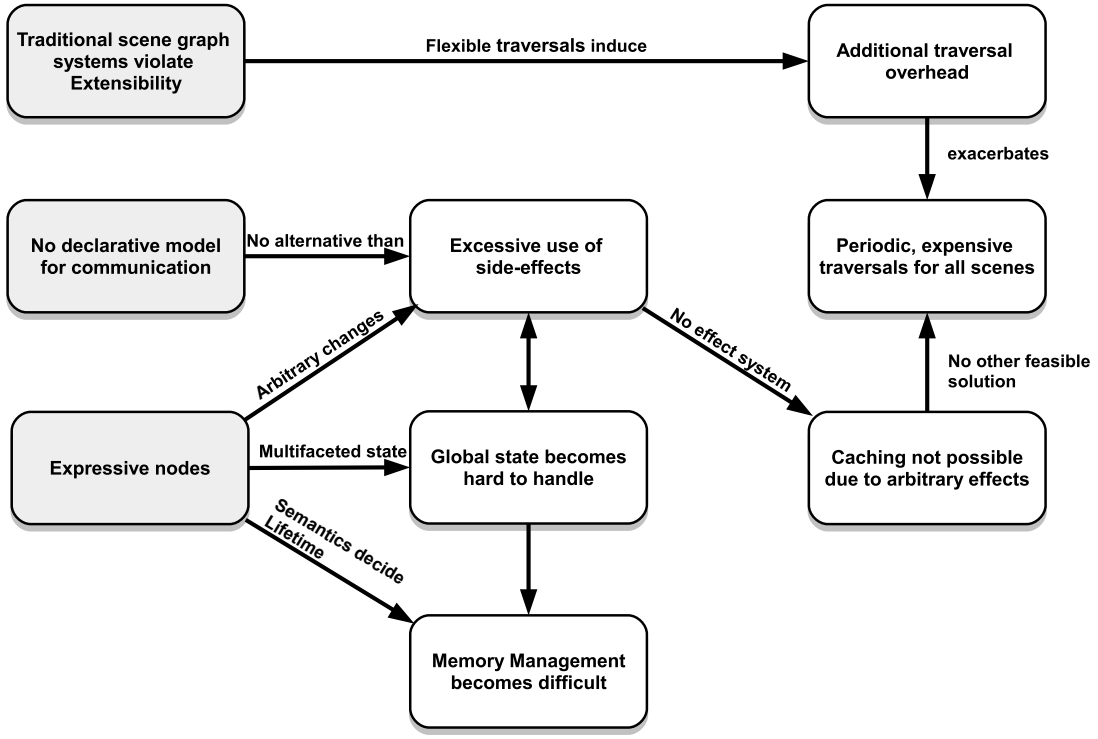


Figure 3.1: Shortcomings of current scene graph systems and interactions thereof. Traditional scene graph systems provide no **declarative mechanism for communication**, therefore **side-effects** remain the only viable option for implementing dynamism. Additionally scene graph nodes perform **arbitrary changes** within other parts of the scene graph. Performance optimizations like scene graph caching is not feasible in presence of arbitrary effects.

representing the semantics of interpreting the syntax tree.

Each of the three phases has been studied extensively. Context-free grammars are a common way for describing the syntactical structure of a programming language. Compiler generator tools like Yacc [24] can be used to generate parsers, given context-free grammars in EBNF form. Thus, context-free grammars can be seen as theoretical foundation for *parsing*. Unfortunately the formalism cannot be used to describe *semantic analysis*, which is inherently context sensitive. In his seminal paper Knuth [26] introduced the concept of *attribute grammars* which can be used to assign meaning to sentences in some input language, defined by an associated context-free grammar. Attribute grammars extend context-free grammars with semantic actions and form a solid foundation for *semantic analysis*. Attribute grammars provide two mechanisms for communication:

- Synthesized attributes
- Inherited attributes

Attributes are defined in a purely declarative manner using attribute equations (*semantic functions*). Therefore attribute grammars make data-flow and communication within hierarchical structures declarative, essentially solving the data-flow and communication problem: When writing operations there is no need to access global variables. Inherited attributes in fact replace the traversal state, while synthesized attributes replace data-gathering sub-traversals.

Let us first consider scene graphs without sharing, i.e. scene trees. Parse trees directly correspond to scene trees. Since scene graphs traditionally issue draw calls immediately while traversing, there is no obvious correspondence for traditional scene graph traversals since semantic functions must be side-effect free. Attribute grammars however can be used to model sequential composition. Instead of issuing draw calls as side-effect while traversing, we accumulate an explicit representation of the draw calls. In a final execution step this explicit representation can be executed safely, while maintaining pure semantic functions. At first glance this decision seems arbitrary, limiting and inefficient — in this thesis we will show exactly the opposite:

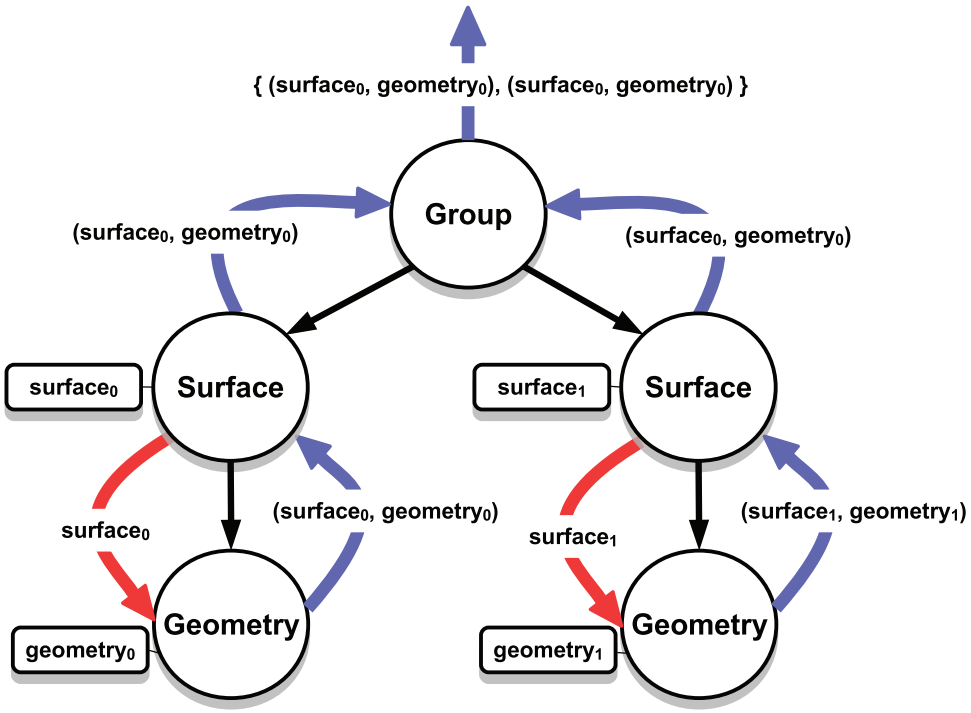
- Explicit representation of draw calls provides the opportunity for a rich set of optimizations.
- Draw calls and compositions thereof become first class values in the system, can be modified and passed around allowing for novel flexibility[4].

Thus, each scene node defines semantics for rendering which explicitly generates a sequence of draw calls that then can be executed by the rendering system. Figure 3.2 shows an attributed scene tree as well as its attribution rules (i.e. attribute grammar). Later in this thesis we show how to generalize attribute grammars to work with directed acyclic graphs instead of tree-like structures.

As noted in the previous section, scene graph systems traditionally traverse the complete scene over and over again, even if most parts remain static between successive frames. In order to provide interactive frame rates for big scenes we looked into execution models for attribute grammar systems. As noted in Related Work, attribute grammars are well studied and serve as a solid basis for optimizations and incremental evaluation. Early work on attribute grammars (e.g. Reps et al. [37], Hoover et al. [21]) use static dependency analysis in order to generate optimal execution plans. This works well if user modifications are explicitly visible to the evaluation system. In our system however, we don't want to impose any restrictions regarding modifications of the scene graph. We therefore use a fundamentally different approach: Motivated by recent advances in the field of incremental computation (Acar et al. [1, 2]) we use a general purpose incremental computation framework as basis for attribute grammar evaluation. This essentially gives us *incremental* attribute evaluation for free and has advantages over specialized attribute evaluation algorithms:

- Attribute evaluation incorporates well with other incremental data structures required for efficient rendering because general purpose incremental computation is composable [1].

[4] In fact this concept is very similar to monads in pure functional programming and its expressiveness has been studied extensively [58]. Another analogy is intermediate code, as known from compiler technology.



(a) Attributes and their data-flow. Blue arrows denote synthesized attributes, while red arrows denote inherited attributes. Surface values ($surface_0$,$surface_1$) as well as geometry values ($geometry_0$,$geometry_1$) are provided by respective scene graph nodes.

```
syn SEM RenderResult:
 | Group    this.RenderResult = { child0.RenderResult, child1.RenderResult }
 | Surface  this.RenderResult = child.RenderResult
 | Geometry this.RenderResult = ( this.surface, this.geometry )

inh SEM Surface:
 | Group    child0.surface = this.surface;
           child1.surface = this.surface;
 | Surface  child.surface = this.surface
```

(b) Semantic functions for the synthesized attribute *RenderResult* and the inherited attribute *Surface*. Geometry nodes have access to a private field *geometry* containing vertex data etc.

Figure 3.2: Attribution of a simple scene graph. Surface nodes publish provided surface values ($surface_0$,$surface_1$) downwards. Geometry nodes produce render results by returning a tuple consisting of its geometry and the inherited surface attribute. Render results propagates towards the root node, which eventually combines render results of its children.

- We can decide not to use adaptive evaluation for parts of the scenes which either never change or are cheaper re-evaluated from scratch than evaluated incrementally by the computation framework.
- Our algorithm decides explicitly how to resolve shared nodes in the input scene graph. We therefore expand the graph into tree form, while evaluating attributes.

Let us now restate how we tackle problems of current approaches.

Extensibility We use attribute grammars as the theoretical foundation for scene graph semantics [26]. From a design point of view this solves the expression problem.

Data-flow and communication With attribute grammars, hierarchical relationships and semantics are modeled in a uniform manner and provide well defined ways for communication between semantically related nodes. *Semantic functions* are pure by definition, which facilitates reasoning and optimization. Additionally data-flow remains free from side-effects which gives raise for incremental evaluation.

Performance We introduce an incremental attribute grammar evaluation scheme based on recent advances in incremental computation [3]. The evaluation integrates well with semantic functions, which are guaranteed to be pure and incremental. Unlike previous approaches [22, 37] our attribute evaluation is general purpose and naturally composes with other adaptive program parts.

3.7 Scope of this work

We formalize rendering by capturing draw calls in a hierarchical data structure which we call *RenderJobs*. In the scene graph system we use a synthesized attribute (*RenderJobs*) of type *RenderJobs*. Note that *RenderJobs* is not intrinsic to our system, but is merely an interpretation of rendering semantics defined for all scene graph nodes. In order to exploit temporal coherence between frames, we reuse *RenderJobs* from previous frames. This is accomplished by making all involved functions for computing *RenderJobs* incremental. Incremental evaluation of semantic function makes the *RenderJobs* attribute incremental as well and therefore responsive to arbitrary input changes. Given that *RenderJobs* are incremental itself, we synthesize the set of changes (Δ) between successive frames. We represent these changes as a set of additions and removals of render jobs. An overview of the approach is given in Fig. 3.3.

In this work we focus on modules of the frontend. Parallel to this work we developed a rendering backend, which efficiently maps *RenderJobs* to concrete draw calls, which can be executed by graphics API's like OpenGL and DirectX. However, efficient mapping of *RenderJob* deltas to API calls is out of the scope of this work. To summarize our system consists of:

An attribute grammar system including a domain specific language for convenient authoring of semantic functions. Attribute system evaluation is well understood and many efficient evaluation algorithm have been proposed. In our system however, we assume attribute

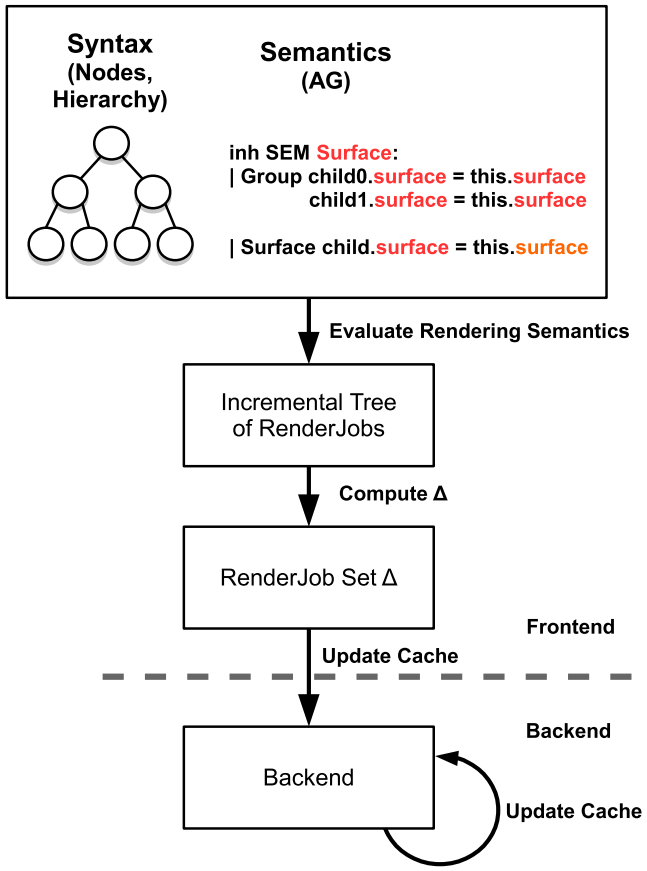


Figure 3.3: Modules of the proposed system. Node types define their hierarchical relationship. Adaptive semantics operating on nodes define attributes, including semantics for rendering. Evaluation of the RenderJobs attribute generates an incremental tree containing all RenderJobs represented by the original scene graph. Eventually, we compute the change-set (Δ) of RenderJobs, which is finally transferred into the rendering backend.

evaluations to be rare because most scenes remain static most of the time and incremental evaluation reduces attribute evaluation to a minimum. We therefore keep attribute grammar evaluation simple and focus on extensibility and design.

A scene graph framework implemented on top of our attribute grammar system.

A incremental computation framework capable of expressing dynamism of scene graph systems. Although general purpose incremental computation is a relatively new research field, algorithms for incremental evaluation are well understood.

In this thesis we present a prototypical implementation of the system. For evaluation we compare our system with state-of-the-art scene graph systems in terms of design and perfor-

mance.

CHAPTER 4

Methodology

In order to take advantage of specialized graphics hardware, scene graph systems need to provide fast traversals while maintaining high-level semantics and dynamism.

Naively, scene graph systems traverse the scene over and over again, although most scene parts remain static. As mentioned in Related Work, scene graph systems often use caching in order to speed up scene graph rendering (e.g. Durbin et al. [15], Wörister et al. [62]).

In this thesis we take a different approach. Instead of introducing rendering caches for specific parts of the scene, we use a more general approach by describing scene graph semantics fully incremental.

Our approach is influenced by *lazy incremental scene graph caching*, but our methods differ significantly. In this section we revisit the computation model as used by Wörister et al. [62].

Subsequently we give a short overview on *adaptive functional programming* which serves as basis for incremental evaluation of scene graph semantics. Moreover, the concept of *attribute grammars* plays a central role in our approach. In the last section of this chapter we show how to solve common programming tasks using attribute grammars.

4.1 Lazy Incremental Scene Graph Caching

Wörister et al. [62] utilize so called *rendering caches* which are placed in the scene graph manually. Rendering caches can be executed which has the same effect as traversing the sub graph represented by the cache (see Fig. 4.1). After modification, prior to rendering, the rendering system first makes rendering caches consistent[1] and executes the rendering cache instead of traversing the scene graph it stands for.

Crucially for performance, rendering caches need to be updated incrementally whenever attributes within the cached scene graph change.

Wörister et al. distinguish between two types of changes:

[1] in this thesis we use the term *change propagation* for making caches or dependencies consistent

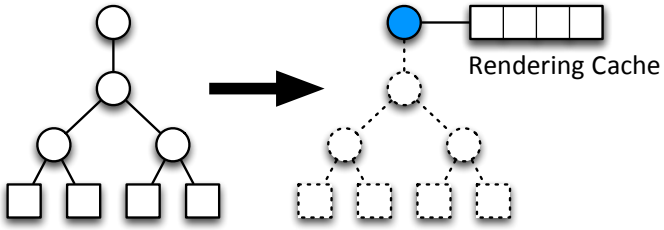


Figure 4.1: A rendering cache captures all rendering instructions in a sub graph and takes over its rendering responsibilities (taken from Wörister et al. [62]).

In-place updates This type of change describes value changes, which have no effect on the structure of the scene graph. Change propagation updates all necessary values like *uniform buffers*, *textures* in place i.e. no resource allocation needs to occur (see example in Figure 4.2).

Structural updates This type of change describes changes of the scene graph itself. Thus, *addition*, *removal* or *reorganization* of scene graphs is considered to be a structural update. Update propagation involves creation and grafting of computation graphs.

The system proposed by Wörister et al. supports incremental updates for *in-place updates*. Their system however cannot handle *structural changes* in an incremental manner i.e. rendering caches need to be invalidated and rebuilt from scratch in order to make the system consistent again.

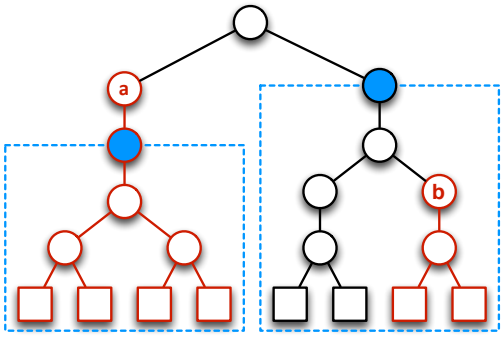


Figure 4.2: Modifying some attributes (*a* and *b*) requires the update of rendering caches (blue nodes) that capture on them (affected scene graph parts are shown in red). Taken from Wörister et al. [62].

In order to track changes within the system, Wörister et al. use a dependency graph. In order to model dynamism efficiently it is necessary to provide a flexible way for describing dependencies. Therefore Wörister et al. distinguish between:

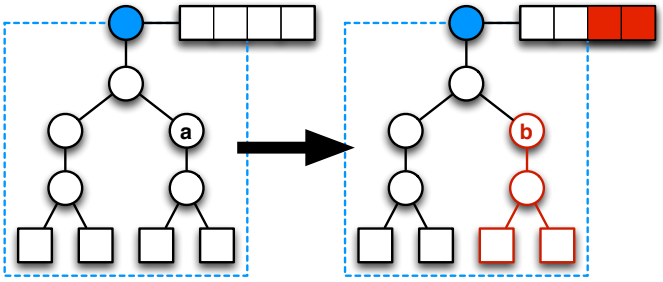


Figure 4.3: An in-place update of a rendering cache due to a modification of an attribute (affected scene graph parts and rendering cache parts shown in red). Taken from Wörister et al. [62].

Dependencies. A dependency is a conservative predicate which describes if some value has changed. As an example consider computations depending on camera position. The predicate for example checks whether the camera has moved more than n units.

Value Sources. A value source is an item in the scene graph or global environment. User inputs are typically value sources.

Dependent Resources. A dependent resource is an object whose content may change whenever one of its dependencies changes. Whenever a dependency of the dependent resource evaluates to true, the object needs to be updated explicitly.

Nodes in the dependency graph represent intermediate computations as well as value sources (inputs). Directed edges represent abstract dependencies operating on inputs of nodes. Given some input scene graph, their system constructs the *implied dependency graph* (see Fig. 4.4 and Fig. 4.5).

Leaf nodes in the dependency graph i.e. *dependent resources* correspond to hardware resources like *uniform buffers*. Since the structure of rendering caches remains stable, a rich set of optimizations (e.g. *redundancy removal* [62]) can be applied to rendering caches.

4.1.1 Update propagation

If value sources change and their output dependencies evaluate to true, all transitively reachable dependent resources need to be updated appropriately.

To this end Wörister et al. use a variant of Hudson's [22] algorithm for evaluating attributes in attribution graphs. Hudson's algorithm works in two phases:

1. Given some input change v, mark all nodes out of date depending on v.

2. Whenever a value is demanded, check whether it is marked out of date. If so, recursively update all of its inputs, and recompute the value.

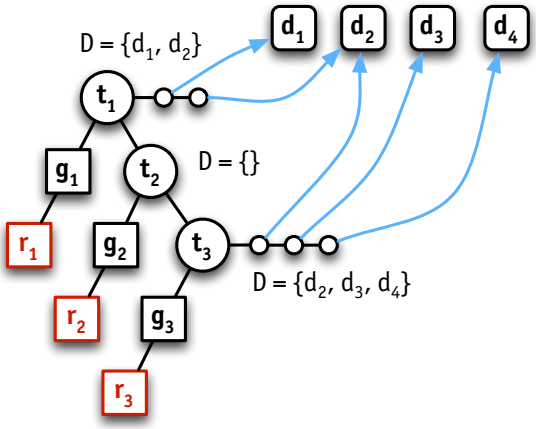


Figure 4.4: A small scene graph with transformation nodes as value sources (t_i), a number of dependencies for these value sources (d_j), a number of geometries (g_k), and a number of dependent resources (r_l). Taken from Wörister et al. [62].

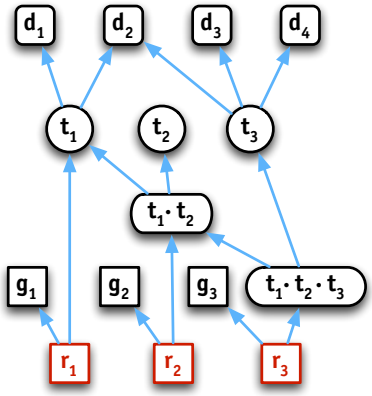


Figure 4.5: The dependency graph constructed from the scene graph in Figure 4.4. Taken from Wörister et al. [62].

Wörister et al. use a flattened version of the dependency graph. Each dependent resources is associated with its transitively reachable input dependencies. If a dependent resource is demanded by the rendering backend, it first checks all its dependencies and makes them consistent.

Rendering the contents of a rendering cache involves two steps:

1. Check all dependencies for consistency.

2. Update dependent resources.

3. Execute the rendering commands stored in the rendering cache.

This evaluation scheme[2] might appear inefficient at first glance, but since dependent sets are small for most draw commands, the algorithm turns out to be very efficient in the setting of scene graph rendering.

4.2 Limitations of Lazy Incremental Computation

Lazy incremental computation for scene graphs provides high-performance for scenes with value changes. Rebuilding caches as necessary in case of structural changes however is expensive and does not employ benefits of incremental evaluation. In Hudson's two phase approach *evaluation* cannot be safely interleaved with *marking*. As a consequence, structural modification need to be done right before marking. Technically this is no problem, but requires the programmer to be explicit about modifications. Furthermore, the two-phase approach is impractical if the structure of the computation is dynamic, i.e. depends on the evaluation of other computations. In our attribute grammar based approach however, we strive for arbitrary dynamic semantics which renders the approach taken by Wörister et al. infeasible.

4.3 Adaptive Functional Programming

In their seminal paper *adaptive functional programming*, Acar et al. introduce *dynamic dependency graphs*. Unlike previous approaches (e.g. Hudson [22]) the dependency graph is constructed implicitly by running an instrumented version of the computation. As demonstrated by Acar et al. the framework can be implemented purely as a library, i.e. no compiler support is necessary. In this section we will give a short overview on the concepts of adaptive functional programming.

The concept is based on the idea of *modifiable references* (*modifiables* for short). Modifiables support three basic operations:

mod constructs a new modifiable with some initial content.

read reads the current content of a modifiable, and updates the dependency graph to reflect that dependency.

write writes a value into a modifiable cell.

All operations implicitly maintain the underlying dependency graph. An expression depending on changeable values (contained by modifiables) needs to read its inputs explicitly using the `read` primitive. Consequently, such expressions are also changeable and need to be represented by modifiables. Expressions that are not changeable are said to be *stable*.

Adaptive functional programs produce a result of type `t mod`[3], which is constructed by possibly reading other changeable values and writing the result to the destination cell. The `SML`

[2] Wörister et al. use the term *lazy polling*

[3] Since code examples in this section are given in *Standard ML (SML)* we use SML notation for polymorphic types. In Java and C# $t\ mod$ would be written like `mod<T>`

```
signature ADAPTIVE = sig
  type 'a mod
  type 'a dest
  type changeable
  val mod: ('a * 'a -> bool) -> ('a dest -> changeable) -> 'a mod
  val read: 'a mod * ('a -> changeable) -> changeable
  val write: 'a dest * 'a -> changeable

  (* Meta operations *)
  val change: 'a mod * 'a -> unit
  val propagate: unit -> unit
  val init: unit -> unit
  val extract : 'a mod -> 'a
end
```

Listing 1: Signature for the SML adaptive library as introduced by Acar et al. [2] (Code is given in SML)

signature for the adaptive functional programming library is given in Listing. 1. `mod` takes a conservative comparison function, and a continuation function which writes a value to the destination cell, passed as argument. `read` takes a modifiable and a continuation function which is supplied with the actual value of the modifiable. When reading from a modifiable (using `read`), the library introduces an edge to the dependency graph, which is called *reader*. Whenever the source modifiable changes, all associated readers which correspond to edges in the dependency graph need to be re-executed.

Adaptive functional programs look similar to ordinary functional programs except for their types and some wrapper functions. Changeable expressions operate on $t\ mod$, while stable expressions operate on ordinary types. Additionally, the original library implementation dictates the programmer to write programs in destination passing style. The meta operation `propagate` makes all output consistent again after modifications, while `extract` makes the changed outputs observable for the program. When used in adaptive computations, the `extract` operations is unsafe, because it does not maintain the dependency graph properly. Thus, `extract` should be used soley for extracting modifiable values after making them consistent using `propagate`. Note that in our implementation we use the name `unsafeRead` instead of `extract`, in order to stress the fact, that it shall not be used within adaptive computations.

Listing 2 shows a simple adaptive program. The program first creates two modifiable cells containing integer values. Another modifiable first reads both values and writes the sum of both as a result to the destination cell. The system generates the dependency graph on-the-fly, thus in line (16), we can extract the value 3 and print it to screen. After changing m_1 to 100 and a call to `propagate`, the system automatically updates m with its new value 102.

The dependency graph of Listing 2 is given in Fig. 4.6. The system maintains containment relationship for all nested reader functions. The reader function r_9 introduced in line (9) is contained in r_8 (line (8)). If m_2 changes, all readers depending on the value of m_2, need to be re-executed. In this case r_9 is re-executed, thus the line (10) is re-executed as well. Finally,

```
1  init () (* initialize library *)
2
3  (* create changeable expressions, write constants to modifieables *)
4  let val m1 = mod cmp ( fun d => write(d,1) )
5  let val m2 = mod cmp ( fun d => write(d,2) )
6
7  m = mod cmp (fn d =>
8            read(m1, fn v1 => (* read m1 *)
9            read(m2, fn v2 => (* read m2 *)
10                 let val r = v1 + v2
11                 in write(d,r) (* write result to new destination of m *)
12                 end)
13              end)
14       )
15
16 printfn "%d" (extract m) (* prints 3 *)
17 change (m1,100)          (* change value of m1, this makes reads in line
18                            8 and 9 inconsistent. *)
19 propagate ()             (* reexecutes inconsistent reads,
20                            computes new v1 + v2 and updates m's value *)
21 printfn "%d" (extract m) (* prints 102 *)
```

Listing 2: Adaptive program adding two integers, contained by modifieables. (Code given in SML)

line (11) is re-executed and updates the value of m. In case of modification of m_1 all readers appearing after t_1 need to be re-executed. Since r_9 is contained in r_8 it is executed as well. The change propagation, as well as the complete *ML* library is given in Acar et al. [2]. Note that the approach works for arbitrary functional programs including recursion. Implementation details as well as optimality arguments is out of the scope of this theses and can be found elsewhere [1,2].

In contrast to Hudson's approach, adaptive functional programming has no separate marking phase. Additionally, change propagation is strict, i.e. change propagation updates all outputs in an eager fashion.

4.4 Problem solving with Attribute Grammars

In this section we will give a short overview on how to solve common programming tasks declaratively by utilizing attribute grammars. The aim of the section is to make the reader familiar with attribute grammars, semantics and associated vocabulary. Therefore, we use an illustrative example, which operates on a simple recursive data structure[4].

Our example and its implementation is heavily influenced by work of Swierstra [48]. Given is a singly linked list. The task is simply to compute a new list with all elements replaced by the

[4] structurally similar to scene graphs

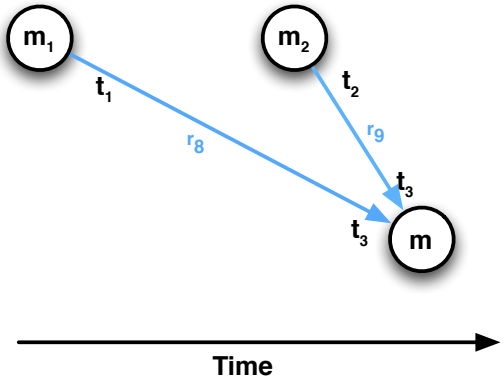


Figure 4.6: Dependency graph corresponding to the example program, given in Listing 2. Circles denote modifiable cells, blue directed edges denote readers attached to modifiables. In this example, reader r_9 is contained in reader r_8.

average of the input list. For reference, a straightforward functional implementation in the F# programming language is given in Listing. 3.

```
1  (* Sum type for describing linked lists. Each element
2  is of type float *)
3  type MyList = MyCons of float * MyList
4              | MyNil
5
6  let rec computeSum xs = match xs with
7       | MyCons(x,xs) -> x + computeSum xs
8       | MyNil -> 0
9  let rec computeLength xs = match xs with
10      | MyCons(x,xs) -> 1 + computeLength xs
11      | MyNil -> 0
12 let replace xs =
13    let avg = computeSum xs / computeLength xs
14    let repl avg xs = match xs with
15                        | MyCons(v,xs) -> MyCons(avg,repl xs)
16                        | MyNil -> MyNil
17    repl avg xs
```

Listing 3: Recursive solution to repl-avg. (Code given in F#)

As expected the solution is simple but not really extensible. Adding data variants requires all traversal functions to be adapted. As noted earlier, no standard programming technique provides a solution to the expression problem (also see *The Expression Problem*). Furthermore, the implementation is not efficient, since sum and length of the list can easily be computed in a single pass.

Let us now solve the problem using the *The Utrecht University Attribute Grammar Compiler (UUAGC)* [47].

First let us define the syntax, which is identical to a sum type declaration[5]:

```
data List
   | Cons  head :: Float
           tail :: List
   | Nil
```

Let us now implement the attribute grammar counterpart of standard traversals i.e. *synthesized* attributes and their semantics.

```
attr List           -- for syntax List, define attributes:
   syn sum :: Float -- a synthesized attribute of type Float
   syn len :: Int   -- a synthesized attribute of type Int

sem List
  | Cons  lhs.sum  = @head + @tail.sum
          lhs.len  = 1 + @tail.len
  | Nil   lhs.sum  = 0
          lhs.len  = 0
```

For each alternative (each derivable child production), we denote attribute equations (semantic functions). The keyword `lhs` denotes the left-hand side of the production i.e. the current non-terminal. For *synthesized* attributes the left-hand side contains the attribute we want do compute (`sum` and `len`). The semantic function is arbitrary Haskell code, whereby identifiers starting with `@` are replaced by their semantics. `@tail.sum` for example, is conceptually replaced with code computing the synthesized attribute *sum* for the *tail* list. Unqualified names refer to field variables of syntactic nodes.

Next, we compute a new list by replacing all values of the initial list with some value, defined by `val`. Note that `val` is a free (inherited) variable when computing the semantics, i.e. it is not bound to an expression yet.

```
attr List
  syn repl :: List
  inh val  :: Float

sem List
  | Cons lhs.repl = Cons @lhs.val @tail.repl
         tail.val = @lhs.val
  | Nil  lhs.repl = Nil

```

For the `Nil` case we simply return an empty list again. For `Cons`, we propagate the replacement value downwards to its tail. The synthesized attribute `repl` for `Cons` cells can be constructed by constructing a new list using the replacement value and the replaced list of the tail. We can now use the attribute grammar compiler to generate a composed semantic function for our grammar. The function has type `List -> Float -> (Int, Float, List)` which

[5]Note that the example is given as working UUAGC code, which is similar to Haskell code

means: provided with some syntax node of type `List` and a replacement value of type `Float` the function computes a tuple consisting of list length, list sum and a replaced list, which are all synthesized attributes defined for lists.

It remains to combine the attributes of computing replacement values and performing the actual substitution. Of course we could manually apply the semantic functions defined on lists, but here we use the attribute system for composing attributes which enables efficient evaluation and composability. Therefore we define an artificial syntax node which essentially performs the required attribute plumbing:

```
1  data Root
2    | Root list :: List
3
4  attr Root
5    syn repl :: List
6
7  sem Root
8    | Root lhs.repl = @list.repl
9           list.val = @list.sum / (fromIntegral @list.len)
```

`Root` now wraps a list, computes its `average` value by computing `length` and `sum` and assigns the replaced list to the *synthesized* attribute `repl`. As expected, the final result of compiling the attribute grammar is a function of type `Root -> List` which elegantly solves the *rep-avg* problem.

Indeed the UUAGC system generates an efficient evaluation function by combining multiple traversals (*length* and *sum*) into a single one.

The attribute grammar implementation has several advantages compared to the naive F# implementation:

- The solution is composable respective to different aspects of tasks. Aspects can be freely composed by using attributes.
- The implementation is purely declarative – independent attributes can be evaluated in parallel, multiple traversals can be folded into single traversals.
- Theoretically, attribute grammars can be checked for cycles, i.e. if they are well-formed [25]
- The concept allows for automatic incremental evaluation.

In the next chapter we show how to use the concept of attribute grammars for authoring scene graph systems leveraging all previously mentioned advantages, ultimately solving problems synthesized in Summary of Problems and their Interaction.

CHAPTER 5

Incremental Scene Graph Semantics

At the beginning of this chapter we describe design decisions taken in this thesis and present important modules of our concrete implementation.

Next, we discuss the implementation of simple scene graph nodes and associated operations. We show the benefits of our declarative implementation and demonstrate how our approach solves common problems of scene graph systems.

5.1 Design Goals

Extensibility Our approach should provide extensibility for both, nodes and operations (see Chapter The Expression Problem).

Familiarity Object-oriented concepts and patterns should be reusable in our system. Another goal is to use development tools programmers are already familiar with. Instead of introducing special editors or languages, we aim to reuse existing infrastructure. Additionally, we want to maintain as much static type information as possible in order to support existing development tools such as auto-completion or refactoring.

Compatibility In order to prevent code-duplication, existing scene graph nodes should integrate with our new system.

Expressiveness Semantics must be expressive. The system should support completely dynamic scene graphs i.e. children may be computed in semantic functions. As an example consider a *switch* node, which returns different sub scene graphs depending on some other changeable property.

Aspect-oriented design When extending the system with a new feature, i.e. new nodes and attributes, type definitions and functions associated with this new feature should be defined as coherent package (as an *aspect*). This allows features to be bundled logically. As an example consider level of detail, implemented as extension to an existing scene graph

system. For the application programmer, it shall be possible to implement the functionality in a new module, without modifying other existing modules. This allows for development of single features without interferring with other features.

Memory management When working with scene graphs, the computation framework must take care of proper memory management. If dynamic semantics or user modifications change the structure of a scene graph, then all values associated with the old structures should be reclaimed automatically.

Efficiency One dedicated goal of our system is to maintain incremental performance for all modules involved. This goal also involves backend integration. Ultimately, rendering semantics need to be mapped to actual graphics hardware. Thus, we require the system to provide integration techniques for incrementally mapping changes in scene graphs to streams of draw calls.

5.2 Key Design Decisions

5.2.1 Embedded DSL for attribute grammars

Domain specific languages can be separated into two categories:

External Domain Specific Languages the language is implemented with its own parser, analyzer etc. Programs are usually compiled into another programming language or used directly to model semantics.

Embedded Domain Specific Languages (EDSL) The language is implemented within some host language. In its extreme, APIs can be seen as simple internal domain specific languages. Parsing as well as the type system is also provided by the host language.

Most attribute grammar systems use an external domain specific language for describing syntax and semantics.

In this work we use a *internal domain specific language* as implementation strategy in order to cope with the requirement of seamless *integration* as well as *separate compilation*. Also, this decision enables application developers to work in their familiar programming environment.

5.2.2 A mixed computation model for incremental evaluation

Adaptive functional programming introduced by Acar et al. [2] provides a powerful framework for adaptive computations.

The system is general purpose and fits our needs regarding expressiveness and dynamism. The framework allows for incremental evaluation of scene graph semantics and integrates with other adaptive algorithms since adaptive programs are composable. Memory management of dynamic dependency graphs is clearly defined: obsolete parts of the dependency graphs are reclaimed automatically by change propagation and garbage collection.

However, note that *change propagation* is eager, i.e. change propagation makes all modifiables consistent, even if the output is not actually required for rendering (e.g. because of view frustum culling).

In the field of rendering, Wörister et al. [62] use a specialized computation model for evaluating computations in an on-demand manner. As we found in early experiments, adaptive computation turns out to be quite expensive for simple value changes, especially if not all values need to be computed. Similar to Wörister et al., we distinguish between *structural changes* and *value changes*. In order to get best of both worlds — lazy evaluation for value changes and eager evaluation for structural changes, we use both computation models for describing dynamic semantics. Application programmers are free to decide which framework to use. Simple computations like dynamic transformations should be modeled with lazy computation, while modifiable scene graph nodes should use adaptive functional programming.

5.2.3 Monads for Incremental Computation

Acar's adaptive library for adaptive computations is based on destination passing style. The library requires to write destination cells of changeable expressions exactly once. Acar uses a modal type system to enforce this restriction. Carlsson [10] introduced a monadic interface on top of Acar's library, which also enforces this restriction without the need to extend the type system. Additionally, the monadic API takes away the burden of passing around destination modifiables. In order not to alienate programmers not used to continuation pass style programming we adapt Carlsson's approach for F#. Carlsson's approach is implemented in Haskell, which has syntactic support for monadic computation exposed via *do-notation* [33]. F# provides a similar concept, called *computation expressions* [32, 50].

5.3 An EDSL for Attribute Grammars

F#, a mixed paradigm programming language, fully integrated in the .NET development environment provides a variety of facilities for embedded languages. The language comes with support for code quotations [49]. Similarly to *LISP*, code can be interpreted as data which makes code quotation a viable option for compiling F# expressions to other target languages. At first glance the feature seems tailor-made for implementing attribute grammar descriptions.

The approach however turns out to be cumbersome since generated code is hard to understand and not well suited for debugging.

In this thesis we take a pragmatic approach. We simply use standard object-oriented classes, annotated with some special *semantics* attributes.

At load time we inspect all methods and its parameters and store their entry points in a global dictionary mapping from semantic name to entry point. Given a mechanism for defining entry points for semantic functions, accessing attributes is still unsolved. With semantic functions being ordinary methods, their body cannot be analyzed. We therefore introduce special operators for accessing attributes at run-time. F# has support for dynamic typing via dynamic operator expressions [50, §6.4.4]. The binary dynamic operator `?` of type `obj -> string ->`

'a is statically valid on any object. As noted in the specification, the operator is useful for implementing custom property lookups like Expando objects [28] in C#. As the type suggests, its return type can be of any type 'a. Thus, the type unifies with any type, i.e. static type safety is violated. Interestingly, this feature integrates fine with type inference of F#. Since the context of the attribute dictates types of the attribute and the type is inferred statically, we accept the loss of static type safety. Theoretically, our system can be used from other languages such as C#. Yet, our design significantly benefits from F# features. Thus, in the rest of this chapter we will stick to F# code.

In our approach we use ? for looking up attributes and the ternary <-? operator for assigning inherited attributes. Our top level attribute operators look like:

```
1  // given a syntactic entity of type obj, lookup the
2  // name attribute, and coerce the value to 'R.
3  let (?) (o:obj) (name:string) : 'R =
4          getAttribute o name typeof<'R> |> unbox<'R>
5
6  // Given a syntactic entitry of type obj, set its
7  // attribute name to some value of type 'a.
8  let (?<-) (syntax : obj) (name : string) (value : 'a) =
9          setAttribute syntax name value
```

Note that our implementation technique also works for languages without dynamic operator overloads by simply providing functions for looking up attribute names for arbitrary objects. For interoperability, we use this approach when accessing attributes from C# code.

In our F# implementation we distinguish syntactically between *inherited* and *synthesized* attributes.

Assigning inherited attributes Inherited attributes can be assigned by using the ternary (?<-) operator:
e.g. child?VertexAttributes <- va assigns va to the inherited attribute (VertexAttributes), given in the child scene graph (child).

Accessing attributes We provide two syntactic forms for accessing attributes. Both operate on the binary (?) operator. In order to distinguish between accessing inherited and synthesized attributes we use different types:

1. In order to stress the operative intention of synthesized attributes, which is to compute an attribute with some execution context, synthesized attributes are considered to be functions of type: unit -> 'a. Thus, when accessing **synthesized** attributes we apply unit explicitly, e.g. va.Child?RenderJobs(), computes the synthesized attribute RenderJobs for the node va.Child.

2. For **inherited** attributes we simply use a the attribute as value,
e.g. let va = renderNode?VertexAttributes
binds the inherited attribute VertexAttributes for the syntactic entity renderNode to va.

5.4 Attribute Evaluation

In contrast to attribute grammar systems with analysis and code generation (e.g. Reps et al. [37]), our approach does not generate explicit static evaluation schedules, but evaluates attributes at run-time in an on-demand manner. This approach is particularly well suited because:

1. Attributes are only evaluated if absolutely needed.

2. The approach integrates fine with the EDSL, which is dynamic and requires no explicit analysis.

Our evaluation scheme basically simulates lazy evaluation, thus is comparable to traditional implementation techniques as used in UUAGC [47] and Slone et al. [42].

At top level, only synthesized attributes can be demanded (e.g. *RenderJobs*), so our evaluation always starts with evaluating a synthesized attribute a for a given syntax node s. We use an auxiliary stack for maintaining the context while evaluating attributes (`Ctx`). In order to prevent reevaluation of semantic functions for the same context we use a map from `Ctx * Semantic` to the result of the semantic function (*attribute cache*). Additionally we store semantics in a semantic map with the signature:

```
1 type SemanticMap =
2         abstract member GetSemanticFunction : Type * Identifier -> EntryPoint
```

Recall the definition of semantic functions:

```
1 member x.BoundingBox(n : RenderNode) : Mod<Box3d> = f
```

In this example, the semantic map contains an entry:
`typeof<RenderNode> * 'BoundingBox' -> EntryPoint(f)`.

Evaluation of synthesized attributes simply executes the appropriate semantic function. The dynamic operators take care of attribute evaluaton, which occurs at run-time in an on-demand manner. The algorithm for evaluating synthesized attribute a for the node s proceeds as follows:

- Safe the current context.
- Push s onto the node stack.
- Lookup (ctx, a) in the *attribute cache*. There are two cases:
  - (ctx, a) exists, thus restore the context and simple return its cached value.
  - Lookup (s, a) in the *semantic map*.
    - No entry point for given (s, a): evaluation fails immediately.
    - Invoke entry point with argument s and store its result in the *attribute cache*, restore the context and return the result.

Whenever an inherited attribute is accessed we recursively search for the attribute value. This corresponds to searching for the requested attribute in all parent nodes. Semantic functions for inherited attribute have return type `unit`, thus the semantics depends on a side-effect.

The dynamic assignment operator `s?a<-v` writes v with key (s, a) to the *attribute cache*. Crucially, evaluation of inherited attributes, works in an on-demand manner. If evaluation of another attribute encounters the need for a inherited attribute a in node s, the algorithm proceeds as follows:

1. Safe the current context.
2. Push s onto the node stack.
3. Lookup (ctx, a) in the *attribute cache*. There are two cases:
   - There exists a cache for attribute a. Restore the context and return the cache value.
   - No cache value exists. Lookup (s, a) in the *semantic map*:
     - If the attribute stack is empty, fail immediately. Otherwise, pop the syntax node from the attribute stack and proceed in (2).
     - Invoke entry point with argument s. Semantic functions for inherited attributes have return type `unit`. If the semantic function was well-formed, now there exists an entry in the *Attribute*. Lookup (ctx, a) in the *attribute cache*:
       * No entry found. Semantic function was not well-formed. Issue error message and fail.
       * Restore the context and return cache value.

5.4.1 Capturing context for reevaluation

Since *semantic functions* are typically adaptive, attribute evaluation needs to restore the execution context of the initial execution. Thus, we provide a convenience function (`Captured`) for pairing nodes with their initial exeuction context. The implementation simply attaches the execution context to the current object. Whenver the captured syntax node is accessed, the original execution context is restored, which gurantees attribute getters to be safe while change propagation.

5.5 Adaptive Functional Programming

Our implementation of adaptive functional programming is based on the *ML* library introduced by Acar et al. [2].

In order to provide the user with a simpler API, we use a variant of Carlsson's monad for incremental computation [10]. Carlsson uses an adapted continuation monad for representing adaptive computations. Our implementation languages, C#, and F# do not provide proper support for monads. F# however, provides *computation expressions* which allow for overloading of most F# language constructs. Computation expressions are syntactic sugar for calling methods of a dedicated class instance (a *computation-expression builder*).

At this point we shall note one difference regarding the user API. Carlsson's monad is basically a continuation monad, which makes it necessary to lift modifiable reads into the monad by applying `readMod` to modifiables.

```
do  m <- newMod (return 1)
    mplus1 <- newMod (do v <- readMod m)
                         return (v+1))
    r <- readMod mplus1
    return r
```

By contrast, our implementation does not require readMod. We achieve this by statically overloading `let!`. One overload corresponds to continuation bind, exactly like Carlsson's monad bind. The other one has an argument of type `Mod<T>`, thus *readMod* is applied implicitly:

```
adaptive
 {
    let! m = initMod 1
    let! mplus1 =  adaptive
                   {
                       let! v = m
                       return v + 1
                   }
    return mplus1
 }
```

Note that this is not directly possible in Haskell, since do-notation operates on generic monad operations only. The implementation of our *computation-expression builder* is given in Listing. 4, which defines the syntactic translation to Acar's adaptive API. By applying the semantics[1], the (simplified) desugured version of our example essentially looks like:

```
adaptive.Bind(initMod 1, fun m ->
  adaptive.Bind(  adaptive.Bind(m, fun v ->
                                   adaptive.Return(v+1)),
                  fun mplus1 ->
                    adaptive.Return(mPlus1)
                )
```

```
type Adaptive<'a> = ('a -> unit) -> unit

type Adapt() =
    member this.Bind (m, f : 'a -> Adaptive<'b>) : Adaptive<'b> =
        fun c -> m (fun a -> (f a) c)
    member this.Bind (m : Mod<'a>, f : 'a -> Adaptive<'b>) : Adaptive<'b> =
        this.Bind(readMod m, f)
    member this.Return (x : 'a) : Adaptive<'a> =
        fun k -> k x
    member this.ReturnFrom (m : Adaptive<'a>) : Adaptive<'a> = m

    member this.Run(m : Adaptive<'a>) = newMod m  // correponds to Acars mkMod
```

Listing 4: A computation expression builder for adaptive computations. Its implementation is similar to the continuation monad used by Carlsson [10]. Our computation expression builder provides two implementations of the *bind* operator. One implementation corresponds the *bind* operation of the continuation monad. The other one creates a continuation by implicitly reading the content of a modifiable. Overload resolution resolves the appropriate bind operation statically.

[1] Precise translation rules for computation expressions is given in [32]

5.6 A mixed Computation Model

In order to distinguish between structural changes and value updates, we use a second type for modeling incremental evaluation, which we call `Computation<T>` (in contrast to `Mod<T>`). For computations we build dependency graphs in an explicit manner. Each computation gives rise for a new node in the dependency graph. At construction time of compound computations (computation with computations as input), we connect the entities in the dependency graph. For evaluation and update propagation we use Hudons algorithm [22].

5.7 Representing Draw-calls purely functional

Instead of issuing draw calls via side-effects while traversing the scene graph, we use the *synthesized* attribute *RenderJobs* for modeling rendering semantics. Let us first introduce a description of a single draw call, which we call *RenderJob*.

```
1  type RenderJob = Empty
2                 | RenderJob of isActive : Computation<bool> *
3                                surface  : Computation<ISurface> *
4                                rasterizerState : Computation<RasterizerState> *
5                                drawCallInfo    : Computation<DrawCallInfo> *
6                                (* ... uniform attributes etc. *)
```

RenderJobs are either empty or a complete description of the draw call including all parameters for the draw call and *uniform parameters* like surfaces, spatial transformations etc. For convenience, each render job is equipped with a boolean flag (`isActive`), which allows to disable specific render jobs, without changing the scene graph structurally. If appropriate, parameters are changeable i.e. computations or modifiables. For example, the field *drawCallInfo* is a computation, which makes in-place updates of rendering parameters possible.

Next, we need a data structure for describing multiple *RenderJobs*. Rendering is not commutative in general, i.e. semantics imposes an order, in which render jobs need to be executed. In this thesis however, we ignore order-dependent rendering and focus on unordered sets of render jobs[2].

In order to maintain the set of active render jobs efficiently, the structure needs to be incremental. Modifiable lists are not sufficient, since we need constant time concatenation of render job sets. Thus, we use a modifiable binary tree where leafs are RenderJobs.

```
1  type BTree<'a> = Node of Mod<BTree<'a>> * Mod<BTree<'a>>
2                 | Leaf of 'a
3
4  type RenderJobTree = Tree<RenderJob>
5  type RenderJobsMod = Mod<RenderJobTree>
6
7  let renderJobSet (sg : ISg) : RenderJobsMod =
8    // compute modifiable render job tree, given a scene graph
9    sg?RenderJobs()
```

[2]This limitation can be lifted easily by associating render jobs with keys, denoting their intended order of execution.

5.7.1 Composing render job sets

Consider a binary group, which composes two render job trees. The adaptive semantics for *RenderJobs* could be defined like:

```
type BinaryNode(l : Mod<ISg>, r : Mod<ISg>) =
    interface ISg

    member x.Left  = l
    member x.Right = r

[<Semantic>]
type RenderBinaryNode() =
    member x.RenderJobs(b : BinaryNode) =
        adaptive
         {
            let! l = b.Left
            let! r = b.Right
            let! leftJobs  = l?RenderJobs()
            let! rightJobs = r?RenderJobs()
            return BNode(leftJobs,rightJobs)
         }
```

Sequences of arbitrary length can be composed by folding concatenation over the sequence, starting with a leaf containing an empty render job. Traditionally scene graph systems support dynamic *Group* nodes with an imperative programming interface for adding and removing scene graphs. For dynamic group nodes we maintain a render job tree explicitly. In order to support addition, it is necessary to maintain an extension point within the binary tree. In our implementation we use a node, containing an empty render job. In case of additions we simply modify the node, insert another internal node containing the new node, as well as a dummy leaf node which serves as an extension point for future additions. For efficiency, we maintain an auxiliary data structure which maps scene graph nodes to tree nodes. When removing a scene graph node, we look-up its associated binary tree node and update it using the adaptive framework.

5.7.2 Adaptivity

In order to maintain incremental evaluation, the render job tree is stored in a modifiable (*RenderJobsMod*).

Thus, the render job tree modifiable remains stable at all times, while the rendering backend observes changes in the tree. Naively, the rendering backend traverses the complete render job tree each frame. However, this approach defeats the purpose of incremental scene graph semantics.

Next we show how to integrate a rendering backend which observes the render job tree while maintaining efficiency, i.e. incremental evaluation.

5.8 Rendering Engine Integration

Our rendering backend handles dynamic semantics by maintaining a set of active render jobs (*active*). In order to efficiently map dynamic semantics of scene graphs to render jobs, the

rendering backend keeps *active* consistent by incrementally applying changes to *active*. Changes are defined by the data type:

```
type Change = Removal  of RenderJob
            | Addition of RenderJob
```

Our backend provides functionality for applying a set of changes to its internal set of active render jobs:

```
type IBackend =
   (* Update rendering with a list of changes *)
   abstract member Update : List<Change> -> unit
```

Computing the set of changes incrementally is inherently a stateful operation, since Δ depends on the previous state of $active$. In this thesis we propose a customized algorithm for computing the change set incrementally.

5.8.1 Preliminaries

Recall, that our binary tree structure is defined inductively by a sum type[3]:

```
type Tree = Node of Mod<Tree> * Mod<Tree>
          | Leaf of RenderJob
```

Definition 1. *A binary tree T is full if each node is either a leaf or possesses exactly two child nodes.*

Lemma 1. *By construction, our binary tree is always a full binary tree.*

Theorem 1. *Let T be a nonempty, full binary tree then: If T has L leaves, the total number of nodes is N = 2L – 1.*

Definition 2. *Two RenderJob trees are equal, if they have the same reference, i.e. point to the same memory location.*

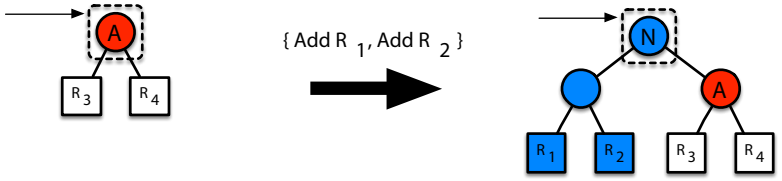
Definition 3. *Let $R(A, B)$ be the containment relation of two RenderJob trees. We say $R(A, B)$ iff there exists a sub tree of A equal to B (A contains B).*

Structure of the algorithm

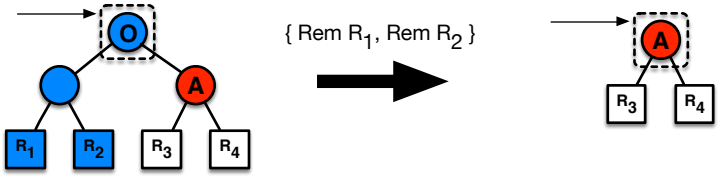
The proposed algorithm only works for full binary trees. From a high-level point of view, the algorithm installs dummy readers for each inner binary tree node. If the structure of the node changes, the algorithm computes a list of changes reflecting the modified structure.

Let us summarize the preconditions for computing changes:

[3] A non generic version of `BTree<'a>` with type parameters specialized to `RenderJob`



(a) $R(N, A)$, i.e. N contains A. All nodes of N which are not A are additions to A. The operation corresponds to `write(root, BNode(N, unsafeRead root))`.



(b) $R(O, A)$, i.e. O contains A. All nodes of O except for A and its sub tree are removals. The operation corresponds to `write(root, unsafeRead root.Right)`.

Figure 5.1: Two cases of containment relation and its meaning in terms of *additions* and *removals*. Change propagation or user modification modifies the root cell (dotted rectangle).

1. Each node reader is executed before its child nodes i.e. execution is essentially pre-order.
2. The binary tree is full.

The algorithm works in two phases:

Initialization. We traverse new trees exactly once and install independent readers r for each modifiable node. Each reader is re-executed if the node cell is modified structurally. Since Δ is the new tree, additional traversal has runtime proportional to the size of the new tree.

Change Propagation. A reader r of an inner node is re-executed if the structure of the node is modified. Since readers are independent (due to construction) and executed in pre-order due to pre-assumption (1) each reader handles modifications of its original node modifiable. Change propagation involves three steps:

Computing containment. Nodes internally mutate via modification and change propagation. When computing the containment relation for some non leaf node n, comparing n by reference is sufficient, since child nodes have their own readers installed, thus their own registration, which takes care of internal changes. A sketch computing the containment relation is given in Listing. 5. Note that when computing containment, we traverse both trees (old and new) in in parallel, i.e. the traversal expands child nodes alternatingly. Intuitively this scheme provides an argument for

the performance of the algorithm, since the algorithm terminates exactly when one tree has been visited completely.

Compute delta In order the compute the complete set of removals and additions, we compute both sets seperately. Given the containment relation $R(old, new)$ the algorithm traverses old, while omitting the new. All visited leaf nodes of old are no longer in new, thus add them to change list (removals). Given containment relation $R(new, old)$, i.e. new contains old, the algorithm traverses new while omitting old. All visited leaf nodes of new are additions, thus add them to change list (additions). The concept is depicted in Fig. 5.1. A sketch of the implementation is given in Listing. 5.

Maintain registrations Remove registrations of inner nodes no longer appearing in the new tree. Analogously add new registrations for new inner nodes.

Complexity of computing containment relationship

Let A, B be RenderJobs sets represented as full binary trees. If A and B are equal, i.e. describe the same RenderJob set and share the same structure, change propagation and therefore the algorithm terminates immediately. Otherwise A and B are not equal, i.e. the change list is non-empty containing n elements consisting of a additions and r removals.

Theorem 2. *Given the old state of an inner node of some render job tree, and the new state, the containment relationship can be computed in $O(\Delta)$.*

Proof 1. *There are two cases:*

- *For A and B there is neither $R(A, B)$, nor $R(B, A)$, i.e. they have no contains relation which means the complete scene graph changed. In this case the algorithm visits $2 * r - 1$ nodes, reflecting removals, and $2 * a - 1$ additions. Thus the algorithm runs in $O(\Delta)$.*

- *Without loss of generality $R(A, B)$, i.e. A contains B. Let k be new RenderJobs appearing in A, but not in B.*

  1. *In worst case, the original tree B is lowest, right in the new sub tree. With k being the number of leafs in the new sub tree, the number of nodes to be visited in total is $2 * k - 1$.*
  2. *Our alternating traversal scheme ensures, that the number of traversed nodes differs at most by 1. Thus, computing the containment relation runs in $O(2 * k)$.*

  Given (1) and (2) our algorithm for computing containment relation runs in $O(4 * k)$, which is indeed $O(\Delta)$.

Given a proper containment relation, it suffices to traverse the bigger tree omitting the contained smaller tree. Let N be the new updated tree and O the old structure of a modified inner node. There are two cases:

- $R(N, O)$, i.e. the new tree is bigger. All nodes visited are additions (Fig. 5.8.1). The algorithm visits add all new nodes, which is Δ, thus computing additions runs in $O(\Delta)$.
- $R(O, N)$, i.e. the new tree is smaller. All nodes visited are deletions (Fig. 5.8.1). The algorithm visits all removed nodes. Thus computing deletions runs in $O(\Delta)$.

Output of the algorithm

The proposed algorithm may detect self-vanishing[4] add-remove sequences (i.e. `remove x`,`add x`). Since the complete delta list needs to be traversed by the backend, such phantom modifications can be removed with memory overhead proportional to the delta list (using a hash table).

Discussion and other approaches

Computing deletions is very similar to automatic memory management. Additions in turn are similar to allocations in memory management. In an early implementation we therefore used change propagation (*splice-out*) and allocation for tracking changes in render job sets. Although a valid approach, we found it to be intransparent and cumbersome. Our current algorithm by contrast is explicit about modification.

[4] Phantom changes, occur due to using shallow equality for comparing trees.

```
let oldValues = Dictionary<Mod<BTree<RenderJob>>,BTree<RenderJob>>()
// active reader functions
let m_readerFunctions = Dictionary<Mod<BTree<RenderJob>>,Edge>()

let deregister (delta : List<Change>) (m : Mod<BTree<RenderJob>>)  =
    match m_readerFunctions.TryGetValue(m) with
        | (true,reader) -> (m:>IMod).RemoveOutEdge(reader)
                           m_readerFunctions.Remove(m) |> ignore

// read modifiable node and store reader function in m_readerFunctions
let rec register (delta : List<Change>) (m : Mod<BTree<RenderJob>>) =
    mkMod (fun d ->
                let reader = readAndGetEdge (m, fun t -> computeDelta m delta)
                m_readerFunctions.Add(m,reader)
                write (d, ())
          )   |> ignore

// traverse binary tree omitting endNode, execute edgeAction per node,
// execute leaf action per leaf (use to build Δ set)
and traverseEdgesTo (original : BTree<RenderJob>) (endNode : BTree<RenderJob>)
                    (edgeAction : Mod<BTree<'a>> -> unit)
                    (leafAction : BTree<'a>> -> unit) = ...

// traverse complete binary tree, execute edgeAction per node,
// execute leaf action per leaf (use to build Δ set)
and traverseAllEdges (tree : BTree<RenderJob>)
                     (edgeAction : Mod<BTree<'a>> -> unit)
                     (leafAction : BTree<'a>> -> unit) = ...

// computes delta for changes of one single modifiable node.
and computeDelta (m : Mod<BTree<RenderJob>>) (result : List<Change>) : unit =
    let removeLeaf t = match t with
                        | Leaf(v) -> result.Add(Removal v)
    let addLeaf t = match t with
                      | Leaf(v) -> result.Add(Addition v)
    // read new structure, new structure is consistent due to
    // change propagation of adaptive
    let newValue = unsafeRead m
    match oldValues.TryGetValue(m) with
        | (true, oldValue) ->
                // find oldValue in m, compute containment relation.
                let cmp = findRelation oldValue newValue result

                // adjust registration, deregister values no longer
                // appearing in newValue. same for new values
                if cmp = NewContainsOld
                    then // newValue is in oldValue
                         traverseEdgesTo oldValue newValue (deregister result) removeLeaf
                elif cmp = OldContainsNew
                    then // oldValue is in newValue
                         traverseEdgesTo newValue oldValue (register result) addLeaf
                else
                    // no relation, deregister all in old value,
                    // register all in newValue
                    traverseAllEdges oldValue (deregister result) removeLeaf
                    traverseAllEdges newValue (register result) addLeaf
        | _ -> traverseAllEdges newValue (register result) addLeaf
```

Listing 5: `computeDelta` computes a list of changes and adds them to the result list. The algorithm runs in $O(\Delta)$. `findRelation` is given in Listing. 6

```
type Current = Old | New
type Relation = NewContainsOld | OldContainsNew | NoRelation
let findRelation (old : BTree<'a>) (node : BTree<'a>) : Relation =

    let oldActive = Queue<BTree<'a>>()
    let newActive = Queue<BTree<'a>>()
    let current = ref Old // start with old

    // enqueues children of node to q
    let enqueueTree t (q:Queue<BTree<'a>>) =
        match t with
         | BNode(l,r) -> q.Enqueue l; q.Enqueue r;
         | _          -> ()

    // enqueue root nodes
    enqueueTree old oldActive
    enqueueTree node newActive

    // switches current
    let alternate c = match c with
                      | Old -> New
                      | New -> Old

    // takes from current queue if possible, otherwise returns none
    let takeNext (c:Current) =
        match c,newActive.Count,oldActive.Count with
         | New,cnt,_ when cnt > 0 -> Some <| newActive.Dequeue()
         | New,_,_ -> None
         | Old,_,cnt when cnt > 0 -> Some <| oldActive.Dequeue()
         | Old,_,_ -> None

    let rec run current =
        // no more nodes on both sides, so there is no relation, i.e. all new
        if oldActive.Count = 0 && newActive.Count = 0 then NoRelation
        else
            let other = alternate current
            let next  = takeNext current
            match next,current with
             | None,_    -> run other
             | Some(v),New ->
                if System.Object.ReferenceEquals(v,old) then
                  // old is in new (new contains old, R(new, old))
                  NewContainsOld
                else enqueueTree v newActive
                     run other
             | Some(v),Old ->
                if System.Object.ReferenceEquals(v,node) then
                  // new is in old (old contains new, R(old, new))
                  OldContainsNew
                else enqueueTree v oldActive
                     run other

    run New
```

Listing 6: Parallel traversal algorithm for finding containment relationship of two full binary trees.

5.9 Scene Graphs as Attribute Grammars - Nodes and Semantics

Consider a simple scene graph with some basic node types:

RenderNode Contains draw call parameters (e.g. draw mode[5] and primitive count) as an immutable structure called `DrawCallInfo`. `RenderNode` allows `DrawCallInfo` to be modified using the *computation framework*. This enables efficient manipulation of draw calls, like switching draw modes.

TrafoApplicator Applies spatial transformation to its modifiable child sub graph. Note that the transformation itself should be changeable as well.

Group Renders all child nodes in unspecified order. `Group` group is basically a set of scene graphs.

VertexAttributeNode Applies a set of vertex attributes (e.g. *Positions*, *Normals*, *Coordinates*,...) to some modifiable child sub graph.

In order to abstract over scene graph nodes we introduce a common marker interface, implemented by all nodes: `ISg`.

Additionally observe, that `TrafoApplicator` and `VertexAttributeNode` share a similar structure.

Let us abstract over the structure by introducing a common interface: `IApplicator`.

`IApplicator` contains exactly one member property, returning a modifiable sub scene graph.

Note that traditionally leaf nodes contain all necessary vertex data to be rendered. In our design we describe geometry attributes with ordinary attributes, thus `RenderNode` contains no geometry itself.

A straightforward object-oriented encoding using F# looks like:

```
1  (* marker interface *)
2  type ISg = interface end
3
4  (* all nodes with modifiable childs *)
5  type IApplicator = interface
6      inherit ISg
7      abstract member Child : ISg Mod
8      end
9
10 (* render node equipped with modifiable draw call info
11    struct (draw mode, primitive count etc.) *)
12 type RenderNode(callInfo : DrawCallInfo) =
13     interface ISg
14     member x.DrawCallInfo = new Mutable<DrawCallInfo>(callInfo)
15
16 (* applies a trafo computation to some modifiable child scene graph *)
17 type TrafoApplicator(trafo : Mod<Trafo3d>, child : ISg Mod) =
18     interface IApplicator with
19         member x.Child = child
20
21     member x.Child = child
22     member x.Trafo = trafo
23
```

[5] `PrimitiveTopology` in Direct3D 10

```
(* adaptive group implementation with imperative interface
   for additions and removals (implementation omitted *)
type Group(xs : seq<ISg>) =
    interface ISg

    member x.Add(elem : ISg) = ...
    member x.Rem(elem : ISg) = ...
    member x.Children : Mod<BTree<ISg>> = ...
```

Since RenderNode contains only metadata of the draw call, we define an additional node which we use for providing vertex data to some sub scene graph:

```
type VertexAttributeNode(vertexAttributes : Dictionary<Computation<Array>>,
                         indices : Computation<Array>, child : ISg Mod) =
    interface IApplicator with
        member x.Child = child

    member x.Child = child
    member x.Attributes = vertexAttributes
    member x.Indices = indices
```

In order to access vertex data in leaf nodes we need a way to expose vertex data to child nodes. Therefore, let us introduce an inherited attribute `VertexAttributes`, which we assign in the semantics for `VertexAttributeNode`:

```
[<Semantic>]
type VertexAttributeSem() =
    member x.VertexAttributes(va : VertexAttributeNode) =
        va.Child?VertexAttributes <- va.Attributes
```

Next let us introduce *semantics* for rendering. As indicated earlier instead of using side-effects we build an explicit representation of *RenderJobs*, which we compute for each scene graph node by synthesizing the attribute *RenderJobs*.

```
[<Semantic>]
type RenderSem() =

    member x.RenderJobs(r : RenderNode) : Mod<BTree<RenderJob>> =
        let va = r?VertexAttributes    // Lookup inherited attrib
        let t = r?ModelTrafo           // Lookup uniforms (e.g. trafos)
        let job = makeRenderJob r va t // make renderjob for given
        initMod (BLeaf job)            // Wrap into incremental RenderJob tree

    member x.RenderJobs(e : IApplicator) : Mod<BTree<RenderJob>> =
        adaptive' {
            let! c = e.Child       // Read child
            return! c?RenderJobs() // Compute RenderJobs for child
        }

    member x.RenderJobs(g : Group) : Mod<BTree<RenderJob>> =
        aggregateBTree g.Children (fun sg -> sg?RenderJobs())
```

Note that the implementation of *RenderNode* shown here is simplified. In practice a *RenderJob* requires additional attributes like *surface* and *uniform parameters*. In our prototypical implementation however, we simply use *VertexAttributes* and *ModelTrafo* for specifying a *RenderJob*. The semantic function for group nodes involves computing an aggregate *RenderJob* for its children. In order to accomplish this, we use an adaptive version of the *map* function, known from functional programming (see Listing 7)

```
let rec aggregateBTree (tree : Mod<BTree<'a>>)
                       (f : 'a -> Mod<BTree<'b>>) : Mod<BTree<'b>> =
    adaptive {
        let! t = tree // read node
        match t with
            | BNil -> return BNil        // keep nils
            | BLeaf(v) -> let! nv = f v  // use adaptive per leaf function
                          return nv
            | BNode(l,r) -> // Create a new BNode, memoize its result
                            // Hence, contained nodes handle inner modifications
                            // themselfs
                            return! memoize (fun l1 r1 ->
                                                initMod (BNode(modMapFoldBTree l1 f,
                                                        modMapFoldBTree r1 f))
                                            ) l r

    }
```

Listing 7: Adaptive aggregation function mapping over adaptive binary trees preserving the structure.

The implementation is very similar to the standard map function for binary trees. In order to make it efficient, recursive calls need to be memoized. Details on memoization for recursive adaptive function is out of the scope of this paper and can be found in Acar's thesis [1].

Next, as an example for introducing new operations, we define semantics for computing the world-space bounding box. Therefore, we use the synthesized attribute *BoundingBox* to compute the world-space bounding box of a given scene graph. There are essentially two methods for computing bounding boxes using attribute grammars:

1. Additionally to *BoundingBox*, define an *inherited* attribute describing model to world transformations (*ModelToWorld*). `TrafoApplicators` modify the *inherited* transformation attribute of its sub graph, by applying their transformation. In all leaf nodes (`RenderNodes`) we now compute a bounding box in model space and transform it by the inherited *ModelToWorld* attribute. In this case, for each render node the bounding box is computed directly in the leaf node. This implementation technique is efficient, if the structure of the scene tends to change rarely.

2. The semantic function for *BoundingBox* in *TrafoApplicator* productions now computes the *BoundingBox* for its subgraph, applies its transformation and returns the transformed bounding box. In this case each *TrafoApplicator* transforms the bounding box of the sub scene graph.

Both variants can be useful in practice. In our implementation, we use method (2):

```
[<Semantic>]
type BoundingBoxSem() =
    member x.BoundingBox(node : RenderNode) : Mod<Box3d> =
        // Lookup inherited attribute <VertexAttributes>
        let va = node?VertexAttributes : Dictionary<Computation<Array>>
        computeBBForVertices va

    member x.BoundingBox(app : IApplicator) : Mod<Box3d> =
```

```
9           adaptive {
10              let! c = app.Child      // read child scene graph
11              return! c?BoundingBox()
12          }
13
14      member x.BoundingBox(g : Group) : Mod<Box3d> =
15          foldBTree ( fun c -> c?BoundingBox()  )  // extract BB for each child
16                    ( fun c b -> Box3d(c,b)     ) // compose BBs by using union
17                    ( initMod <| Box3d.Invalid)   // initial seed value
18                    g.Children                    // children
19
20      member x.BoundingBox(app : TrafoApplicator) : Mod<Box3d> =
21          adaptive {
22              let! c = app.Child
23              let! bb = c?BoundingBox()              // compute BB for child
24              let! trafo = app.Trafo                 // read app's trafo
25              return bb.Transformed(trafo)           // apply trafo and return
26          }
```

Note, the implementation for groups. Similar to `aggregateRenderJobs`, `bTreeFoldM` corresponds to an adaptive version of *fold*, known from functional programming.[6]

Let us now extend the scene graph system with a new node type stressing extensibility. Level of detail (LoD), can be described hierarchically by using binary scene nodes where one child represents a low quality version of the second child. Switching nodes shall be flexible, i.e. not hard-coded in the semantics of the node. A traditional solution for plugging in different implementations of LoD is the *strategy pattern* [18]. In our implementation however, we use a first class function of type: `Box3d -> Trafo3d -> bool` for parameterizing LoD nodes:

```
1   type LodNode(viewDecider : (Box3d -> Trafo3d -> bool),
2                low : Mod<ISg>, high : Mod<ISg>) =
3       interface ISg
4
5       member x.Low = low
6       member x.High = high
7       member x.ViewDecider = viewDecider
```

Our scene graph system currently supports synthesized attributes:

1. BoundingBox
2. RenderJobs

Thus, we need to provide *semantic functions* for those attributes for level of detail nodes as well.

Note that in our implementation the structure of semantic functions is up to the user. One could extend *BoundingBoxSem* with additional semantic functions for LoD. However, this code could be part of a library, the programmer has no direct access to. Alternatively, one can implement syntax and semantics of the new feature in one central place.

One important property of our approach is abstraction of similar nodes. Since *BoundingBoxSem* contains an implementation for nodes of type `IApplicator`, new node types can use those implementation by implementing `IApplicator`. For clarity we use an explicit implementation of *BoundingBox* for nodes of type *LoD*.

[6] C# programmers know the function from *LINQ*, where it is called *Aggregate*

For computing a bounding box of a LoD node, we simply delegate the computation to the simplified child. For *RenderJobs*, we actually need to execute our decision function in order to decide what to render:

```
[<Semantic>]
type LoD() =
    member x.BoundingBox(n : LodNode) : Mod<Box3d> =
        adaptive {
            let! c = n.Low
            return! c?BoundingBox()
        }

    member x.RenderJobs(node : LodNode) : Mod<BTree<RenderJob>> =
        adaptive' {
            // Read <low> sg
            let! lowSg = node.Low
            // Compute bounding box of lowSg
            let! bb = lowSg?BoundingBox()
            // Read <high> sg
            let! highSg = node.High
            // Compute <low> render jobs
            let! lowJobs = lowSg?RenderJobs()
            // Compute <high> render jobs
            let! highJobs = highSg?RenderJobs()
            //read the trafo
            let! trafo = node?ModelViewTrafo

            //view logic
            if node.ViewDecider bb trafo then
                return highJobs
            else
                return lowJobs
        }
```

Since all fields are modifiable itself, the *RenderJobs* semantic must be adaptive. In our implementation we first read the scene graphs in order to compute both render jobs. Additionally, we use the *BoundingBox* attribute to extract a simplified bounding box which guides the level of detail decision. Level of detail systems often perform pre-fetching and asynchronous loading which can be integrated in the semantic function directly.

Let us now analyze distinctive features of our design and implementation:

- There is no typing restriction for node types. Semantics may be defined for arbitrary nodes, there is no need to inherit special abstract classes or interfaces. This way, it is easy to add semantics to existing node implementations.

- In contrast to traditional scene graph systems, geometry is defined just like any other attribute. This uniformity allows for flexible node descriptions and abstraction.

- All computations subject to value changes can be implemented using computation types. Structural changes by contrast are modeled by adaptive computations. This design allows for fined-grained control of updates without resorting to event style programming.

- Subtyping can be used to abstract similar features. As an example consider *IApplicator*. Whenever a new node type has a single child scene graph, we implement the *IApplica-*

tor interface for the new node. Given implementations of other attributes, operating on *IApplicator* instances, default implementations for the new node come for free.

- The system is extensible in respect to new operations, i.e. synthesized attributes — each aspect (*Rendering*, *BoundingBox*, *Level of Detail*) can be implemented as a coherent unit providing all necessary semantic functions.
- Rendering semantics is explicit via render jobs and render job trees. The explicit modeling allows render jobs to appear as arguments to other functions, basically promoting render jobs to first-class values (e.g. *Group* semantics for *RenderJobs* and *BoundingBox*).
- Adaptive semantic functions allow for precise memory management.
  Consider `TrafoApplicators`: If the child scene graph is modified, change propagation (*splice out*) gets rid of all data associated with the old scene graph. Same applies for *LoD* nodes. In traditional scene graph systems, memory management is cumbersome when it comes to dynamic scenes.
- Although the attribute grammar is essentially dynamically typed, type inference integrates fine with adaptive semantic functions, since types of attributes can be inferred locally in most cases.

CHAPTER 6

Evaluation

This chapter is divided into two sections. Firstly we evaluate how our design based on attribute grammars compares to traditional object-oriented scene graph designs. To this end we analyze required lines of code for reusable scene graph nodes as required by prototypical scenes. Secondly we compare our system's performance with traditional scene graph rendering based on handcrafted traversals.

6.1 Software Design

In this section we consider the scenario of dynamic transformations assigned to specific scene entities. As an example consider a windmill with rotating sails. The task should be solved as simple as possible while providing reusability for similar use cases.

We compare our system to a simple visitor based solution as well as Tobler's semantic scene graph [51], which is most similar to our system in respect to extensibility and expressiveness.

Our embedded domain specific language for attribute grammars and adaptive functional programming is implemented in F#. Although in principle, our system can be used from C#, the F# version is cleaner and provides additional syntactic sugar, which we consider to be important from a software engineering perspective. In order to provide a fair comparison we provide all implementations in F#.

In our measurements we exclude basic framework functionality. More specifically we assume implementations of default scene graph nodes such as `Group`. However, in cases which require extensions of framework code we take those into account, when counting lines of code.

6.1.1 Example: Animated Trafo

The scene graph system shall be used to animate entities depending on some time-varying value. In order to provide a modular system we use a special scene graph node which supplies time values to its sub scene graph. The advantage of this approach is that different scene graphs can use different time values. Additionally, code for supplying time values is decoupled from

code depending on time, which can be handy for non real-time applications. In our example we simply use a rotation matrix, whose angle depends on time. Code concerning animation should be modular and replaceable. Therefore, our animation node (`MyRotorTrafo`), uses a user supplied animation function instead of computing the animated transformation directly. In object-oriented terms this decoupling can be achieved by the *Strategy pattern* [18], while in functional programming we simply use a function which is used by the animation node.

Visitor

The visitor pattern [18] as used by OpenSceneGraph [36] is a common implementation technique for scene graphs.

For simplicity and familiarity our implementation is based on the standard design pattern given in Gamma et al. [18].

`ISg` represents our base interface for all node types. Additionally we define a generic variant of `accept` method as well as a generic visitor.

```
type ISg =
  abstract member Accept : Visitor<'a> -> 'a

and Visitor<'a> =
  abstract member Visit : Leaf -> 'a
```

Leaf nodes, i.e. geometry nodes are represented by the `Leaf` class, implementing `ISg`. In our test setup we simulate rendering by printing a string representation of the draw call containing all arguments needed for rendering (transformation matrix in our simplified case).

```
type Leaf(s : string) =
  interface ISg with
    member x.Accept(v) = v.Visit(x)

  member x.Name = s

and DefaultRenderVisitor() =
    interface Visitor<int> with
        member x.Visit(r : Leaf) =
          printfn "render node: %s with trafo: %A" r.Name x.Trafo; 0

    member val Trafo = Trafo3d.Identity with get,set
```

Next, we introduce a new node `Scene`, which is constructed with a function providing time values (function `f`). Additionally we define another node `RotorTrafo`, which is constructed with an update function of type: `double -> Trafo3d` representing a function computing a transformation matrix given some time value of type `double`.

```
type Scene(f : unit -> double, sg : ISg) =
  interface ISg with
    member x.Accept(v) = v.Visit x

  member x.SceneGraph = sg
  member x.Time = f

and RotorTrafo(f : double -> Trafo3d, sg : ISg) =
  let trafo = ref (f 0.0)

  interface ISg with
```

```
12      member x.Accept(v) = v.Visit x
13
14    member x.UpdateWithTime(t : double) =
15      trafo := f t
16
17    member x.GetTrafo () = !trafo
18    member x.SceneGraph = sg
```

In order to render the scene graph we need to assign proper values to all dynamic attributes (time and transformation). Therefore we construct a new visitor `SetTimeVisitor` which assigns the current time to a given sub scene graph. To this end, the `visit` method overload taking values of type `Scene` computes a new time and assigns the computed value to some local state of the visitor (`currentTime`). For arguments of type `RotorTrafo`, we update the instance with the new time value given in the local state. Since the visitor pattern provides no proper solution to the expression problem, we need to adapt framework interfaces in order to provide the visitor with additional visit methods for new scene graph nodes. Therefore we inherit the original rendering visitor and provide additional visitor methods for `Scene` and `RotorTrafo`. The later computes the transformation by using its `GetTrafo` method and assigning the result to the base visitors public `Trafo` field.

```
1   type SetTimeVisitor() =
2     let currentTime = ref 0.0
3     interface Visitor<int> with
4       member x.Visit(scene : Scene) =
5          currentTime := scene.Time () // get time and set mutable ref
6          scene.SceneGraph.Accept(x)
7
8       member x.Visit(r : RotorTrafo) =
9          r.UpdateWithTime(!currentTime)
10         r.SceneGraph.Accept(x)
11
12      member x.Visit(l : Leaf) = 0;
13
14    member x.SetTime t =
15      currentTime := t
16
17  // introduce new Render visitor. modify ALL uses of Render visitors!
18  and MyRender() =
19    inherit DefaultRenderVisitor()
20    interface Visitor<int> with
21      member x.Visit(scene : Scene) =
22        scene.SceneGraph.Accept(x)
23      member x.Visit(r : RotorTrafo) =
24        x.Trafo <- r.GetTrafo ()
25        r.SceneGraph.Accept(x)
```

Note that due to the addition of new scene graph nodes, we need to adapt framework code in order to satisfy the type checker[1]:

```
1   // fix up Visitor interface or introduce inherited Visitor and
2   // fixup all uses in class hierarchy.
3   type Visitor<'a> =
4     abstract member Visit : Leaf       -> 'a
5     abstract member Visit : Scene      -> 'a
```

[1] This violates separate compilation which makes clear, that the visitor pattern provides no solution to the *expression problem*. This limitation can be accepted if separate compilation is of no concern and application developers may access and modify framework code.

```
6     abstract member Visit : RotorTrafo -> 'a
7
8   // fixup default render visitor
9   //(either abstract method declarations or virtual method stubs)
10  and DefaultRenderVisitor() =
11    interface Visitor<int> with
12      member x.Visit(scene : Scene)  = failwith "extend with proper method"; 0
13      member x.Visit(r : RotorTrafo) = failwith "extend with proper method"; 0
14      member x.Visit(r : Leaf) =
15        printfn "render node: %s with trafo: %A" r.Name x.Trafo; 0
```

Client code needs to be adapted carefully as well. For correctness it is necessary to perform traversals in the correct order. Otherwise attributes (like `time`), may not be assigned properly:

```
1   let time = ref 0.0
2   let objs = RotorTrafo((fun t -> Trafo3d.RotationZ t), Leaf "object 1" :> ISg)
3   let scene = Scene((fun () -> let t = !time
4                                time := t + 1.0
5                                t),
6                     objs) :> ISg
7   let setTime = SetTimeVisitor()
8   let renderVisitor = MyRender()
9   scene.Accept(setTime) |> ignore
10  let result = scene.Accept(renderVisitor) |> ignore
11  result
```

Semantic scene graph

In this section we provide a solution to our use case using the *semantic scene graph* approach introduced by Tobler [51]. First, we introduce a scene node which provides time values to its sub scene graph. Therefore we create a scene graph node `Scene`:

```
1   type Scene(f : unit -> double, sg : ISg) =
2       inherit Instance("Scene")
3
4       member val Identifier = "Time" with get,set
5       member x.Time = f
6       member x.SceneGraph = sg
```

Next, we provide a rule implementation for scene nodes. Each traversal function is supplied with a traversal state object, which is used to store time values. Note that for convenience we use an entry in the environment field of the traversal state which is essentially a map from string to object with some convenience functions hiding runtime casts. This approach allows other traversals to simply access this property. Note that this is an implementation detail and we could as well use an extra field in the traversal state. This however, requires modification in framework code because, the traversal state is part of the core framework.

Since `SetParameters` returns the sub scene graph stored in the instance, the rendering scene graph is dynamic in general. At first glance this flexibility comes for free in Tobler's system. This is not quite true, since for proper memory management it is necessary to track all returned sub scene graphs in order to prevent traversals from leaking memory. In our implementation we use a local field of type `HashSet` to track returned scene graphs. When removing the scene rule (`DisposeAndRemove(.)`) we dispose all sub scene graphs as well:

```
[<Rule(typeof<Scene>)>] // register rule for scene instance
type SceneRule(instance : Scene, t : AbstractTraversal) =
    // keeps track of all returned scene graphs
  // in order to support proper disposal
    let returnedSgs = HashSet<ISg>()
    interface IRule with
        member x.InitForPath t = ()

        member x.SetParameters t =
            // compute time and publish in traversal state
            t.EnvironmentMap.[instance.Identifier] <- instance.Time ()
            // register returned scene graph
            x.ReturnSceneGraph instance.SceneGraph

        member x.DisposeAndRemove(t) =
            t.TryDisposeAndRemoveRule(instance, t, returnedSgs)

    member x.ReturnSceneGraph g =
        if not (returnedSgs.Contains g)
            then returnedSgs.Add g |> ignore
            else ()
        g
```

Next, we introduce our actual `RotorTrafo` which takes care of computing a new transformation matrix. Similarly to the animation function as used in the visitor implementation we use a function which computes a new transformation, given a traversal state:

```
type MyRotorTrafo(f : AbstractTraversal -> Trafo3d, sg : ISg) =
    inherit Instance("MyRotorTrafo")

    member x.Rotation = f
    member x.SceneGraph = sg


[<Rule(typeof<MyRotorTrafo>)>]
type MyRotorTrafoRule(instance : MyRotorTrafo, t : AbstractTraversal) =
    let leaf = MyTrafoLeaf(instance.Rotation(t))
    // effectively make m_instance.SceneGraph immutable,
    // values not synchronized between rule and instance
    let rsg = Trafo3dApplicator(instance.SceneGraph, leaf) :> ISg

    interface IRule with
        member x.InitForPath t = ()

        member x.SetParameters t =
            leaf.Value <- instance.Rotation t
            rsg

        member x.DisposeAndRemove t =
            t.TryDisposeAndRemoveRule(instance, t)
```

Finally let us take a look at client code creating a simple scene graph. Note that in contrast to the visitor based approach there is no need to modify framework code. Additionally no modifications at client code are necessary (no special new traversal etc.):

```
let obj = MyRotorTrafo((fun t ->
             // Consume time.
             let timeObj = t.EnvironmentMap.Get("TimeProvider")
             let time = timeObj :?> double // extract time.
             Trafo3d.RotationZ(time * 0.1)),
```

```
            Primitives.Box(C4b.DarkBlue).ToVertexGeometrySet())
let scene = Scene((fun () -> Kernel.T), // grab system time
          obj)
scene.Identifier <- "TimeProvider" // publish time in slot "TimeProvider"
let result = scene.Render()
```

Attribute grammar

First, we define the Scene node which provides time values to its sub scene graph, similarly to previous approaches. Our system works fundamentally different than previous approaches. Instead of repeatedly traversing the complete scene for rendering we compute the RenderJobs attribute once. Due to the incremental nature of our approach changes of arbitrary input modifiables are reflected in the RenderJobs attribute. Thus, instead of using a function for computing time values we use the computation type Computation<T>. Similarly, the child scene graph shall be modifiable in order to allow external modification of the scene. Since scene graph modification might be a structural change we use Mod<ISg> instead of lazy computations.

```
type Scene(f : Computation<double>, child : Mod<ISg>) =
  interface ISg

  member x.Time  = f
  member x.Child = child
```

Next, we need to define *semantics* for *rendering* and *time*. *time* is an inherited attribute i.e. the semantics function Time for Scene nodes assigns its current time computation to the inherited attribute Time. For rendering we need to define the semantics function RenderJobs. Applied with objects of type Scene, we read the modifiable sub scene graph, and compute the synthesized attribute RenderJobs recursively.

```
[<Semantic>]
type SceneTimeSem() =
  member x.Time(s : Scene) = s.Child?Time <- s.Time

  member x.RenderJobs(s : Scene) : Mod<BTree<RenderJob>> =
    adaptive {
      let! child = s.Child
      return! child?RenderJobs()
    }
```

Actually we need to provide semantic functions for all attributes, which might be passed through the new node. We solve this issue by abstraction. A common pattern is to define custom attributes for a specific nodes, while delegating all other attributes to the modifiable sub scene graph. Our scene graph framework contains a node which provides exactly this functionality:

```
// defined in core library

type IApplicator = interface
    inherit ISg
    abstract member Child : ISg Mod
    end

[<Semantic>]
type RenderApplicatorSem()
    member x.RenderJobs(e : IApplicator) : Mod<BTree<RenderJob>> =
        adaptive' {
```

```
            let! c = e.Child
            return! c?RenderJobs()
        }
```

By using the `IApplicator` interface our complete implementation of `Scene` and `Time` semantic reduces to:

```
type Scene(f : Computation<Time>, child : Mod<ISg>) =
  interface IApplicator with
   member x.Child = child

  member x.Time  = f

[<Semantic>]
type SceneTimeSem() =
  member x.Time(s : Scene) = s.Child?Time <- s.Time
```

Next, we introduce the new scene graph node `RotorTrafo` for describing dynamic transformations. Again we use the `IApplicator` interface instead of duplicating semantics for our new node. Note that transformations are of type `Computation<Trafo3d>` in order to support adaptivity. Thus we need to use the `|>=` combinator which performs some operation on two arguments of type `Computation<T>`.

```
type RotorTrafo(f : Computation<Time> -> Computation<Trafo3d>, child : Mod<ISg>) =
  interface IApplicator with
     member  x.Child = child

  member x.Rotation = f

[<Semantic>]
type RotorTrafoSem() =
  member x.ModelTrafo(rotor : RotorTrafo) : Computation<Trafo3d>=
    rotor.Child?ModelTrafo <- ( rotor?ModelTrafo,
                                rotor.Rotation rotor?Time) |>= (*)
```

From a client perspective, animated scene graphs can now be constructed like:

```
let mkScene () =
  let time : Computation<Time> = .. // aquire system time computation

  let cube = vgToSg (PolyMeshPrimitives.Box(C4b.Gray).GetIndexedGeometry()) :> ISg
  let cubeMod = initMod cube
  let trafoFun t = adaptive { let! t = t
                              return Trafo3d.RotationZ(t * 0.1)
                            }
  let objs = RotorTrafo( trafoFun, initMod cube) :> ISg

  Scene(time, initMod objs) :> ISg
```

A constructive comparison of previously presented approaches

Next, we analyze the different approaches and summarize distinctions. Additionally, we analyze code complexity and measure lines of code of each approach.

Our toy use case is of manageable complexity for all approaches. All reference solutions however have weaknesses regarding extensibility, flexibility and maintainability.

We evaluate the presented solution in terms of extensibility, type-safety, code complexity and boilerplate code.

Extensibility When extending the *visitor* implementation with additional traversals the approach scales fine (e.g. `SetTimeVisitor`). New nodes can be handled by using default implementations and subtyping [29]. If a new node cannot be made proper subtype of an existing node, extensibility breaks and framework code as well as other visitors need to be adapted in order to support the new node type: New node types like `RotorTrafo` and `Scene` require modifications in framework code i.e. new methods need to be introduced in the `IVisitor<'a>` interface.

Tobler's *semantic scene graph* is extensible, since `SetParameters` can always be used as fallback solution if a node type does not implement behavior specific to new traversals. Adding new node types is easy as well, since all changes remain local to the implementation of its rule.

Similarly, our proposed system provides rich extensibility. With appropriate and extensible implementations of reusable framework nodes (e.g. `IApplicator`), new node types automatically inherit all attribute implementations of similar nodes. Thus, new node types define solely attributes associated to the new node type. Additionally, other attribute implementations remain local to the definition of the node, which allows for coherent implementation of similar attributes (aspects). As an example consider redefining rendering semantics for all nodes. In this case a single semantics class suffices:

```
1  [<Semantic>]
2  type RenderNoneIncremental() =
3    member x.NonIncrementalRenderJobs(v : VertexAttributeNode) =
4      makeRenderCall v
5    member x.NonIncrementalRenderJobs(app : IApplicator) =
6      (unsafeRead app.Child)?NonIncrementalRenderJobs()
```

Type safety The *visitor* approach provides full static type safety i.e. the type checker guarantees that no type error occurs at runtime.

Tobler's *semantic scene graph* does not provide full static type-safety. Recall, that rules are constructed by the runtime system (according to *semantic map*).

Thus, rule creators have type `Instance -> AbstractTraversal -> IRule`. Concrete rule constructors have type:
`ConcreteInstance -> AbstractTraversal -> ConcreteRule`. The instance parameter appears at contravariant position, hence the runtime system needs to cast instances to the concrete instance type. We consider this type hole to be benign, since type errors may only occur at instantiation time of rules which usually appears in startup code and is therefore easy to spot at development time. Another type hole occurs, when using the environment map for data-flow between traversals. Again, we consider this type hole to be benign, since it occurs rarely and data-flow needs to be documented anyways.

Our approach based on *attribute grammars* has several type holes when accessing attributes. Each attribute lookup using the `?` operator is basically unsafe. This limitation essentially corresponds to environment map lookups, as often used in Tobler's semantic scene graph. F# type inference however integrates well with `?` operators and appropriate types are typically deduced by the type-checker. Trivial contexts are vulnerable to

runtime errors, since types are usually weak and generic. Yet, annotations usually help to spot those runtime errors. Furthermore our implementation provides meaningful error messages in error situations which makes run-time type errors easier to spot.

Complexity We found that the *visitor approach* is quite complex to understand. Even if the pattern is clear, extensions are not trivial. The complex dispatch mechanism (static overloading and dynamic binding) makes the code difficult to understand which may result in subtle errors due to calling wrong overloads.

The *semantic scene graph* approach has clean semantics: Either a specific traversal implementation is called or traversal proceeds in the scene graph returned by `SetParameters`.

Attribute grammars are composed of semantic functions and attributes. The semantic is clean, well-founded and no magic is involved. Additionally the implementations are guided by types, inferred automatically by the type system.

Boilerplate code The *visitor approach* imposes a considerable amout of boilerplate code (`Accept`, `Visit` methods and inheritance).

Tobler's *semantic scene graph* comes with virtually no boilerplate code at client side. Explicit memory management requires tracking of dynamic scene graph, resulting in similar code in many rules. Yet, inheritance can be used to reduce boilerplate code to a minimum.

Our approach has a direct mapping from semantics to implementation and keeps boilerplate code to a minimum.

Memory management In the *visitor approach*, memory management is completely up to the user. Unmanaged resources (i.e. resources requiring explicit disposal) need to be collected (e.g. using a visitor) and disposed manually[2].

Tobler's *semantic scene graph* separates semantics from rendering which complicates memory management. Active rules are stored in the traversal cache and need to be reclaimed if parts of the scene graph become inactive or inaccessible. Thus, a special `DisposeAndRemove` traversal is required to invalidate traversal caches. This task however is not trivial due to the dynamic nature of scene graphs. In order to properly dispose all scene graphs returned by a dynamic rule, the rule needs to keep track of returned scene graphs. Furthermore, in general reference counting is required in order to support proper sharing of rules.

Integration of modifiables and semantics allows our approach to be implicit about memory management. Although our approach keeps track of active attributes, inactive attributes can be invalidated automatically.

Table 6.1 provides an overview of our analysis.

The direct mapping of semantics and semantic function, as employed by our approach significantly reduces the complexity and code length. In Table 6.2 we show required lines of code for each approach.

[2] disposal needs to be deterministic, thus traditional garbage collection cannot be used.

Approach	Extensible	Type safe	Complexity	Boilerplate code	Memory management
Visitor pattern	✗	✓	High	High	Explicit
Semantic scene graph	✓	✓	High	High	Explicit
Attribute Grammar	✓	✗	Low	Low	Automatic

Table 6.1: Software engineering and design aspects in comparison.

Use case	Semantic scene graph	Visitors	Attribute grammar
Animated Trafo	68 F# (132 C#)	50 F#	35 F#

Table 6.2: Code length for animated trafo case study.

6.2 Performance

In this section we perform empirical performance analysis of our system. Recall that our system operates fundamentally different than traditional scene graph systems. Instead of traversing the complete scene each frame in order to issue draw calls, our approach performs incremental updates of the *render job set*. Thus, comparing performance is non-trivial since traditional scene graph systems have completely different performance characteristics. Our system's performance is independent of the actual scene size, while traditional scene graph systems need to traverse the complete scene each frame. While traditional scene graph systems typically have small constant overhead in case of changes, our system's performance heavily depends on the size of the modification. In our tests we validate our system by measuring overheads introduced by incremental attribute evaluation. For fairness, our benchmarks are typically worst-case for our system. We show, that incremental evaluation pays of, even if large parts of the scene change each frame.

For reference, we use an implementation of Tobler's semantic scene graph [51], which is to the best of our knowledge the only system providing rich extensibility comparable to our system. Although at the time of writing there exists an actual rendering backend for our system, our benchmarks are purely artificial for simplicity and clarity. Instead of traversing the scene graph our system computes changes in a resulting render job set, which remains persistent at run-time. From a functional point of view this can be simulated in traditional scene graph systems by computing a set of geometries each frame. To this end we use a custom traversal, which computes the set of all containing geometries at the root node. In order to be fair, those geometries need to be renderable, i.e. equipped with transformation and surface attributes. Although not expressed by this setup, the incremental setup of render jobs serves as a good starting point for backend optimizations, since optimizations may directly operate on Δ-sets of render jobs. In traditional scene graph rendering, backend optimizations need build optimizations from scratch each frame, since their input is transient and changes each frame.

Although our approach is conceptually superior to traditional scene graph traversal, our approach imposes constant overhead in time and memory. In order evaluate overhead in practical scenarios, we evaluate those overheads in worst-case situations for our system.

The overall performance of our system is mainly influenced by:

- The attribute grammar evaluation algorithm.
- Our incremental computation framework.

Although the modules cannot be tested completely independent of each other we focus on benchmarks stressing either attribute grammar evaluation or incremental computation.

For all our tests we used an Intel(R) Core(TM) i7-920 @ 2.67GHz (4 cores with Hyper-Threading), 10GB RAM, 64bit Windows 8 and a NVIDIA GeForce GTX 680 graphics card with 2048MB memory. Both systems run on top of .NET4.5. In order to warm up the runtime system we discard first iterations of each test run.

6.2.1 Performance of structural changes

In contrast to traditional scene graph rendering which needs to traverse the complete scene over and over again, our system's update mechanism has running time proportional to the size of the change. In order to validate this claim in an empirical setting we use a dynamic scene.

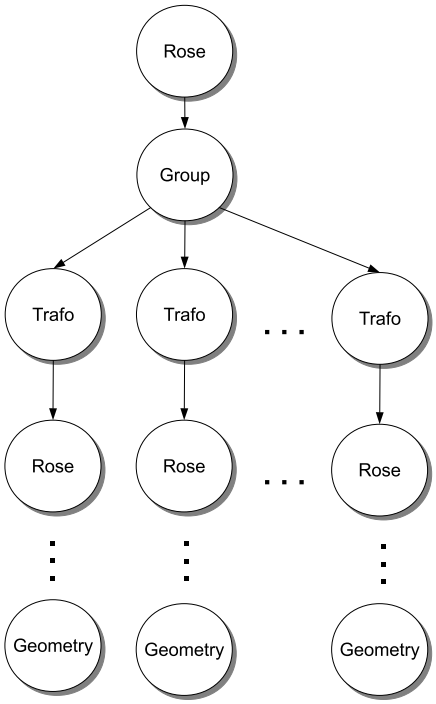


Figure 6.1: Tree structure of a scene graph. Each Rose tree contains a group with n children (branching factor). In order to simulate depth complexity we use additional transformation nodes as well as group nodes, which accumulate rendering semantics for child nodes.

The structure of our test scene is shown in Fig. 6.1. In order to simulate user interaction, as found in interactive applications we perform structural changes. Virtually these changes occur each frame. Note that in practice, the scene remains stable for most of the time, while changes appear rarely. In order to show overheads of our system, we perform structural changes of varying size. The scene, organized as a quad tree contains 4096 leaf nodes. At runtime, we repeatedly add nodes to the existing scene. We therefore randomly pick some inner node and add an additional child scene graph of varying size. The results are shown in Fig. 6.2.

Indeed, update propagation is practically free if there are no changes. The reference system by contrast uses time proportional to the original scene size (each frame), although nothing changes. If actual rendering is considered to be free, still the frame rate of the reference system is limited by traversal overhead. We also observe linear growth in update time, as the scene modification increases. Note that the reference system also needs additional time for modification. This is mainly due to object allocation and growth of the actual scene size.

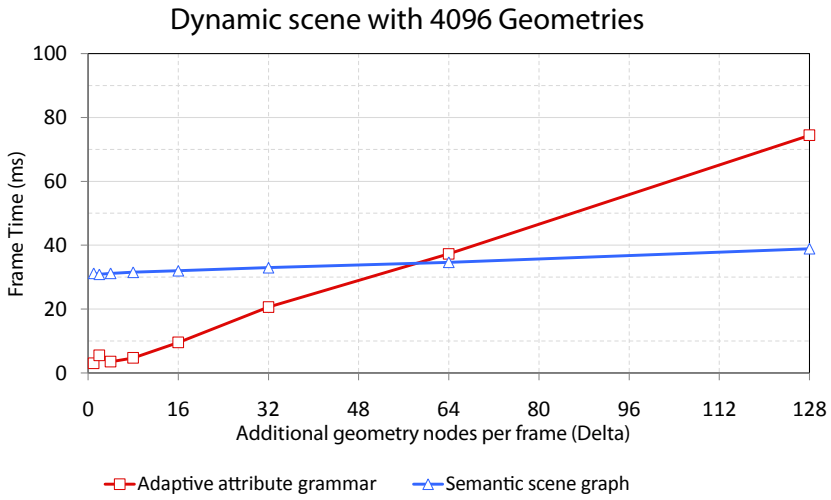


Figure 6.2: Hierarchically structured scene (see Fig. 6.1) with 4096 leaf nodes. We compare update propagation execution time of our system with non-incremental scene graph traversal time for varying size of modifications. For small changes our system is significantly faster than the reference solution. Beginning from 64 new geometry nodes per change, traditional scene graph traversal outperforms our approach due to overheads in attribute grammar evaluation and incremental computation.

At first glance our system seems to provide worse scalability, when compared to traditional scene graph traversals. Our results are promising, since scenes typically change gradually at run-time.

6.2.2 Performance of our incremental computation framework

In order to further assess overheads induced by our system, let us now consider constant overheads of the incremental computation framework. Incremental computation does not pay out, if most of the scene changes each frame.

In order to measure overheads induced by the framework we use a scene, with dynamic level of detail decisions. Our scene is naively structured as depicted in Fig. 6.3.

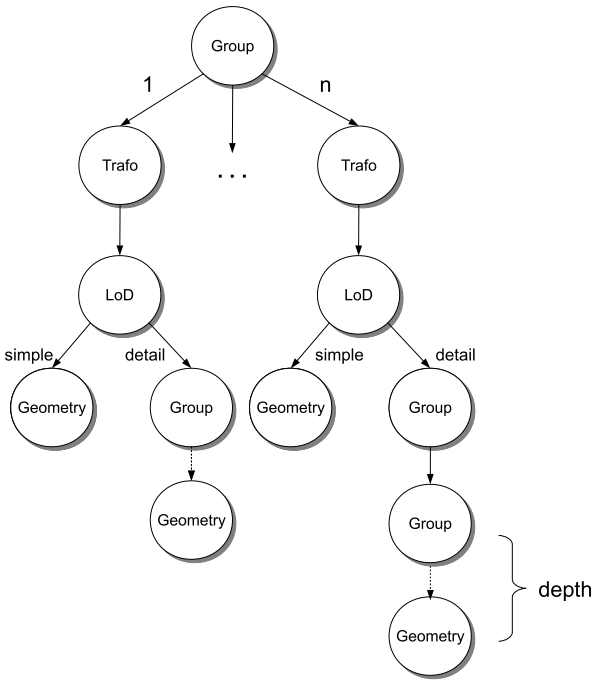


Figure 6.3: Scene graph structure of naive level of detail. Level of detail nodes are contained in one monolithic group node. Intermediate singleton group nodes are inserted right above level of detail nodes in order to simulate varying depth complexity.

Level of detail decisions are dependent on the current view transformation as well as the local bounding box of the sub tree. This structure represents a worst case situation for our system, since each level of detail node needs to be reevaluated if at least one input changes i.e. incremental evaluation virtually provides no benefit over non-incremental approaches.

Fig. 6.4 compares our approach with traditional scene graph traversals. Note that although, each level of detail decision needs to be re-executed each frame, our system outperforms traditional traversal. This is mainly due to the fact that our incremental computation framework is precise i.e. re-execution starts immediately at the changed input. In our test this means, update propagation starts re-execution directly within level of detail nodes i.e. no other nodes need to be traversed. In summary we found that the constant overhead to be low, compared to the benefits gained from incremental evaluation.

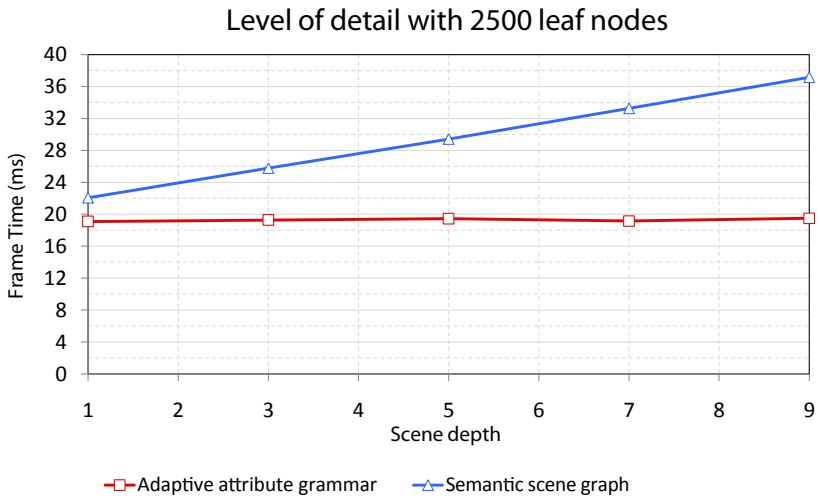


Figure 6.4: Flat scene graph with dynamic level of detail decisions for each leaf node. In our test setup, the camera changes each frame i.e. each level of detail node needs to be re-evaluated each frame. In order to analyze overheads of incremental computation running time of update propagation is compared to handcrafted traversal as used in traditional scene graph systems.

6.2.3 Performance of attribute grammar evaluation

Due to incremental evaluation, attribute grammar evaluation appeared not to be crucial for our system performance. Thus the design of our attribute grammar language was basically driven by usability and extensibility. At the current state, our system does not perform performance optimizations for attribute grammar evaluation. As expected, our current implementation of attribute grammars is slower than handcrafted traversals. In our benchmark we compute the resulting render-job set for a given scene graph of varying size. The structure of the input scene graph is depicted in Fig. 6.1. Our system assumes attribute evaluations to be rare at runtime. Still our proposed system is significantly slower at startup (Fig. 6.5). Note that this overhead is actually the overhead observed in the structural change benchmark presented earlier (6.2.1). The reference solution has practically no startup cost in our artificial test setup. Note that this is not true in practice, since resource allocations (GPU uploads) are typically the bottleneck at program startup. For scenes with thousands geometries of moderate size, our test setup including GPU uploads requires several seconds to load. In practical scenes, startup time is dominated by GPU uploads, geometry and texture processing which effectively makes startup time of the scene graph system less important. Still, relatively high startup cost revealed poor performance of our attribute evaluation scheme (see Chapter 7.1).

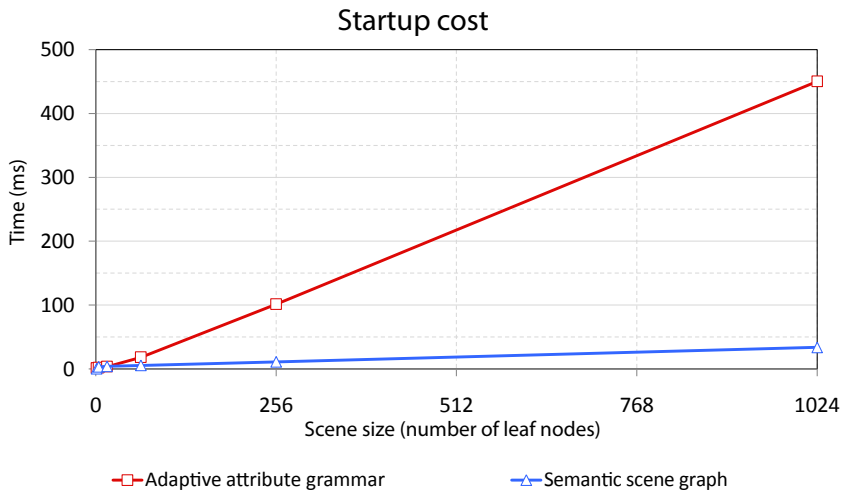


Figure 6.5: Time required to compute render jobs for scene of varying complexity. The scene structure is shown in Fig. 6.1. Additional overhead is introduced by the attribute grammar. Furthermore currently attribute evaluation employs no static analysis and optimization.

6.2.4 Outlook: Performance in Real-World Scenes

In the previous section we analyzed the worst case performance of our system in fully synthetic test scenes. We have shown, that our system is highly competitive even in fully dynamic scenes, while providing a high-level programming interface with high flexibility and novel expressiveness.

In practical scenarios most parts of the scene remain static, while small parts are modified due to simulation, animation or user input. Previously, scene graph caching (e.g. Wörister et al.) was used to reduce the traversal overhead for those static parts. Furthermore Wörister et al. demonstrated, that rendering caches give rise for a number of optimizations like *state sorting*, *redundancy removal* or *overdraw sorting*. In their system, those optimization operate on a per-cache basis. Placement of rendering caches is completely up to the user, thus dynamic parts of the scene need to be known in advance.

By contrast, our approach works completely incremental, without the need for explicit placement of rendering caches. This design gives raise to known optimizations operating on the complete scene, instead of static parts only. Although not part of this thesis, our group[3], implemented a proof of concept implementation of an incremental rendering backend. The architecture, as well as the interface corresponds to techniques presented in 5.8.

In order to validate our approach, we render the well-known Sponza scene, which represents a part of a typical game level (see Fig. 6.7). For comparison we use:

Semantic Scene Graph An implementation of Tobler's semantic scene graph concept. The

[3]Special thanks to my research group at VRVis Research Center, especially Georg Haaser

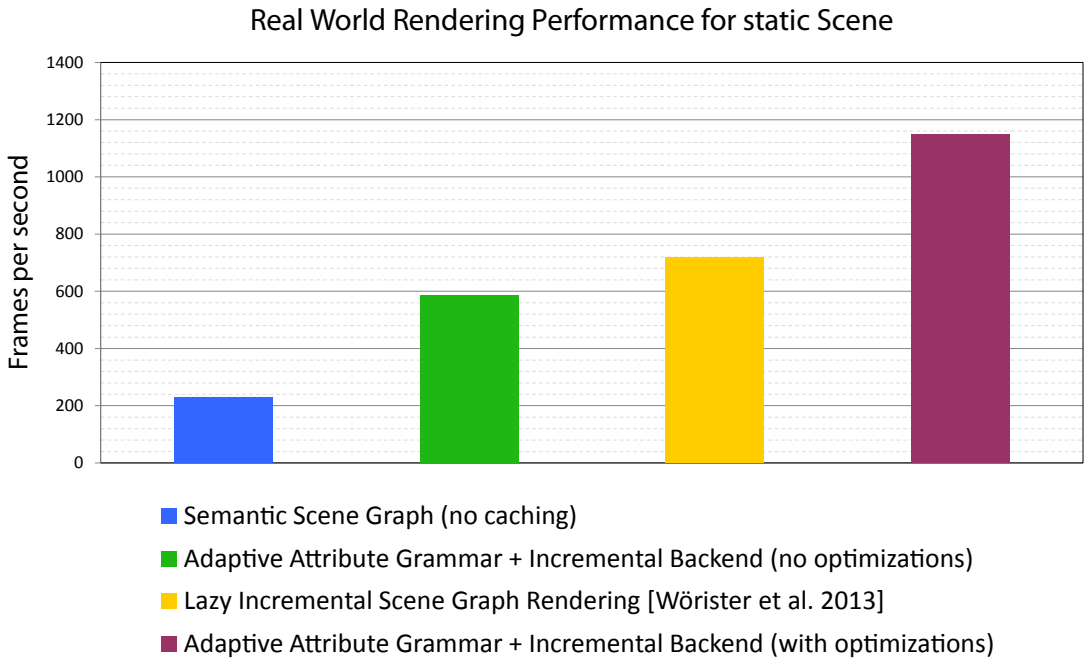


Figure 6.6: Performance (in frames per second) for different rendering systems. As test scene we use the well-known *Sponza* model (see Fig. 6.7). *Semantic Scene Graph* approach performs worst, since rendering is CPU bound due to high traversal overheads. Our system with optimizations enabled significantly outperforms *Lazy Incremental Scene Graph Rendering* Wörister et al.

systems traverses the scene each frame and issues draw calls when visiting a leaf node. The implementation is based on DirectX 10.

Lazy Incremental Scene Graph Rendering An implementation of the system proposed by Wörister et al. with all optimizations turned on. The implementation is based on DirectX 11.

Adaptive Attribute Grammar + Incremental Backend Our system with a prototypical rendering backend implemented in DirectX 11. For comparison we use two variants: One with all optimizations turned off, another one with *state sorting* and *redundancy removal*.

Although we have not been able to make our first tests with real scenes completely unbiased due ot the large number of variables involved (e.g. different optimization strategies, different culling strategies, ...) we include some first results here, as an indication of the potential of our approach, and as a confirmation, that we do not sacrifice rendering performance in our quest for an easily programmable and easily extendible framework.

The results are given in Fig. 6.6. Our system indeed integrates well with optimizing rendering backends and allows for a wide range of optimizations. In fact, our system outperforms

Figure 6.7: The Crytek Sponza Atrium Scene, representing part of a typical game level [12].

previous approaches, although we use a more generalized approach to caching, which allows for arbitrary changes in the scene.

6.2.5 Additional costs and overhead

Our system introduces some additional runtime overhead as shown previously. Most significantly attribute evaluation is slow, compared to handcrafted traversals. Additionally the incremental computation requires auxiliary data structures and numerous small objects as function objects. However, our system performs reasonably well at startup considering the benefits at runtime.

CHAPTER 7

Conclusions and Future Work

7.1 Future Work and Discussion

Given the promising results, presented in Evaluation, we think our approach is an important step towards more efficient scene graph development and prototyping. We expect our system architecture to integrate well with optimizing rendering backends and future trends towards stateless graphics programming interfaces. Our system, however has some limitations mitigating some of the benefits of the system:

Threading Our incremental computation framework is not well suited for multi-threaded applications. In experiments we tried to overcome these issues, but we think this topic requires more basic research in the field of incremental computation.

Lazy evaluation Our system's incremental evaluation framework heavily builds on *adaptive functional programming* for structural changes. We found, that lazy evaluation is vital for rendering performance in many situations like culling. However, it is not clear how to integrate efficient lazy evaluation with *adaptive functional programming*.

Attribute Grammar evaluation Our current implementation is naive, i.e. we perform no optimizations, although attribute grammars allow for a rich set of static analysis and optimization.

Order dependent rendering Render jobs are currently treated as an unordered set. In order to support order-dependent rendering (e.g. transparency), we plan to extend render job sets with facilitates for explicit ordering of render jobs. As implementation technique on could for example associate each render job with values for which a linear order can be computed easily.

7.2 Conclusions

In this thesis we provided an in-depth analysis of state-of-the-art scene graph systems and their limitations. We identified design goals and synthesized implementation techniques for solving important challenges in scene graph design. In our work we use the concept of attribute grammars for describing scene graph semantics in a clean and declarative manner. We developed an embedded domain specific language for authoring attribute grammars. In contrast to previous approaches, our solution provides high-level semantics while enabling efficient scene graph rendering. To this end we utilize adaptive functional programming, a general purpose framework for incremental evaluation. In order to support incremental rendering backends we developed mechanisms for providing the rendering backend with incremental changes of the system's state.

In our evaluation we compare traditional implementation techniques for scene graphs with our system based on attribute grammars and incremental evaluation. Our system achieves similar flexibility in a more concise way than previous approaches, without sacrificing performance. In fact the incremental design our system gives rise for a rich set of optimizations in the field of optimizing rendering backends, which we would like to explore in future work.

APPENDIX A

The Expression Problem

A.1 Extensibility, a tension in language design

When dealing with hierarchical data structures like scene graphs it is very likely to run into a central tension in language design: Extensibility in operations and data variants. First noted by Reynolds [40], the problem was later formulated by Wadler as the *The Expression Problem*:

> The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). [57]

Consider a simple example scene graph with nodes: `Square`, `Circle`. Both should be considered as `Shape` whereby for all shapes we would like to compute its area. In object-oriented programming style this functionality can be modeled like:

```
1  interface Shape
2  {
3    double computeArea();
4  }
5
6  class Square : Shape
7  {
8     double side;
9     Square(side) { this.side = side; }
10
11    computeArea() { return side * side; }
12 }
13
14 class Circle extens Shape
15 {
16    double radius;
17    Circle(radius) { this.radius = radius; }
```

```
18
19      computeArea() { return PI * radius * radius; }
20  }
```

In functional programming our example looks like:

```
1   type Shape = Square of double
2              | Circle of double
3
4   let computeArea (s : Shape) =
5     match s with
6           | Square a = a * a
7           | Circle r = r * r * pi
```

If we now add an additional variant of Shape, say `Rectangle` this is easy in the object-oriented solution and can be achieved by simply adding another class.

```
1   interface Shape
2   {
3     double computeArea();
4     double computePerimeter();
5   }
```

By contrast, in the functional implementation all functions operating on Shapes (particularly `computeArea`) need to be extended with the additional data variant.

```
1   type Shape = Square of double
2              | Circle of double
3              | Rectangle of double * double
4
5   let computeArea (s : Shape) =
6     match s with
7           | Square a = a * a
8           | Circle r = r * r * pi
9           | Rectangle (l,r) = l * r
```

Next, we extend the example with another function operating on `Shapes`. In the functional approach we simply add an additional function. In the object-oriented version however, all `Shape` implementations must be updated in order to properly implement the interface:

```
1   interface Shape
2   {
3       double computeArea();
4       double computePerimeter();
5   }
```

To sum up: Functional programming style makes it easy to add additional operations on data types, but it is hard to add additional data variants. To the contrary, object oriented modeling makes adding additional variants easy, but additional functions operating on data types hard. In scene graph design we actually run into this problem. Adding additional nodes should be easy for user code. Additionally adding operations like `GetBoundingBox()` or `ExtractGeometry()` should be easy as well. Unfortunately mainstream languages do not fulfil these requirements

without violating separate compilation, type safety or understandability.

Recently Oliveira et al. [31] presented a simple solution to the expression problem, which can be implemented in object-oriented languages supporting generics like Java or C#.

Bibliography

[1] Umut A. Acar. *Self-adjusting computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. AAI3166271.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 247–259, New York, NY, USA, 2002. ACM.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, November 2006.

[4] Umut A. Acar and Ruy Ley-Wild. Self-adjusting computation with delta ml. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[5] Lex Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Eindhoven Technical University, October 1993.

[6] Mark Barnes. Collada. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[7] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[8] Don Burns and Robert Osfield. Open Scene Graph A: Introduction, B: Examples and Applications. In *Proc. of the IEEE Virtual Reality 2004*, VR '04, pages 265–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[10] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 26–35, New York, NY, USA, 2002. ACM.

[11] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. *SIGPLAN Not.*, 35(10):130–145, October 2000.

[12] Crytek. http://www.crytek.com/cryengine/cryengine3/downloads. Accessed: 2014-15-01.

[13] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 105–116, New York, NY, USA, 1981. ACM.

[14] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM.

[15] Jim Durbin, Rich Gossweiler, and Randy Pausch. Amortizing 3d graphics optimization across multiple frames. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 13–19, New York, NY, USA, 1995. ACM.

[16] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *ECOOP*, pages 144–169, 2004.

[17] Torbjörn Ekman and Görel Hedin. The JastAdd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, December 2007.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] Görel Hedin. An overview of door attribute grammars. In *Proceedings of the 5th International Conference on Compiler Construction*, CC '94, pages 31–51, London, UK, UK, 1994. Springer-Verlag.

[20] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[21] R. Hoover. *Incremental graph evaluation (attribute grammar)*. PhD thesis, Cornell University, Ithaca, NY, USA, 1987. UMI Order No. GAX87-24200.

[22] Scott E. Hudson. Incremental attribute evaluation: a flexible algorithm for lazy update. *ACM Trans. Program. Lang. Syst.*, 13(3):315–341, July 1991.

[23] Gregory F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 141–151, New York, NY, USA, 1985. ACM.

[24] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, AT&T Bell Laboratories, 1979.

[25] Ken Kennedy and Scott K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 32–49, New York, NY, USA, 1976. ACM.

[26] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 1968.

[27] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[28] Microsoft. MSDN, C# online reference, 2012.

[29] Nordberg. Variations on the visitor pattern. In *PLoP '96*, 1996.

[30] NVIDIA Corporation. SceniX | NVIDIA Developer Zone, 2013. developer.nvidia.com/scenix [Online; accessed February 12, 2013].

[31] Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: practical extensibility with object algebras. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 2–27, Berlin, Heidelberg, 2012. Springer-Verlag.

[32] Tomas Petricek and Don Syme. The F# computation expression zoo. In Matthew Flatt and Hai-Feng Guo, editors, *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science*, pages 33–48. Springer International Publishing, 2014.

[33] Simon Peyton-Jones. *Haskell 98 language and libraries : the revised report*. Cambridge University Press, Cambridge U.K. ;;New York, 2003.

[34] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[35] Dirk Reiners. *OpenSG: A scene graph system for flexible and efficient realtime rendering for virtual and augmented reality applications*. PhD thesis, TU Darmstadt, 2002.

[36] Dirk Reiners, Gerrit Voss, and Johannes Behr. Opensg: Basic concepts. In *1. OpenSG Symposium*, 2002.

[37] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, July 1983.

[38] Thomas W. Reps and Tim Teitelbaum. The synthesizer generator. In William E. Riddle and Peter B. Henderson, editors, *Software Development Environments (SDE)*, pages 42–48. ACM, 1984.

[39] Thomas W. Reps and Tim Teitelbaum. *The synthesizer generator: a system for constructing language-based editors*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[40] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to type abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. IRIA, 1975.

[41] John Rohlf and James Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 381–394, New York, NY, USA, 1994. ACM.

[42] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. *Electron. Notes Theor. Comput. Sci.*, 253(7):205–219, September 2010.

[43] Emma Söderberg and Görel Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report LU-CS-TR:2012-249, report number 98, Computer Science, Faculty of Engineering , Lund University, 2012.

[44] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[45] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 341–349, New York, NY, USA, 1992. ACM.

[46] R.S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *Proceedings 18th Symposium on Principles of Programming Languages*, pages 1–13. ACM, January 1991.

[47] Doaitse S. Swierstra, Pablo, and Joao Sariava. Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[48] Wouter Swierstra. Why Attribute Grammars Matter. *The Monad.Reader*, 4, July 2005.

[49] Don Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 43–54, New York, NY, USA, 2006. ACM.

[50] Don Syme. The F# 3.0 Language Specification, 2012.

[51] Robert F. Tobler. Separating semantics from rendering: a scene graph based architecture for graphics applications. *Vis. Comput.*, 27(6-8):687–695, June 2011.

[52] Steve Upstill. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[53] Marcos Viera, Doaitse Swierstra, and Arie Middelkoop. Uuag meets aspectag: how to make attribute grammars first-class. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 6:1–6:8, New York, NY, USA, 2012. ACM.

[54] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 245–256, New York, NY, USA, 2009. ACM.

[55] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *SIGPLAN Not.*, 24(7):131–145, June 1989.

[56] G. Voß, J. Behr, D. Reiners, and M. Roth. A multi-thread safe foundation for scene graphs and its extension to clusters. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[57] P Wadler. Email, Discussion on the Java Genericity mailing list, November 1998.

[58] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[59] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993.

[60] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[61] Michael Wörister. A caching system for a dependency-aware scene graph. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, December 2012.

[62] Michael Wörister, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. Lazy incremental computation for efficient scene graph rendering. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 53–62, New York, NY, USA, 2013. ACM.