FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Caching System for a Dependency-Aware Scene Graph

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Wörister
Matrikelnummer 0402917

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 12.12.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Caching System for a Dependency-Aware Scene Graph

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Wörister
Registration Number 0402917

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 Tobler

Vienna, 12.12.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Wörister
Hippgasse 38/19, 1160 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)

Abstract

Scene graphs are a common way of representing 3-dimensional scenes for graphical applications. A scene is represented as a hierarchical structure of nodes which represent 3D geometry, spatial transformations, surface properties, and other—possibly application specific—aspects. Scene graph systems can be designed to be very generic and flexible, e.g. by allowing users to implement custom node types and traversals or by providing facilities to dynamically create subgraphs during a traversal. This flexibility comes at the cost of increased time spent in pure traversal logic. Especially for CPU-bound applications this causes a performance drop.

This thesis proposes a *scene graph caching system* that automatically creates an alternative representation of selected subgraphs. This alternative representation poses a *render cache* in the form of a so-called *instruction stream* which allows to render the cached subgraph at lower CPU cost and thus more quickly than with a regular render traversal. Additionally, a number of optimizations for *render caches* were implemented to further increase the performance gain with respect to uncached rendering.

In order to be able to update *render caches* incrementally in reaction to certain scene graph changes, a *dependency system* was developed. This system provides a model for describing and tracking changes in the scene graph and enables the *scene graph caching system* to update only those parts of the *render cache* that needs to be updated without necessitating a full rebuild of the cache.

The actual performance characteristics of the *scene graph caching system* were investigated using a number of synthetic test scenes in different configurations. These tests showed that the caching system is most useful in scenes with a high structural complexity (high geometry count and/or deep scene graph hierarchies) and moderate primitive count per geometry. In this kind of scene the *scene graph caching system*, with all optimizations enabled, reduced average frame times by a factor of 5 to 8 with all objects in the scene changing their transformation each frame. This performance gain could be achieved at the cost of startup times increased by 3 to 4 seconds for scenes with 3000 to 8000 *geometry nodes*. The additional main memory consumption was measured at 4 MiB for the scene with 3000 geometries and a flat transformation hierarchy and 20 MiB for the scene with 8000 geometries and a deep transformation hierarchy.

Kurzfassung

In grafischen Anwendungen werden 3-dimensionale Szenen häufig als sogenannte *Szenengraphen* dargestellt. Die Knoten in einem solchen Szenengraphen repräsentieren dabei geometrische Formen, deren räumliche Transformationen, Oberflächeneigenschaften und andere, möglicherweise anwendungsspezifische, Aspekte. Szenengraphensysteme können sehr flexibel und erweiterbar gestaltet werden, indem z.B. die Definition eigener Knoten- und Traversierungstypen erlaubt wird. Dieses Flexibilität bringt aber meist aufwendigere Laufzeitlogik mit sich, was besonders in CPU-beschränkten Applikationen zu einer niedrigeren Ausführungsgeschwindigkeit führt.

Um dieses Problem zu verringern, schlägt diese Arbeit ein *Caching System* für Szenengraphen vor. Dieses System legt automatisch sogenannte *render caches* für ausgewählte Subgraphen an. Ein *render cache* enthält dabei einen sogenannten *instruction stream*, der es erlaubt den gecachten Subgraph effizienter zu rendern als dies mit einer herkömmlichen Traversierung der Fall ist. Zusätzlich wurde eine Reihe von Optimierungen für *render caches* implementiert, die eine weitere Leistungssteigerung ermöglichen.

Damit *render caches* nicht bei jeder Szenengraphänderung komplett neu erstellt werden müssen, wurde ein sogenanntes *dependency system* entwickelt. Dieses System erlaubt es, bestimmte Änderungen in der Szene zu beschreiben und automatisiert darauf zu reagieren. Als Folge können *render caches* inkrementell auf dem neuesten Stand gehalten werden, was eine wesentliche Effizienzsteigerung bedeutet.

Die tatsächlichen Leistungscharakteristika des *Caching Systems* wurden in einer Reihe synthetischer Testszenen in verschiedenen Konfigurationen getestet. Diese Tests haben gezeigt, dass das *Caching System* in Szenen mit hoher struktureller Komplexität (d.h. hohe Anzahl an Geometrieknoten und/oder tiefer Szenengraphhierarchie) und moderater Anzahl von Dreiecken pro Geometrie den größten Nutzen hat. In dieser Art von Szene führte das System (mit allen Optimierung aktiviert) zur einer Reduktion der durchschnittlichen *frame time* um einen Faktor von 5 bis 8. Die Testszenen waren dabei voll dynamisch, d. h. alle Objekte änderten ihre räumliche Transformation jeden *frame*. Diese Leistungssteigerungen wurden zum Preis erhöhter Ladezeiten und erhöhten Speicherverbrauchs erreicht. Die Ladezeiten erhöhten sich um 3 bis 4 Sekunden für Szenen mit 3000 und 8000 Geometrieknoten. Der zusätzliche Hauptspeicherverbrauch lag bei 4 MiB für die Szene mit 3000 Geometrien und einer flachen Transformationshierarchie, und bei 20 MiB für die Szene mit 8000 Geometrien und einer tiefen Transformationshierarchie.

Contents

CHAPTER 1

Introduction

Scene graphs are a common way of modeling 3-dimensional scenes for graphical applications. There are many existing toolkits like Performer [40], OpenSceneGraph [8], OpenSG [44], or SceniX [17]. A scene graph is a directed acyclic graph (DAG) where nodes represent geometric models and their properties. Geometry nodes are typically located at the leaves of the graph. The internal nodes of the graph describe a number of different *attributes* like spatial transformation or surface properties. These attributes are inherited along the edges of the graph until they reach a geometry node. This way a geometry node is assigned a set of attributes which are used to display it on the screen (see Figure 1.1).

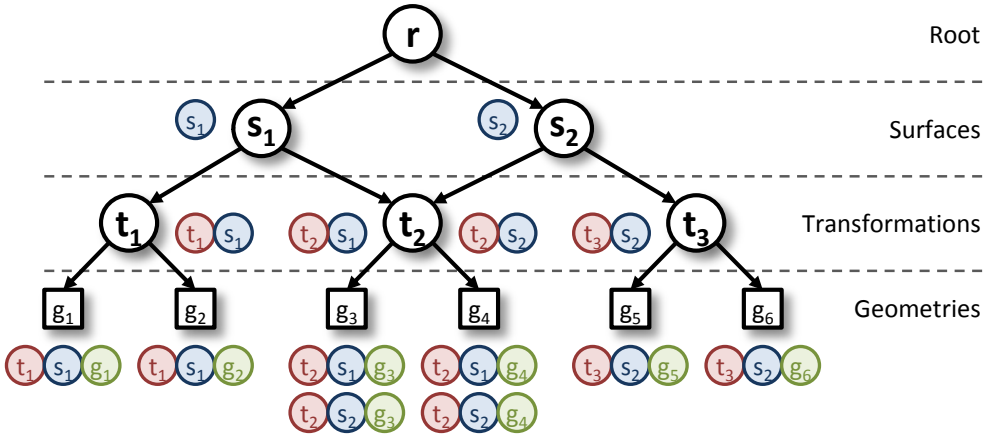


Figure 1.1: A scene graph with attribute annotations.

A scene graph can be rendered by traversing it depth-first, starting at the root. Whenever an attribute node is encountered, the attribute it publishes is stored in the current traversal state. Some attributes—like surface properties—replace the existing attribute value in the traversal

state, while others—like transformation matrices—are combined with the existing value to form a new one. When the traversal reaches a geometry node, the currently active traversal state represents the attributes of the geometry. At this point the geometry can be drawn with the appropriate settings.

Scene graph systems can be designed to allow for great flexibility. Users might be able to define their own traversals for collecting data or mutating the graph. They might also be able to create new node types that describe application-specific attributes. It is even possible to allow for dynamically creating entire subgraphs during traversals. These possibilities strive to enhance the expressiveness and usability of scene graphs as a modeling facility.

Unfortunately, as is often the case, increased expressiveness and ease of use are accompanied by increased execution complexity. For flexible, easily extensible systems—like the AARDVARK rendering framework [42] which provides the implementation context of this work—pure traversal logic will often make applications CPU-bound.

1.1 Aim of this Work

The *scene graph caching* system presented in this thesis tries to reduce the time spent purely on traversing the scene graph and accumulation attributes. A detailed description of goals and scope of the thesis is given in Section 3.1—in short, however, the aim is to create a system that automatically caches an optimized representation of the scene graph (or parts thereof) which will then allow for more efficient rendering. At the same time the system is intended to be non-intrusive and should not require tedious configuration work from the user.

1.2 Methodological Approach

In order to test the proposed *scene graph caching* system, a prototypical implementation is created. This includes the basic caching system mechanism and a host of additional, optional optimizations. With the prototype in place, a number of performance tests using synthetic scenes are conducted. Each test is designed to contrast several configurations against each other: with and without caching or with different sets of optimizations enabled. The resulting difference in the time needed to render the scene is analyzed and discussed. Where appropriate, memory consumption and startup times are also investigated. The synthetic test cases are parameterized to represent different classes of scenes in order to find the cases the caching system is useful in and the ones where it is not. The results of these performance tests are listed in Chapter 4.

1.3 Structure of the Thesis

The thesis is structured as follows:

- Chapter 2 investigates the state of the art on two related areas: *scene graph optimization* and *incremental computation*.
- Chapter 3 constitutes the main part of this thesis and composes itself listed below:
  - At first the *Goals and Scope* of the *scene graph caching system* presented in this work are defined.
  - The *Implementation Context* of the caching system prototype is laid out: What are the lower-level *APIs* the prototype is built against? What does the scene graph toolkit look like that the caching system is embedded in?
  - Next, the fundamental principles and architecture of the caching system are presented in *Caching Architecture*.
  - The so-called *Dependency System* is proposed which—through meta-data annotations—allows to selectively track changes within a scene graph. This section not only introduces the basic concepts of the dependency system but also examines some important properties which can be inferred from these basic concepts.
  - The section *Dependency-Aware Scene Graph Caching* shows how the previously presented dependency system can be used to update a *render cache* incrementally for a certain class of scene graph changes.
  - The next section demonstrates a host of *Optimizations* that can be applied within the caching system.
  - Towards the end of the main part, the two most common types of *render caches* are described in *Default Cache Types*.
  - Finally, the system presented so far is discussed with respect to the initial goals and related work.
- Following the main part, Chapter 4 investigates how the caching system fares in a number of synthetic test cases.
- Chapter 5 brings the thesis to its conclusion and points out possible future work.

CHAPTER 2

Related Work

The related work section can be divided into two different fields. First, the *scene graph caching system* presented in this work is a form of *scene graph optimization*. Second, the caching system can be seen as a special case of *incremental computation*.

2.1 Scene Graph Optimization

Optimizations for scene graphs are very common and have different levels of support in most scene graph toolkits. There are two different approaches to optimizing a scene graph: (1) applying persistent transformations to the scene graph (commonly before runtime), and (2) keeping an additional, optimized runtime representation of the scene graph. Note that these approaches do not necessarily preclude each other. The following sections will examine both cases in more detail and will provide some examples of specific optimization transformations. The next section goes on to discuss existing scene graph toolkits and their use of the described optimizations. In the concluding section, it will be discussed how all of the above relates to the *scene graph caching system* presented in this work.

2.1.1 Persistent Transformations

The defining characteristic of this approach is that the optimization process is applied only once and the outcome of the transformation completely replaces the input scene graph. As is common for optimizations in general, the transformed scene graph should comply to some criterion of equivalence or at least approximate equivalence. In the case of graphical applications an obvious criterion can be that the pictures resulting from rendering both the original and the optimized scene graphs are sufficiently similar to each other. What is deemed as sufficient will depend on the specific application the scene graph is used in. Non-graphical applications such as physical simulations might have stricter constraints on optimization transformations.

The optimization process can have different goals. Typical examples are: reducing the time needed to render the scene, the reduction of traversal cost, or a lower memory footprint. Which

of these goals are desirable will also depend on the specific application and technical context, such as the hardware and software platforms the application is targeted at. The goals can also be conflicting (as, for example, is often the case for rendering time and memory footprint).

As the transformation is only applied once, the computational effort needed to perform it is of less importance than in a runtime or interactive scenario. Depending on the information needed by the optimization process, transformations can be applied early in the content creation process, at load time, or on demand at runtime. Some transformations might benefit from information about the actual execution platform[1] and are therefore better performed at load or runtime. Others might always yield the same result and could already be applied by the exporter of a modeling program or an offline step in a build process, with the benefit of sparing the execution environment the computational workload.

Transformation Examples

The following section will show some examples of optimizing scene graph transformations. Strauss [34] and Boudier [35] provide the main sources for the sections below but implementations of these examples can be found in many of the available scene graph toolkits, as will be shown later in section 2.1.3. Strauss [34] describes three common transformations:

Pull up costly state changes. Changing the state of the rendering pipeline, such as changing the currently active vertex or fragment program, can be a costly operation:

> A change in state may have an adverse impact on the performance throughput of a graphics system. For example, such a change may require transferring the new state information to step 230, and then reconfiguring (for example, resetting a processing pipeline) the apparatus performing step 230. Either of such steps may impede the throughput performance of a graphics system, and such state changes may therefore be undesirable. [34, p. 11]

In order to decrease the number of state changes, this transformation rearranges the scene graph. Due to the nature of scene graphs, descendant nodes inherit attributes introduced by their ancestor nodes. Because of this, it is possible to pull up and merge nodes that apply the same state and consequently it is often possible for a given scene graph to find a semantically equivalent graph that requires fewer state changes while traversing it. Figure 2.1 shows an example of such a transformation. Traversing the original scene graph on the left would require setting state A to a different value four times and state B two times. Assuming that changing state A is more costly than changing state B, the transformed scene graph on the right allows for a cheaper traversal: state A is only changed three times while B is changed two times.

Push transformations into vertices. In a scene graph spatial transformations such as translation, rotation, or scaling can be represented as nodes. The transformation specified in such a transformation node is applied to every descendant node. Also, more than one transformation node is allowed to appear on a traversal path. In this case, transformations are applied one after

[1] For an example see Boudier's description of the *Promote Attribute* optimization. [35, pp. 20-21]

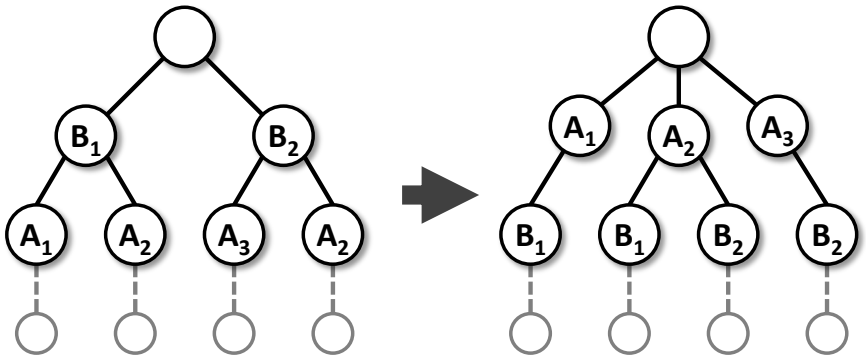


Figure 2.1: Transforming a scene graph to reduce state changes

the other in the order they are encountered on the path. Spatial transformation is thus a special kind of attribute that is inherited down the scene graph with an aggregation-semantic instead of a replacement-semantic.

However, applying a transformation through such a node incurs a computational overhead. Typically, currently active transformations are kept as a stack of transformation matrices and every time a transformation node is encountered the top-most matrix on the stack is multiplied with the matrix stored in the node and pushed onto the stack. These matrix multiplications have to be performed anew every time the scene graph is traversed and thus cause enduring runtime overhead.

One way to alleviate this problem is to pre-compute the transformations for every vertex contained in the geometry leaves of a subgraph and remove the now redundant transformation nodes from the scene.[2] Consequently, the transformations do not have to be computed every frame. Figure 2.2 shows a simple example where it is possible to remove three transformation nodes.

Yet, there are also detriments. First, this optimization is only applicable if none of the transformations can change dynamically, for example as reaction to user input or as a function of time. Second, the vertices of a geometry leaf cannot store more than one (transformed) position or normal. If a geometry leaf is reachable via more than one path containing different transformations, the vertex data has to be copied to account for the different global positions the geometry adopts for each path. Naturally this comes at the cost of additional memory consumption. Whether this is a worthwhile tradeoff has to be decided by the user.

Convert Geometry to Triangle Strips. Strauss also suggests to convert vertex-based geometry into triangle strips. The triangle strip is one kind of «primitive topology» supported by graphics APIs such as *Direct3D* and *OpenGL*. A description and list of primitive topologies supported in *Direct3D 11* are available in the respective documentation. [11] Triangle strips are an efficient

[2] Note that simply reducing the number of transformation nodes by combining them, will also already reduce the number of necessary matrix multiplications.

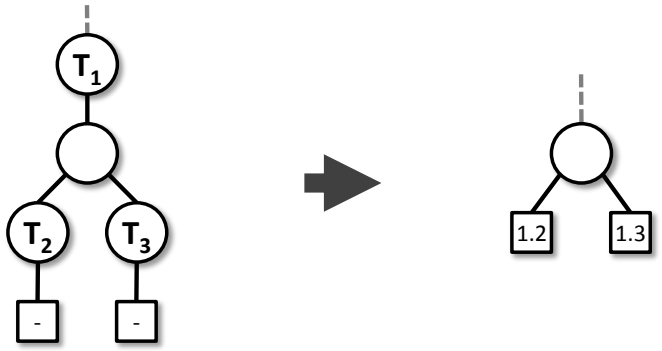


Figure 2.2: Pushing transformations down into vertex coordinates

way of representing vertex-based geometry. After the first triangle every following triangle is specified by adding just one additional vertex without the need for index data. This saves memory space and bandwidth and—because of the structures inherent data locality—allows for good cache utilization on the graphics hardware. For an extensive overview on triangle strip algorithms see Vanček and Kolingerová, 2007. [43]

Vertex data can also be converted into other optimized representations apart from triangle strips. For example, Nehab et al. [32] and Sander et al. [41] describe methods for reordering index data. Vertex indices are reordered in a way to «improve post-transform vertex cache efficiency as well as for view-independent overdraw reduction» [41, p. 1].

Boudier [35] presents a generic optimization system that allows the user to define how a set of prioritized and parameterized *atomic optimizations* is applied to some input scene graph. An *optimization manager* uses a database of *atomic optimizations* indexed in an *optimization registry* containing meta data about each algorithm, such as input parameters or priority information which models constraints on the application order of the algorithms. The manager can be configured by the user through configuration files, a user interface, or by some other means—depending on the actual implementation. The configuration supplies the manager with a set of optimizations to apply and optional input parameters.

The system is meant to be used as a post-processing step while or after exporting the scene graph from a modeling program. [35, p. 18] However, the general principle could also be used at load time in a runtime context.

Boudier also gives an exemplary list of possible *atomic optimizations*. Some of these are better described as validity checks or transformations to adapt the scene data for a specific execution platform. These include the normalization of texture coordinates and alpha values, and converting image file formats.

Also described are optimizations that have already been discussed above, such as pre-computing spatial transformations and storing the result directly in vertex coordinates (here as part of *Collapse Hierarchy*), and converting geometry data into triangle strips. Finally, the *Promote At-*

tributes transformation can be seen as a variation of *pull up costly state changes* described above (p. 6).

However, Boudier's list also includes many transformations that have not been mentioned yet. These transformations provide good examples of typical changes that can be made to a scene graph in order to improve it with respect to some criterion. Below, they are described as part of two broader, recurrent categories:

Removing Redundancies. One important category of scene graph optimizations works by removing redundant nodes or edges from the graph while keeping the graph semantically equivalent. This generally reduces memory consumption (fewer nodes have to be stored), traversal cost (fewer nodes and edges have to be visited), and—in some cases—drawing time (if redundant API calls are implicitly omitted). [3]

Flatten Hierarchy is a simple representative of this category: Scenes and models created by people—or sometimes also procedurally—may contain hierarchies of group nodes that are an artifact of some organizational principle in the modeled object which helps the artist in handling her creation. Yet, the hierarchy may not be needed in this verbose form and can be replaced with an equivalent but more compact substitute (see Figure 2.3).

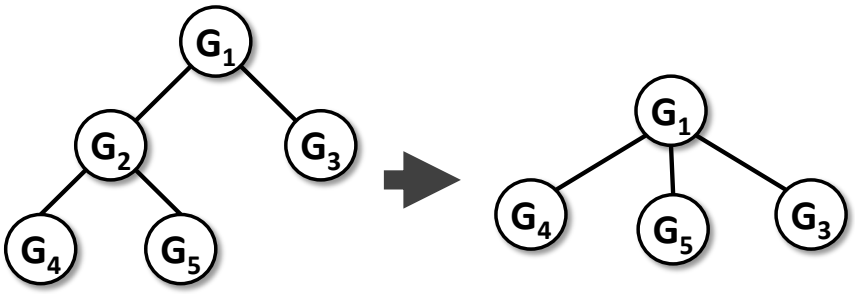


Figure 2.3: *Flatten Hierarchy* optimization

Collapse Hierarchy is very similar. Boudier mentions two examples: First, pushing a transformation directly into child geometries and subsequently removing the transformation node. Second, removing a group node that only has a single child, as it does not add any additional information to the scene graph.

Another optimization in this category is *Remove Attributes*. According to Boudier, «Many modelers and exporters put attributes in an exported scene graph by default, so as to reflect as closely as possible the data in the modeler» [35, p. 21]. As not all of this information may be needed (depending on the application), the optimizer may be able to remove certain attributes without loss of critical data.

[3] If the application is traversal-cost-bound then the overall drawing time also is improved if traversal cost is reduced. Often this is the more pronounced effect, as will be shown later.

The next *atomic optimization* dealing with redundancies is *Share Attributes*. It is a common case that different attribute node instances represent the same state, such as applying the same material to different sub-graphs. The object representing the state may be costly to store and storing the same value in many different instances is a waste of memory. This transformation makes sure that any attributes having the same state also share the same state-object instance. This saves memory and makes it possible to compare state equality by just comparing pointers instead of actual values, which might allow for more efficient implementation of runtime system routines. See figure 2.4 for an illustration.

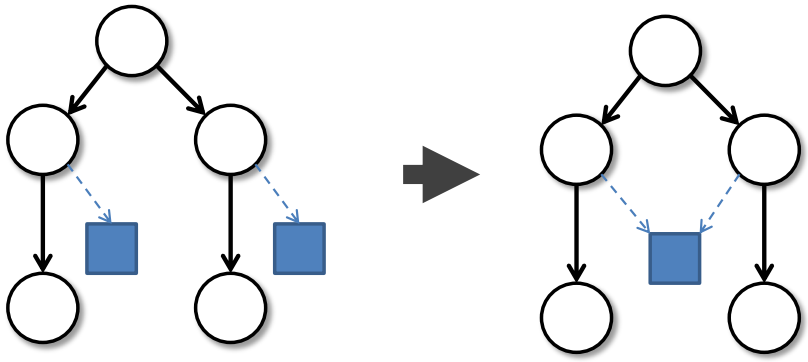


Figure 2.4: *Share Attributes* optimization

Finally, there are *Generate Macro Texture* and *Collapse Geometry*. Both follow the principle of combining multiple items of some type into one larger item of the same type. *Generate Macro Texture* will create a texture atlas from various source textures. The texture coordinates of objects that have used one of the source texture are adjusted to the new texture space of the atlas. As a result there may be fewer state changes (because there are fewer textures to be activated), but this optimization can also increase the number of cases where *Collapse Geometry* can be applied.

Collapse Geometry combines multiple geometry nodes having the same state into one: «In particular, the input scene graph is traversed and, for each node, a determination is made as to whether the subtree consists of geometry. If so, the geometries of the subtree are combined.» [35, p. 19] The result is (slightly) reduced memory usage and fewer state changes. Figure 2.5 shows an example of *Generate Macro Texture* and *Collapse Geometry* applied at once. Both transformations discard redundant structural information, so to speak.

Computing Optimization Data Structures. The *atomic optimizations* in the second category do not remove but add data to the scene graph. This additional data is used to speed up certain runtime tasks. For example, *Create Bounding Boxes* will, as the name says, compute bounding boxes for scene graph nodes. These bounding boxes can be used at runtime in order to perform view frustrum culling.

The second optimization in this category is *Spatial Partition* which again improves frustrum culling performance. There are many spatial partitioning schemes that can be used to hierar-

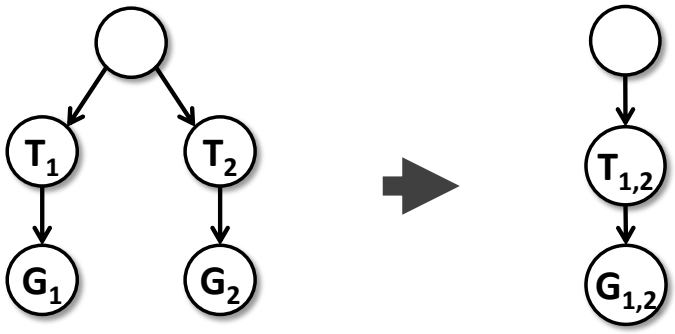


Figure 2.5: *Collapse Geometry* and *Generate Macro Texture* optimizations

chically subdivide or index the scene graph. Which is most suited depends on the given scene graph and has to be specified by the user.

In a way, also *Generate Macro Texture* and *Collapse Geometry* can be seen as belonging to this category, as both create additional, more optimal data structures that help to enhance runtime performance. The difference is that source data is not retained by these algorithms and thus no additional persistent memory is needed.

The validation and adaption transformations briefly mentioned above constitute a third category. However, as they do not really optimize the scene and are of little interest in the context of this work, they will not be described in any more detail.

2.1.2 Alternate Runtime Representation

Another way of implementing scene graph optimizations is to keep an optimized runtime representation of some subgraph in parallel to the original scene graph. At the cost of additional storage space, this alternate representation is tuned to support some special task more efficiently. Mostly this task is rendering but there are also other cases with different requirements such as collision detection. As opposed to the persistent approach described before, the optimized scene graph representation needs to be kept in sync with the source scene graph it represents—even if the source data undergoes changes during runtime. As will be shown, the additional representation can be another scene graph—elaborated on below in the section «Dual Scene Graphs»—or it can be something else completely, as presented in «Scene Graph Acceleration Structures».

Dual Scene Graphs

Hopcroft et al. present a system that keeps two (or more) representations of the same scene graph. [31] The first of these representations is called the *user scene graph* (USG). It acts as the source or blueprint for the second scene graph which is called the *rendering scene graph* (RSG). The USG is meant to be created and manipulated by a human and its purpose is to be

easily understandable and editable with the intended user not having to consider how scene graph properties—such as logical grouping, or spatial distribution—may affect rendering performance. The system will, automatically and transparently to the user, create the *rendering scene graph* in the background, the purpose of which is already given by its name: The RSG is transformed and optimized to be better suited for being consumed by the rendering hardware. This is achieved by doing a number of things:

- The scene's *geometry granularity* is adjusted by merging or splitting geometry nodes. This is done in order to reduce the amount of time spent for preparing geometry nodes for rendering (when merging many small nodes into one large node); or to increase culling possibilities (when splitting a large node into smaller ones). Merging geometry has already been discussed before as *Collapse Geometry* (p. 10).
- The *spatial organization* of the scene graph may be changed to improve culling performance. This is done by splitting and regrouping geometries, similar to the *Spatial Partition* optimization mentioned earlier (see *Computing Optimization Data Structures*, p. 10).
- Once again, the graph may be restructured to *reduce* the number of necessary *state changes*. This seems to be achieved in a similar fashion as described by Strauss (see *Pull up costly state changes*, p. 6).
- *Level-of-detail* nodes are created to improve rendering performance while maintaining an approximate level of output image quality.
- *Curves* and *curved surfaces* can be presented and edited as such in the USG, while they are *automatically tessellated* into triangles for rendering in the RSG.

As already mentioned above, the RSG is kept in parallel to the USG. It also is incrementally updated whenever a change is made to the USG. For example, when a node is simply moved in the USG, it may have to be removed in the RSG and then re-inserted at a completely different place in the hierarchy to account for the different spatial organization of the RSG. Unfortunately, Hopcroft et al. do not go into much detail on how these incremental updates are performed. This would be interesting, especially in the light of having to incrementally apply a modification to the already transformed RSG.

Scene Graph Acceleration Structures

The parallel, optimized representation of the scene graph does not always have to be another scene graph. Typically, a graph structure—especially as created by humans—and rendering APIs such as *OpenGL* or *Direct3D* are not a good match for each other. As a result, it can be beneficial to introduce new kinds of data structures that additionally keep the scene in a form more suited for rendering or some other purpose. A few possible approaches will be described in the following.

Sowizral et al. propose a system that generates and manages a number of ancillary structures which aid in updating and rendering the scene graph. Each structure is responsible for a

certain aspect of the scene data and its processing; they do not mirror the scene graph hierarchy directly but represent a vertical subdivision of responsibilities. One example given is the *geometry structure* which maintains a bounding-box hierarchy of renderable objects used for querying and culling. Analogously the *rendering-environment structure* keeps the state of all spatially located but non-renderable objects (lights, fog, etc.) and provides for querying those. *Render bin structures* manage a set of currently visible objects. They may sort the objects before rendering or apply local optimizations on them, such as combining or splitting geometry.

This parallel representation can take on the original scene graph's functions in varying degrees:

> This may include (...) the use of ancillary structures to reduce the extent of traversal of the original structure, replacement of the original structure by one or more ancillary structures thereby reducing references to the original structure (scene graph), or even the complete replacement of the original structure by new structures containing sufficient information to render and execute the associated scene-graph-based program (...) [25, p. 7]

The initial parallel structure is created from the scene graph when the scene is loaded but can be updated incrementally during runtime. Each of the specialized managers is updated concurrently by its own update thread. Appropriate to this concurrent processing model, communication and change propagation is accomplished via a message passing scheme. Each message receives a logical time-stamp to permit a consistent chronological ordering of these messages and the changes they represent. Concurrency, however, is limited by logical dependencies between the tasks the various structures perform. In an example given by Sowizral et al. the *Transform-Structure* has to finish updating before any other subsystem as most of them rely on consistent object positions and orientations. The resulting necessity of proper update scheduling is provided by the so-called *MasterControl* thread which coordinates the ancillary structures and is also in charge of time-stamping all messages in the system.

The system presented by Durbin et al. [18] shows the most similarities to the scene graph caching system this thesis is about.

Viewed from the perspective of a rendering API, such as *OpenGL*, the scene graph is just the sequence of API function calls which are generated by performing a render traversal. The hierarchical graph structure is involved in producing these calls, but it is not used in any way by the rendering API itself. Consequently, to render a scene, the scene graph would not even be needed if the sequence of API calls for rendering the scene was available. Of course, the scene graph is useful in other ways, such as when creating or manipulating the scene; but for drawing the scene, anything that can produce the desired sequence of API calls (parameterized with the same data) is sufficient. This is what the approach of Durbin et al. takes advantage of.

In their system, for every node in the scene graph a flat list of graphics calls is generated and stored. This list, called the *streamlined array* of the node, is a persistent representation of the API calls that would be issued by a render traversal when encountering the node. This *streamlined array* represents a cache for the node that, when executed, reproduces the API calls and, accordingly, the output image of the object the node models.

Keeping these small caches around has a couple of benefits:

- Caches can be constructed in a hierarchical manner, so that the cache of a parent node is equivalent to the combination of the caches of all its children, and by extension, of all its descendants (Figure 2.6). In other words: Executing the *streamlined array* of the parent produces the same results as rendering the whole subgraph and can thus completely replace a render traversal of the parent node and everything below.

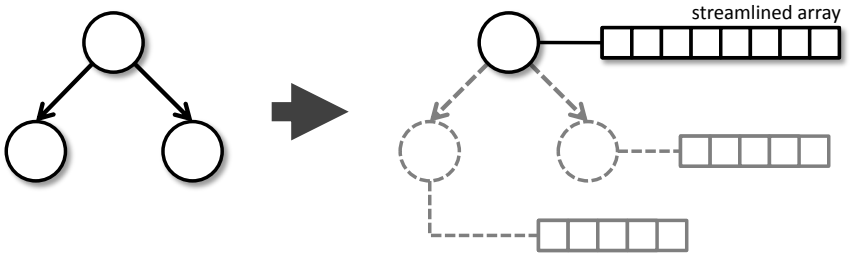


Figure 2.6: «Streamlined arrays» in a scene graph hierarchy

- A linear list of graphics commands can easily be optimized in a couple of ways. First, the final result of multiplying any transformation matrices can be stored directly with the command and hence does not have to be recalculated for every call. This optimization can be compared to *Push transformations into vertices* (p. 6). Second, redundant calls for setting render state can be filtered out from the sequence (Figure 2.7). The result is equivalent to keeping track of the current render state in the render traversal, and conditionally making a state setting call only if the state would actually change. The difference between the former and the latter is that the condition only has to be evaluated once during the optimization step instead of at every API call. [4] Durbin et al. use the term «peephole optimizations» [18, p. 13] in reference to the small, locally bounded optimizations done by many language compilers.

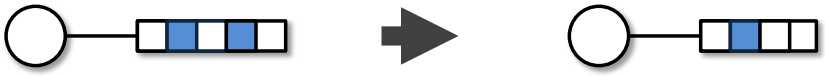


Figure 2.7: «Peephole optimization» of a *streamlined array*

- Executing the flat, linear *streamlined array* of a whole subgraph is much cheaper than performing a full-blown render traversal. From the perspective of memory access locality,

[4] Note that removing these redundant state setting calls is not the same as reducing state changes, as is done, for example, by *pulling up costly state changes* (p. 6). Rather, these two optimizations complement each other—the latter providing the former with best-case input data. This will be shown in more detail in Section 3.6 of this work.

call stack depth, and especially effort needed to prepare the API call and its parameters, the precomputed array of commands is pretty much as good as it gets.

However, this way of caching graphics commands has some implications that need to be taken care of. Since the *streamlined array* always only contains a snapshot of the current state of a certain node, the caches need to be invalidated when changes occur. In this case, the system marks the affected caches as dirty and can re-build them at a later point in time. As a consequence of the hierarchical cache dependencies, a change in some node X not only invalidates the cache of X but also the caches of all ancestor nodes of X.

Building the caches and optimizing them can be expensive while at the same time a cache may be invalidated after just one usage. To remedy this, Durbin et al. let their optimizer estimate whether creating a cache is a good investment of computation resources.

2.1.3 Scene Graph Toolkits

Many available scene graph toolkits support a subset of the optimizations mentioned above:

IRIS Performer [40] was one of the first scene graph toolkits with a strong focus on rendering performance. It natively supports some of the persistent transformations mentioned above, as well as a form of parallel scene graph representation as a means of aiding in multiprocessing.

- Static transformations (represented by *pfSCS* nodes) can be pushed down into vertex coordinates. This one-time transformation of a scene graph is implemented by the *pfFlatten()* routine [40, Section 3.1.4] and is an instance of the *Push transformations into vertices* optimization (p. 6).
- The *pfPartition* node creates an internal spatial index of the underlying subgraph at load-time. This index can then be used to speed up geometry intersection testing at run-time. [40, Section 3.1.4] This is an example for an *Acceleration Data Structure* optimization (p. 10).
- *Performer* uses a pipelined multiprocessing architecture where different stages (application logic, intersection queries, culling traversal, drawing) can be performed in different processes. To facilitate data synchronization between processes in a consistent and efficient manner, each stage (except drawing) has its own view of the scene graph, called the *pfBuffer* of the stage. Changes to the scene graph made by the application logic stage are propagated down to subsequent stages in a *frame-accurate* manner, meaning that the state of the scene graph in a specific frame is consistent across all stages. [40, Section 3.2.3] These multiple views on the scene graph can be considered a form of *Alternate Runtime Representations* (p. 11).
- The drawing stage of *Performer's* rendering pipeline does not access the scene graph directly. Instead, the culling stage produces a *pfDispList* that is consumed and executed by the drawing stage. This *pfDispList* is a sequence of commands similar to a *streamlined array* in the system presented by Durbin et al. (p. 13), however, it is not stored persistently but regenerated for every frame. It only contains commands for drawing objects which

are currently visible and can optionally be sorted to reduce unnecessary state changes. This can be seen as a runtime version of *pull up costly state changes* (p. 6). Both use a flattened representation of the scene graph, a list of fully defined render jobs, to determine a good execution order, with *Performer* then directly executing the list and *pull up costly state changes* using it to create a transformed scene graph.

OpenSceneGraph [8] supports many persistent scene graph transformations through its *osgUtil::Optimizer* class. [33] Table 2.1 shows a selection of possible optimization flags defined in the *osgUtil::Optimizer::OptimizationOptions* enumeration together with the corresponding optimization described in this document.

REMOVE_REDUNDANT_NODES	*Collapse Hierarchy* and *Flatten Hierarchy* (p. 9)
MERGE_GEOMETRY	*Collapse Geometry* (p. 10)
TRISTRIP_GEOMETRY	*Converting geometry to triangle strips* (p. 7)
SHARE_DUPLICATE_STATE	*Share Attributes* (p. 9)
FLATTEN_STATIC_TRANSFORMS_ DUPLICATING_SHARED_SUBGRAPHS	*Push transformations into vertices* (p. 6)
FLATTEN_STATIC_TRANSFORMS	Same as above but without subgraph duplication
SPATIALIZE_GROUPS	*Spatial Partition* (p. 10) using a quadtree or octree
TEXTURE_ATLAS_BUILDER	*Generate Macro Texture* (p. 10)

Table 2.1: *osgUtil::Optimizer::OptimizationOptions* (selection)

In addition to these one-time optimizations, *OpenSceneGraph* also will, prior to rendering, sort render jobs pertaining to opaque geometry to minimize unnecessary state changes.

OpenSG The main focus of *OpenSG* [44] lies on providing a flexible data management scheme for concurrency. Nonetheless, it supports many of the same one-time optimizations as *OpenSceneGraph*. The scene graph can be persistently manipulated via so-called *GraphOps* which can «traverse the scene graph and do optimization operations on the nodes» [38]. Among these *GraphOps* are implementations of *Push Transformations into Vertices*, *Convert Geometry to Triangle Strips*, *Share Attributes*, and *Collapse Geometry*. Also, like *OpenSceneGraph* and *Performer*, *OpenSG* will sort render jobs to reduce state changes. [38]

NVIDIA Scenix provides *OptimizeTraversers* [17] which can transform a scene graph in several ways:

- The *CombineTraverser* performs transformations similar to *Collapse Geometry* (p. 10)
- The *EliminateTraverser* removes a number of redundancies. It includes *Collapse Hierarchy*, *Flatten Hierarchy* (p. 9), and some actions specific to *SceniX*.

- In *SceniX* transformation nodes can have multiple children. If a transformation node only contains an identity matrix, the *IdentityToGroupTraverser* can replace it with a simple group node.
- The *UnifyTraverser* applies a very generic version of the *Share Attributes* (p. 9) optimization to the scene graph.

2.1.4 Relation to this work

As already mentioned, the scene graph caching system presented in this thesis has many similarities to the system described by Durbin et al. ([18]), and as such is an *Alternate Runtime Representation* (Section 2.1.2). The cornerstone of the caching system is the representation of selected scene graph parts as sequences of render instruction, comparable to the *streamlined arrays* described above. These *instruction streams*, as they will be called in this work, have the chief benefit of reducing traversal cost; but they also allow for many optimizations to the same effect as the ones already described. Examples are sorting for state (*pull up costly state changes*, p. 6) and sharing hardware resources (*Share Attributes*, p. 9). As implicit side effect of using *instruction streams*, the structural complexity of the cached subgraph does not affect runtime performance, thus making transformations like *Collapse Hierarchy*, *Flatten Hierarchy* obsolete. Details on the optimizations facilitated in the caching system will be given in the main part of this work, most prominently in Section 3.6.

Some sophistication has also gone into keeping a scene graph cache up-to-date once it is built and, consequently, extending its lifetime. This is important, as building a cache is a costly operation and could negate any positive effects on performance if it was necessary too often. The next section will examine this problem of incrementally keeping complex structures up-to-date.

2.2 Incremental Computation

The *scene graph caching system* presented in this work is not only a form of scene graph optimization. Important parts of the mechanism behind it can be seen as a form of *incremental computation*. What is incremental computation? Ramalingam and Reps give a concise description in the introduction of their «Categorized Bibliography on Incremental Computation»:

> 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. [37, p. 1]

In case of the *scene graph caching system* presented here, the input data x is some (sub-) scene graph and the incremental function f is the function that yields a cache for x, and keeps the cache up-to-date incrementally. The details of how this is facilitated are given in the main part of this work, Sections 3.4 and 3.5.

A particularly interesting work is Acar's *Self-Adjusting Computation* [1] and subsequent publications (e.g. [2], [23], [4]). Therein Acar describes a general approach to creating incre-

mental programs by integrating needed constructs and runtime support directly into programming languages, both imperative ([3], [23]) and functional ([1], [5]). The execution of the incremental program—the instantiated call graph—is represented and stored in memory as a so-called *dynamic dependence graph* (DDG). If the program input is changed later on, the DDG allows to efficiently propagate the changes down to the output. By additionally integrating memoization into the change propagation mechanism (to avoid re-executing computations), the output of the program is thus updated with minimal effort.

However, there are several reasons why the above method was not chosen as the implementation strategy for incremental cache updates:

- There is no language or library support available for the implementation language C#. A complete reimplementation of the described system would have been out of the scope of this work—even more so, as the system would not have been needed in all of its generality.

- The scene graph system that constitutes the implementation context of this work does not provide means to explicitly track changes. Yet, the change propagation algorithm needs such a set of changes as initial input. Requiring all scene graph mutations to be tracked would have been too much of an imposition on production code.

- The above approach has a significant disadvantage in the context of scene graph caching: Change propagation is always performed in an eager manner. At the same time, large parts of a scene may not need any updates due to visibility culling. Applying the above methods directly for facilitating incremental updates may—depending on the scene graph and view configuration—result in redundant updates. This is especially detrimental because large parts of the «output» to be updated consist of resources residing in GPU memory. Mutating them unnecessarily would result in costly driver calls and waste of bandwidth on the memory bus. This is a problem-inherent constraint that would stay in effect even if the two constraints above, specific to the implementation context, were voided.

For these reasons, a different approach for handling incremental caching was chosen for the system proposed in this work. Nonetheless, Acar's work provided valuable inspiration, particularly, the idea of using a form of memoization to further improve incremental resource updates.[5] Also, we do not preclude the possibility that self-adjusting computations can be used in beneficial ways in conjunction with a caching system as proposed here. For now, however, an approach specialized for the task was chosen.

Because of the above considerations concerning visibility culling, in this context a lazy evaluation strategy is preferable to an eager one. Hudson [27] introduces an algorithm for incrementally evaluating attributes in a graph—a problem which is very similar to updating a scene graph cache, and for which efficient (eager) solutions have already been proposed earlier ([26], [6]). Again, like most incremental algorithms, dependency information is used to partially re-compute attribute values. An attribute can have parameter attributes which are needed to compute the attribute's value. Consequently, the attribute depends on its parameter attributes. The algorithm works in two phases: First, using the set of changes as input, all nodes and edges

[5] See section 3.6.7 for details.

in the graph that are influenced by any of the changes, are marked as such. This marking phase is executed in an eager fashion, but is rather light-weight. Second, starting at leaf nodes that have a demand for being updated (that are not «culled»), the algorithm recursively evaluates attribute values. This recursive evaluation implicitly causes attributes to be updated in a topologically sorted order and thus avoids repeated, unnecessary computations of the same attribute. And, because the recursion only descends into parts of the graph which are marked as «influenced», the update complexity is bounded by the size of the change and is independent of the size of the complete graph.

Unfortunately, Hudson's algorithm again relies on an explicit set of changes as input and can therefore not be implemented in the context of this work. The incremental cache updating mechanism needs to have a way of determining whether some cached part of the scene graph has changed. Luckily, we can present a way of implementing this efficiently in the main part of this work. Sections 3.4 and 3.5 will give a detailed description of the system in question.

CHAPTER 3

Scene Graph Caching

This chapter will present the *scene graph caching system*, which is the subject of this work. The first section will set the scope of the system by specifying clear goals for it to attain. The section *Implementation Context* will lay out the environment the system is embedded in. The next section *Caching Architecture* will show the basic functional principle of the caching system and elaborate on its benefits and limitations. In *Dependency System* a mechanism will be presented that allows to overcome some of the aforementioned limitations. *Dependency-Aware Scene Graph Caching* will show how this mechanism can be used to enhance the basic caching system. The next section *Optimizations* will discuss a number of improvements made possible by the system's architecture and show how to implement them. *Default Cache Types* will describe the two specialized kinds of caches that have been implemented for this work and will show examples of their usage. Finally, the concluding section of this chapter will discuss whether the proposed system meets the goals set out in the beginning.

3.1 Goals and Scope

In order to create and implement a clean and elegant design, it is important to clearly state the purpose and scope of the system to be developed. This section will give the aims the system will strive to achieve sorted by their relative importance.

Reduce Traversal Cost. This is the main purpose of the caching system. As graphics hardware and drivers have much improved over the years, applications can now easily become CPU bound and, in the case of scene graph based applications, traversing the scene graph to aggregate the parameters for rendering calls often has become the performance bottleneck. The caching system should try to improve the average number of frames rendered per second (FPS) by optimizing this process.

Allow Partial Caching. It should be possible to manually control which parts of a scene graph are cached and which are not. Caching may result in increased usage of both main memory

and graphics memory. Also, some parts of a scene might not be amenable to caching because they are very dynamic or behave in some non-standard way. Therefore the creator of a scene should be able to actively select for which parts of the scene a cache will be built.

Allow Modification. The user should be able to modify a cached subgraph, with as many kinds of modifications as possible resulting in a fully automated update of the cache. As a fallback for cases where an automated update cannot be facilitated, the user should be able to trigger a full rebuild of the cache.

Easy Configuration. The system should allow for configuring cache settings in a simple way. For the purpose of this thesis we will define this goal as follows:

- A single cache should be configurable by simply setting the appropriate flags on it. A cache configuration consists of the set of optimizations that will be applied within the cache and for which render passes a cache will be built.
- To cache a certain subgraph it should suffice to simply mark the root node in some way.
- Whenever possible, well-defined and practicable default settings and values shall be supplied. In cases with special requirements it should be possible to change the configuration; but setting up a system using only default values should already yield a fully functional setup.

Extensibility. The system should provide points of extension by adhering to the *Open-Closed principle* [30, p. 57] and other sound software design practices. In particular, the system should allow to incorporate custom scene graph nodes, cache types, and dependency predicates into the existing infrastructure.

Optionally Replace Cached Subgraph. It should be possible to erase a cached subgraph with the cache taking over all responsibilities for rendering and resource management. The precondition to this is that the subgraph is not dynamic, meaning it does not change in any way.

Use the Graphics Hardware Efficiently. To further improve rendering performance, the graphics hardware should be used as efficiently as possible. When there is potential to automatically exploit some hardware optimization or characteristic, the caching system should be able to do so. Yet, concerning performance, this is only a side goal. The main focus of the system is still reducing traversal cost.

The last section of this chapter (p. 87) will provide an evaluation of the system and the fulfillment of the goals listed above. To further define the scope of this work, we will also list a few *restrictions* and things the caching system will *not* do:

Leave Original Scene Graph Intact. The caching system should not modify any part of the existing scene graph. In other words, the system should store an *alternate runtime representation* (p. 11) and not apply any *persistent transformations* (p. 5). The original scene graph should stay available for the user to be modified—with the additional option of explicitly discarding the original scene graph and letting the cache take over, as mentioned above.

Avoid Duplicating Large Datasets. Very often the vast majority of storage space needed for a scene graph is geometry data, in particular vertex and index data. Structural data, such as groupings and transformation are comparably light-weight. Since it is a requirement that the original scene graph stays unmodified and fully functional, we chose not to duplicate geometry data for the cache. It rather is shared between original scene graph and caches. That being said, there are scenarios where optimizations like *Collapse Geometry* (p. 10) are beneficial and incorporating this functionality into the caching system might prove interesting in the future.

GPU Programs are not modified. The caching system does not modify shader code in any way. This is a topic of its own and would exceed the limited scope of this work. See *Graphics Pipeline* in the next section for further information.

3.2 Implementation Context

The first part of this section will describe the graphics pipeline model used for reasoning about possible implementations of the caching system. The second part will then shortly describe the rendering backend used in the prototypical implementation of the caching system. This rendering backend constitutes the low-level interface the system is programmed against and abstracts the underlying hardware and software API. The last part of this section will present the scene graph environment that hosts the prototype.

3.2.1 Graphics Pipeline

In order to identify possible points of performance optimizations one needs a model of the hardware and software processes which are active while rendering a 3-dimensional scene using modern graphics hardware. Such a structured model allows to pinpoint possible bottlenecks in the processing pipeline and consequently find specific solutions to improve overall throughput.

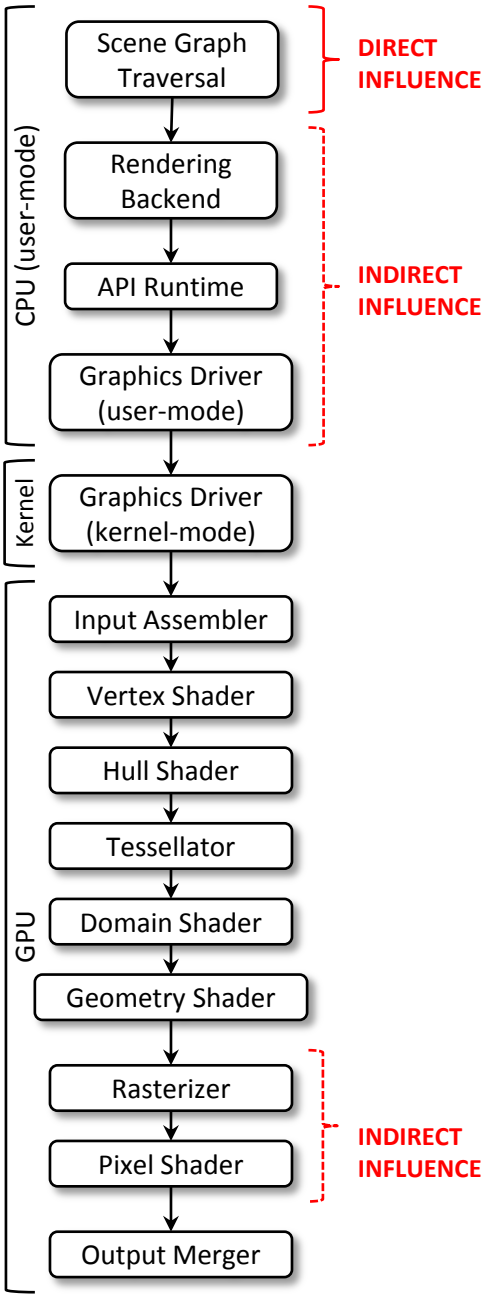


Figure 3.1: *Direct3D 11* graphics pipeline of a scene graph application

Figure 3.1 shows an overview of the various stages involved in drawing a scene graph into a render target. [1] The main sources for this section are documentation provided by Microsoft ([12], [14]), Giesen (2011) [22], and *NVIDIA's GPU Programming Guide* [16]. First, the scene graph needs to be traversed in order to generate commands for the rendering backend, which sits on top of the actual graphics API such as *Direct3D* or *OpenGL*. The backend calls into the graphics API runtime, which communicates with the part of the graphics driver that runs on the CPU in user-mode, which in turn accesses the kernel-mode part of the driver. From here on, work is executed on graphics hardware. The *Input Assembler* reads index and vertex data and prepares it as primitives for the rest of the pipeline. The *Vertex Shader* processes single vertices, at least transforming them into screen space. *Hull Shader*, *Tessellator*, and *Domain Shader* allow for hardware-based tessellation of higher-order surfaces. The *Geometry Shader* processes geometry on a per-primitive level and can, for example, discard an entire triangle from the pipeline. Next, primitives are rasterized into individual pixels which are then processed by the *Pixel Shader*. Finally, the *Output Merger* writes the pixels coming from the *Pixel Shader* into the render target, using a configurable semantic like *alpha blending*.

It is important to note that data and commands between the different stages may be passed in larger batches as opposed to single items. For example, the user-mode graphics driver accumulates a *command buffer* which it only sends to the kernel-mode driver if it is explicitly told to do so or if the buffer reaches a certain size. [15] This batching is performed because switching to kernel-mode is a costly operation. As a side effect, it becomes harder to accurately measure graphics performance, as it cannot be predicted if an API call triggers the execution of the command buffer.

[1] The diagram shows the setup on *Windows 7* running *DirectX 11*. Other operating systems and graphics APIs can have a slightly different configuration.

For an overview of the complexities involved in profiling an application based on *Direct3D 9* see *Accurately Profiling Direct3D API Calls (Direct3D 9)* [10].

Almost any of these stages can become the performance bottleneck of an application. [16, p.11] However, the *scene graph caching system* will only influence the very first item in Figure 3.1 directly: scene graph traversal. As mentioned, we chose to avoid modifying vertex data and shader program code directly. The former can help in reducing vertex shader load—like when using level-of-detail techniques[2]—and the later may reduce the processing cost for all shader stages. While some sort of automated level-of-detail support could be incorporated into the caching system, the optimization of shader code is a very complex issue and would warrant a work of its own.[3]

Nonetheless, certain parts of the pipeline on the GPU can benefit from being fed data in a different order, allowing them to utilize memory caches more effectively or use special graphics hardware features, such as *Early Z Culling*. The caching system supports optimizations that try to aid in this direction (see *3.6.3 State Sorting*, p. 65 and *3.6.4 Overdraw Sorting*, p. 65). Mostly however, the caching system is focused on reducing scene graph traversal cost and consequently reducing CPU workload.

3.2.2 Rendering Backend

The AARDVARK scene graph toolkit—which provides the implementation context of this work—comes with a rendering backend that acts as an abstraction layer over the actual graphics API such as *Direct3D* or *OpenGL*. The programming language interfaces and types it provides are closely modeled after the *Direct3D 11* API, with resource types such as *index, vertex,* and *constant buffers* and a renderer interface similar to *ID3D11DeviceContext*.

Apart from providing an abstraction over different graphics APIs, the main benefit of this backend is its ability to «compile» single instructions provided in the renderer interface into persistent objects. This instruction object encapsulates a renderer call, such as setting a buffer reference or drawing a range of vertices, and can be re-executed at any time. As the arguments of the renderer call are bound to the instruction when it is compiled, resource handles can be stored as resolved implementation-specific references and thus do not need to be resolved for every execution. The main advantage of a persistent form of render instructions is that a sequence of such instructions can be stored, analyzed, and reasoned about. The positive consequences of this properties will be shown later in the sections *3.3.1 Instruction Streams* (p. 26) and *3.6 Optimizations* (p. 64)

3.2.3 Scene Graph Environment

The caching system prototype was implemented within the AARDVARK scene graph environment. AARDVARK is developed at the VRVis Research Center where it is used in a number of

[2]Note that using level-of-detail techniques is not precluded by the *scene graph caching system*.

[3]In addition, the most worthwhile optimizations mostly result from choosing a smarter algorithm to solve a specific problem and can therefore not be performed automatically.

different projects. It provides all of the common scene graph functionality: geometry, transformation, surface, and group nodes to represent data; traversals to implement operations.

A prominent trait of AARDVARK's scene graph model is its extensibility and flexibility. New node and traversal types are easily defined, especially because of the *generic scene graph traversal* semantic the system employs. This frees implementers of new node and traversal types of the burden of handling interaction with existing traversal and node types by providing a traversal forwarding mechanism. In effect, this reduces the implementation effort from $\mathcal{O}(T \cdot N)$ down to the strictly necessary traversal/node combinations. [42, p. 4-5]

Another important feature of AARDVARK is its support for separating the *semantic* scene graph from the *rendering* scene graph, somewhat similar to the approach presented by Hopcroft et al. (p. 11). A node type can be implemented as an *Instance/Rule* pair, where the *Instance* is the declarative, semantically clean representation of the node, while the *Rule* dynamically provides the rendering representation of its *Instance*. This allows to model scenes on a higher abstraction level with implementation details generated programmatically instead of by the artist. It also provides a uniform mechanism for modeling dynamic state because *Rules* describe a translation from *semantic* to *rendering* scene graph which may also incorporate changing, global state. [42, p. 2-4]

Note that the caching system presented in this thesis does not critically rely on any of these special features. Nonetheless, they certainly made implementing the prototype quite a bit easier.

3.3 Caching Architecture

This section will present the basic architecture of the *scene graph caching system*. First, the fundamental mechanism of the system is shown in *Instruction Streams*. Next, the section *Cache Nodes & Render Caches* will show, how *instruction streams* can be integrated with the scene graph and *Cache Creation* will elaborate on the process of building render caches. Finally, the chief limitation of the system presented so far will be discussed, namely the problem of *Cache Invalidation*.

3.3.1 Instruction Streams

The first question to answer is: What constitutes a *cache* in this system? What does it mean to cache part of a scene graph? The answer has already partly been given in the *Related Work* section. Similar to the approach by Durbin et al. [18] the caching system creates so-called *instruction streams* which are a persistent representation of the commands sent to the rendering backend when performing a render traversal on a subgraph. An *instruction stream* can be stored in memory and, when re-executed, yields the same result as a render traversal would, albeit at much lower computational expense.

An instruction encapsulates a single command to be executed by the rendering backend. A command consists of the *API* routine to be executed and the arguments for executing this routine. In the rendering backend described above, the possible routines map very closely to the methods available in *ID3D11DeviceContext*. A partial listing is given in Table 3.1.

ROUTINE	PARAMETERS
SetPrimitiveTopology	a *primitive topology*
SetVertexShader	a *vertex shader*
SetShaderInputLayout	a *shader input layout*
SetFragmentShader	a *fragment shader*
SetGeometryShader	a *geometry shader*
SetVertexBufferBinding	a *vertex buffer binding*
SetIndexedVertexBufferBinding	an *indexed vertex buffer binding*
SetConstantBuffers	a set of *(slot-index, constant buffer)* pairs
SetShaderResourceBuffers	a set of *(slot-index, constant buffer)* pairs
DrawIndexed	*start index* and *element count*
DrawArrays	*start vertex* and *element count*

Table 3.1: Instruction Types

An *instruction stream* is just a linear sequence of instructions, comparable to the byte-code of an interpreted language. As every type of instruction must implement an *Execute()* method (which simply calls the stored routine with the stored arguments), a full *instruction stream* can be executed by iterating over the sequence and calling *Execute()* on each instruction. As can be observed in Table 3.1, instructions often reference resources such as *constant buffers* as their arguments. Figure 3.2 gives an example of small stream. In practice the number of instructions per stream can go well into the tens of thousands.

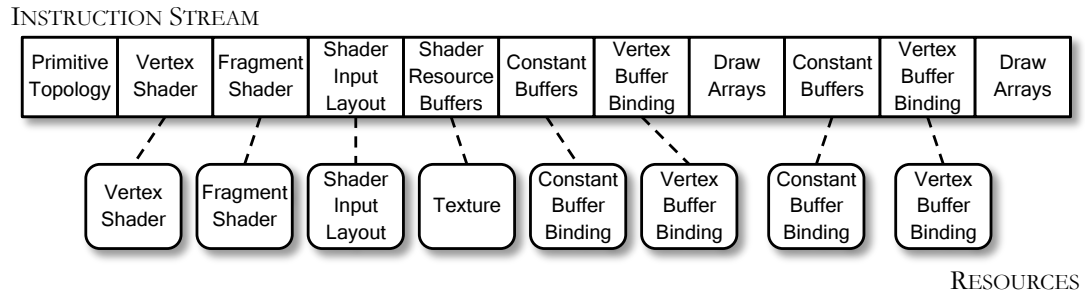


Figure 3.2: A small *instruction stream* with resources as instruction arguments.

There are several benefits an *instruction stream* has over a scene graph representation in the given scenario:

Minimal Traversal Cost. As an *instruction stream* is just a flat sequence—an *array* in the prototypical implementation—iterating over it is very fast. Apart from the dynamically bound call to *Execute()* there is no branching code required in the iterator loop, while traversing a scene graph typically requires more complicated navigation logic.

Pre-Resolved Resource Handles. The rendering backend uses lazy resource handles to reference resources like vertex buffers. On the first use, the requested resource is created by a factory closure stored by the handle. For the underlying graphics API (*D3D* or *OpenGL*) to use the resource, the native value first has to be extracted from the backend wrapper object and cast to the correct type. Since instructions are «compiled» by the backend at runtime, they are already specialized for the underlying graphics API and can thus directly store the pre-resolved, native resource reference. [4]

Compactness and Cache Coherency. Instructions only store the information that is absolutely necessary and in most cases the execution order of an *instruction stream* does not change once it is created. [5] Consequently, it is easy to allocate an *instruction stream* with optimal layout for sequential access by placing the instructions in memory in execution order.

Easy to Analyze. Once an *instruction stream* is created it can be reasoned about and possibly optimized. Because of its strictly linear structure and the absence of jumps and branching, the semantic impact of a code transformation can easily be determined. The section on *Optimizations* will show examples of exploiting this characteristic.

Caching of Computational Results. For spatial transformations the regular scene graph render traversal every frame has to recompute the accumulated transformation matrix required at the leaf nodes. As the *instruction stream* is generated with full knowledge of the traversal state at a given leaf, the *SetConstantBuffers* instruction setting the transformation can store the accumulated matrix already at build time. Later executing the stream will just use the stored value without doing any recomputation.

However, in addition to these positive properties *instruction streams* have one important disadvantage: When an *instruction stream* is created it can always only take a snapshot of the scene state at the time of creation. As the underlying scene graph changes, the semantic of the graph and the stream drift apart: The cache becomes stale. The section on *Cache Invalidation* will go into more detail on the problem and *Dependency-Aware Scene Graph Caching* will show how this problem can be coped with efficiently in many situations.

Another problem to solve is how to exactly integrate *instruction streams* with the scene graph system. The next section will show how this question has been answered in the case of the system presented here.

3.3.2 Cache Nodes & Render Caches

To actually use *instruction streams* for caching a scene graph, they have to be created, stored and maintained somewhere. This can be implemented several ways: storing them at the node level like done by Durbin et al. [18], keeping them in an external caching manager outside of the

[4] Conceptually, this is also possible in a scene graph representation. However, the result would be very similar to storing a small, local instruction stream at the node level.

[5] That is, there are no conditional *jump* instructions. Examples of when the execution order *does* change are: (1) when transparency necessitates back-to-front rendering (see *3.7.2 TransparencyPassCache*), and (2) when performing front-to-back rendering as an optimization (see *3.6.4 Overdraw Sorting*).

scene graph, or—and this is the approach that has finally been chosen by us—introducing a new type of scene graph node, the so-called *cache node*.

To select a subgraph for caching, a *cache node* is placed at the root of the subgraph. From then on the *cache node* will take on all responsibilities for building, maintaining and using *instruction streams* for rendering. A *cache node* can contain a separate *render cache* for each rendering pass it supports. A *render cache* is an object that implements the *IRenderCache* interface. The default cache types supported by the prototype are *SolidPassCache* and *TransparencyPassCache*, which will be discussed later in 3.7.1 and 3.7.2. Each *render cache* holds a type-specific number of *instruction streams* or it may not use *instruction streams* at all, as the *IRenderCache* interface does not prescribe the implementing class to do so. Figure 3.3 shows relationships between the various entities.

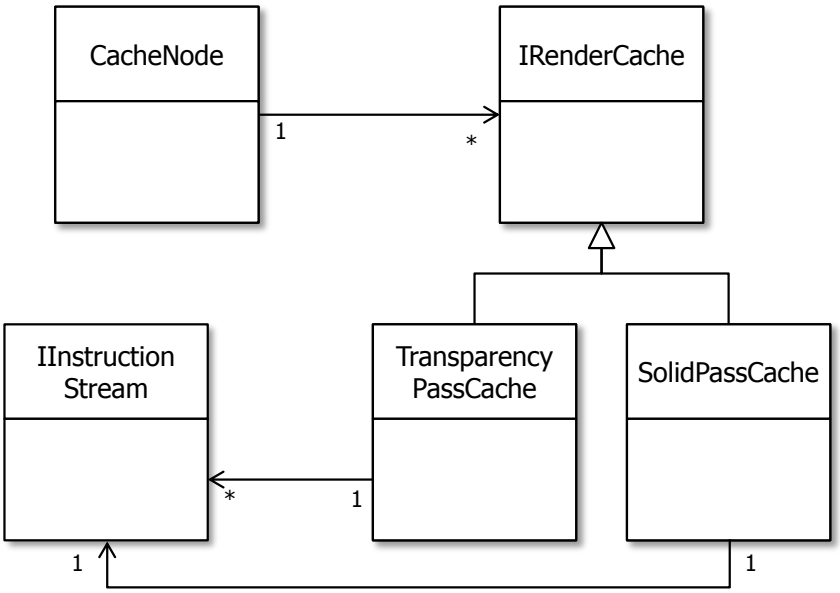


Figure 3.3: Relationships between *cache nodes*, *render caches*, and *instruction streams*

Semantically, a *cache node* caches all of its descendants, which may include *group nodes, transformation nodes, surface nodes, geometry nodes* and any other node type that implements the *ICacheable* interface. [6] Figure 3.4 shows examples of valid setups (blue nodes denote cache nodes).

It is not allowed, however, to place a *cache node* within a subgraph that is already cached by another *cache node*. The subgraphs of two *cache nodes* may overlap, although this is not recommended as it will lead to duplicate storage and maintenance efforts. Figure 3.5 shows examples of these invalid, respectively, discouraged configurations. The intended configuration

[6] Due to the *generic traversal* algorithm [42, p. 4] of the scene graph environment used for implementing the prototype, node types not implementing *ICacheable* are automatically ignored. In most cases, this is the desired behavior. If not, the system stays extensible by allowing custom implementations of *ICacheable*.

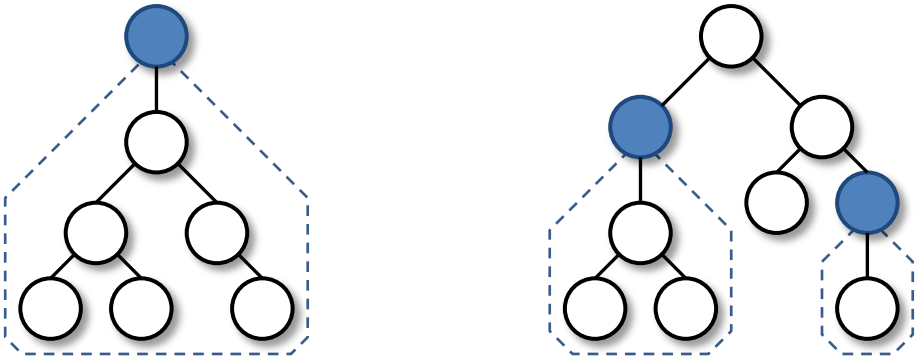
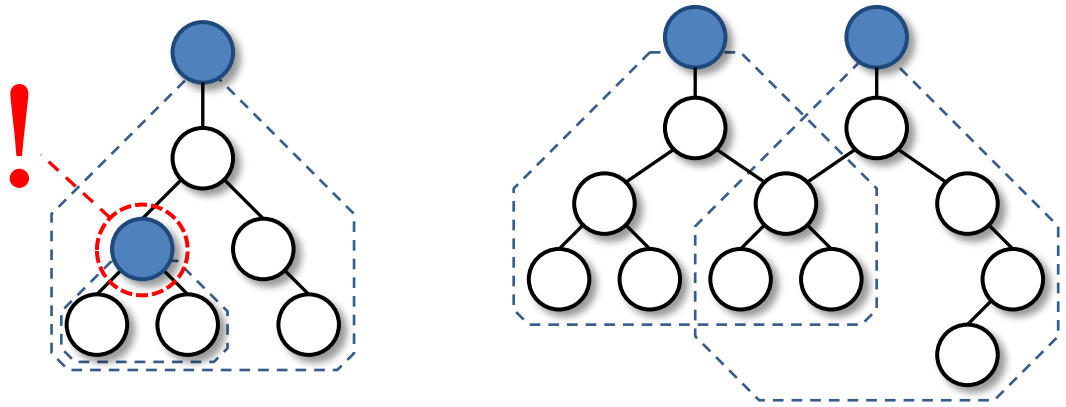


Figure 3.4: Valid placements of *cache nodes* in a scene graph.

is that a cached subgraph has its single root below the *cache node* and is not reachable by any other path.



(a) (b)

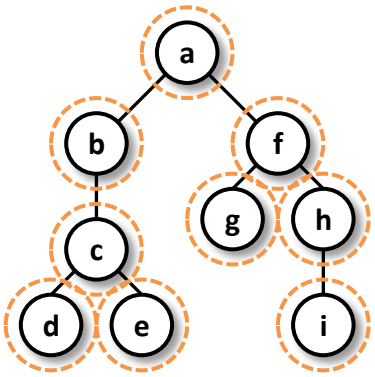
Figure 3.5: A nested—and therefore invalid—cache node placement (a) and a configuration with overlapping, cached subgraphs (b)

Using this method of placing *cache nodes* above certain subgraphs, the render traversal can execute a *cache node's instruction stream* instead of descending into the subgraph. This is where the major performance improvements are to be gained. Figure 3.6 shows a comparison of node processing by a render traversal with and without caching.

3.3.3 Cache Creation

Now that the basic caching structure has been set forth, the cache creation process will be demonstrated using the example of a *cache node* supporting a single *render cache* with a single *in-*

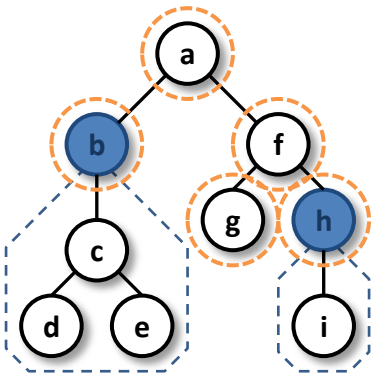
Figure 3.6: Render Traversal with and without cache nodes



(a) Nodes visited (no caches)

Traversal Index	Node
1	a
2	b
3	c
4	d
5	e
6	f
7	g
8	h
9	i

(b) Traversal Order (no caches)



(c) Nodes visited (with caches)

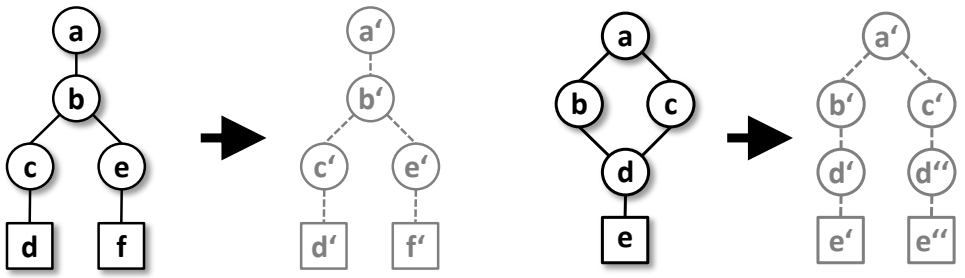
Traversal Index	Node
1	a
2	b
3	f
4	g
5	h

(d) Traversal Order (with caches)

struction stream. As will be shown later, this configuration already is very similar to the default *SolidPassCache* type. The process as presented in this section will undergo various extensions in the rest of this chapter and the complete, final version will be shown in Section 3.7.1. Yet, the following description maintains its validity as the fundamental principle of the cache creation algorithm stays unaltered.

The goal of the creation procedure is to construct the *instruction stream* within the *render cache*. This is achieved by a pipelined process that transforms the scene graph into an *instruction stream* through various stages.

1. If a *cache node* determines that it needs to build a new *render cache* (because it does not already own a cache for the current pass), it will ask the *RenderCacheFactory* to create a new instance.



(a) (b)

Figure 3.7: Two scene graphs and their corresponding *SlimSG*.

2. The factory starts an *ExtractCachingDataTraversal* at the *cache node*. The traversal will invoke the *ExtractCachingData* method from the *ICacheable* interface on every implementing node in the subgraph. This way a filtered, hierarchical representation of the subgraph is created. This representation, called the *SlimSG*, has the structure of the depth-first traversal tree of the subgraph: Every path from the root to a leaf of the subgraph will end in a distinct leaf in the *SlimSG*. If a leaf node in the subgraph is reachable by two paths, all nodes starting from the first divergence of the two paths will be duplicated for the *SlimSG*. If the subgraph already is a tree, the structure of the *SlimSG* will be equal to the structure of the subgraph. Figure 3.7 shows a few examples of this transformation.

   Each leaf node of the *SlimSG* holds references to the resources that are needed to render the leaf—most notably *index*, *vertex*, and *constant buffers*. Ownership and life-cycle management of the various resources will be discussed later in *3.5.2 Resource Management*.

3. Once the *SlimSG* is built, the *RenderCacheFactory* internally selects a factory function appropriate for the current rendering pass. At this point, the cache construction process may diverge for different *render cache* types. As stated above, this example will show the construction of a very simple *SolidPassCache* instance. All subsequent steps are specialized for this type of *render cache*.

4. The *ISlimSg* interface implemented by all *SlimSG* node types contains a *CreateRenderJobs()* method which takes a *RenderJobBuilder* instance as sole parameter. This constitutes an implementation of the *Builder* design pattern [21, p. 97], with the *SlimSG* taking the role of the *director* and the generated list of render jobs being the *product*. The *RenderJobBuilder* possesses a «renderer-like» interface (Listing 3.1) which allows the *CreateRenderJobs()* methods to simply mimic the sequence of rendering commands that would be generated by performing a normal render traversal on the original subgraph. *CreateRenderJobs()* is called on the root of the *SlimSG* and will propagate itself recursively down into the leaves, all the while setting appropriate state on the builder object. Internally the builder will create a list of *render jobs*. Each *render job* encapsulates a single call to one of the *Draw*()* methods of the underlying graphics *API*, plus the complete

state of the rendering pipeline at the time of the draw call: active *shaders* and *shader input layouts, vertex, index*, and *constant buffer bindings*, etc.

```
interface RenderJobBuilder
{
   void SetPrimitiveTopology(PrimitiveTopology pt);
   void SetInputLayout(IShaderInputLayout inputLayout);
   void SetSurface(Surface surface);

   void SetConstantBuffer(int slot, IConstantBuffer buffer, ShaderType shaderType);
   void ClearConstantBuffer(int slot, ShaderType shaderType);
   void SetShaderResourceBuffer(int slot, IShaderResourceBuffer buffer,
      ShaderType shaderType);
   void ClearShaderResourceBuffer(int slot, ShaderType shaderType);

   void SetIndexBuffer(Buffer indexBuffer);
   void BeginVertexBufferBinding();
   void BindVertexBuffer(String semantic, Buffer buffer);
   void EndVertexBufferBinding();

   void DrawIndexed(int startIndex, int elementCount);
   void DrawIndexed(int elementCount);
   void DrawArrays(int startIndex, int elementCount);
   void DrawArrays(int elementCount);
}
```

Listing 3.1: *RenderJobBuilder* Interface

5. Since each *render job* is self-contained, the list of *render jobs* can be sorted in any order. This can be utilized to perform *State Sorting* at this point in the process.

6. Once the *render jobs* are in the desired order, a code generator algorithm produces a sequence of instructions from each *render job* by simply producing one instruction for every item of state in the *render job*, setting the appropriate *shaders*, bindings and so on. The instructions produced by the generator are so-called *semantic instructions*, meaning, they fully define the semantics of the stream but are still independent of a specific graphics *API*. *Semantic instructions* are comparable to an intermediate representation of a program in a compiler, like *LLVM IR* [28], which is not yet translated into machine code for any specific architecture.

7. In the last step, the *semantic instructions* are handed over to an *IInstructionStreamFactory* instance which translates them into so-called *native instructions*. A *native instruction* is bound to a specific graphics *API* and is directly executable. The sequence of *native instructions* is the final *instruction stream* and with it the *render cache* is complete.

Figure 3.8 shows the process in form of an *UML* activity diagram. As can be seen, the process is quite involved and consequently can be very expensive, especially because a potentially great number of resources (mostly *constant buffers*) are created when the *SlimSG* is built by the *ExtractCachingDataTraversal*. It is therefore crucial to avoid the need to rebuild the cache as much as possible; otherwise any positive performance effect of the caching might be negated by the time-consuming cache construction process. Not only could the average *FPS* actually be lower than without caching—as cache construction blocks the rendering thread, it might also introduce a noticeable, indeterministic stutter into the application. The next section will investigate the conditions of cache invalidation as a first step to overcome this undesirable behavior.

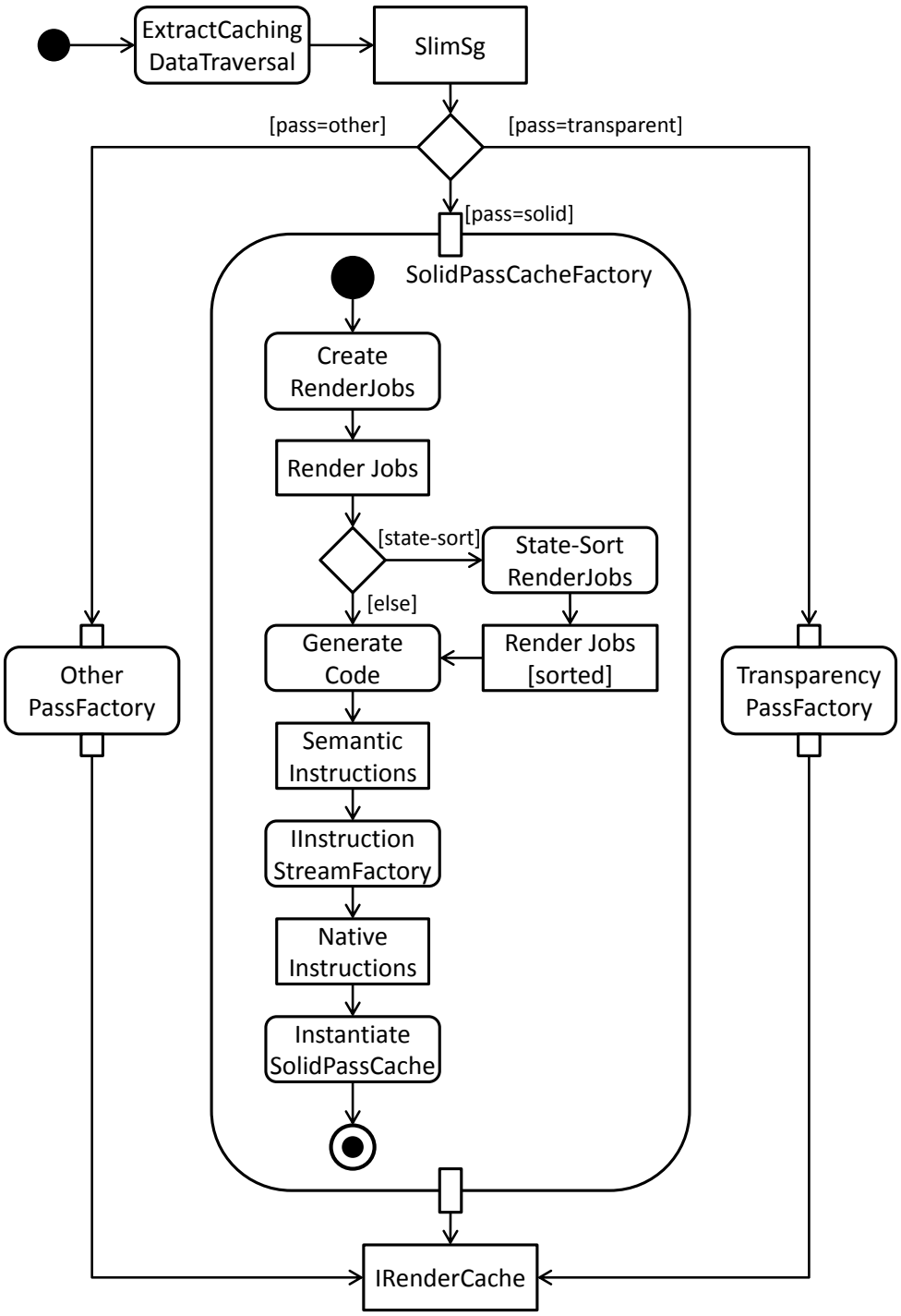


Figure 3.8: Simplified *render cache* construction process

3.3.4 Cache Invalidation

The chief detriment of the caching system presented so far is caches becoming stale due to small changes in the original scene graph and consequently necessitating a full rebuild of the cache. This section will investigate the circumstances that can cause the *instruction stream* in a *render cache* to grow out-of-date. There are a number of mutations that influence the mapping from source scene graph to *instruction stream*:

- Adding a *geometry leaf* to a cached subgraph must be reflected in the stream by additional *draw call* instructions and other instructions setting the appropriate state for the draw call(s), such as *index, vertex,* and *constant buffer bindings*.

  Correspondingly, when a *geometry leaf* is removed from the cached subgraph, the draw call instructions rendering the contained geometries must be omitted from the stream. In addition, the instructions setting the state for the draw call should be removed in order to avoid unnecessary *API* calls. Note, however, that this is not entirely trivial, as a single state-setting instruction affects the state active at any subsequent draw call unless another state setting instruction of the same type starts a new «scope». A state-setting instruction that is thus «bound» by another draw call must not be removed from the stream.[7] Adding or removing *geometry leaves* outside of a cached subgraph does not have any effect on the cache.

- Leaf nodes in a scene graph are a special case because the semantic of no other node depends on them. For nodes setting pipeline state the situation is more complex: Every *geometry leaf* reachable from the node is influenced by it (except if the same state is «overwritten» by another node down the path). Thus, changing for example the material in a single *MaterialApplicatorNode* might change the material of any number of *geometry leaves* and necessitate mutating many *constant buffers* referenced by instructions scattered all over the stream.

  The most important consequence of this «path-based» semantics exposed by scene graphs, however, is that a change *outside* of a cached subgraph can influence the semantic of the subgraph and, as a result, render the cache and its *instruction stream* stale: Mutating a state-setting node on a path from the root to a *cache node* will require changing all instructions and resources referencing the changed state in some way. Figure 3.9 shows an example of a change outside the cache affecting nodes inside the cache (a) and a change contained inside the cache (b). Yet, it is important to note here that neither the number nor the linkage of instructions and resources is altered, leaving the *instruction stream* structurally equivalent. Only argument values stored in resources or directly in instructions are mutated. Neither are any instructions added or removed, nor do they reference any different resource instances than before. This circumstance will later become of great importance.

[7] The topic of state scope in *instruction streams* will be further discussed in *Overdraw Sorting* where it is of vital importance.

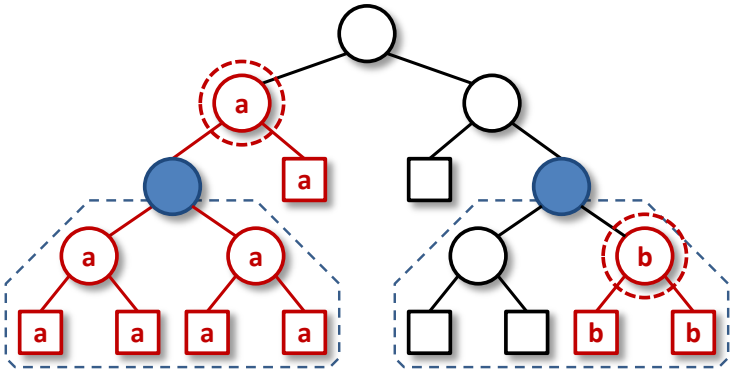


Figure 3.9: Attribute mutations outside and inside a cached subgraph can cause cache invalidation.

- *Transformation nodes* are a special case of the above, as their values do accumulate down the path instead of being overwritten. This means that changing the matrix in a *transformation node* may potentially influence even more graph leaves than changing other attributes.

- Creating new edges between existing nodes can also make an *instruction stream* stale, as a new edge can introduce new paths via which a *geometry leaf* is reachable and consequently causing it to be drawn twice or more times (with different states or transformations). Again, like with *geometry nodes*, removing edges may cause the opposite effect of draw calls needing to be removed. Since changing a single edge can add or remove any number of paths in the scene graph, such a comparatively small change in the original scene graph can mean a very significant change to its semantic and therefore also an *instruction stream* modeling it.

  Creating new paths *within* the cached subgraph will have a similar effect as adding new *geometry leaves* to the subgraph. However, making the *cache node* at the root of the subgraph reachable via a new path will «duplicate» the whole subgraph. The prototypical implementation of this work deals with this situation by creating two entirely separate *render caches*, respectively a separate *render cache* for each path the *cache node* is reachable by. An alternative would be to create one cache containing all concrete instantiations of the subgraph. But having separate *render caches* allows for the possibility of handling them independently when doing visibility culling.

- Adding or removing nodes other than *geometry leaves* has very similar consequences to changing the attribute value in an existing node—as long as no new paths are introduced. If for example a new *transformation node* is introduced along an existing path (Figure 3.10a), the effect is the same as when just multiplying the new matrix with the value of either the preceding or the subsequent *transformation node* (in the right order). The *instruction stream* stays structurally unaltered.

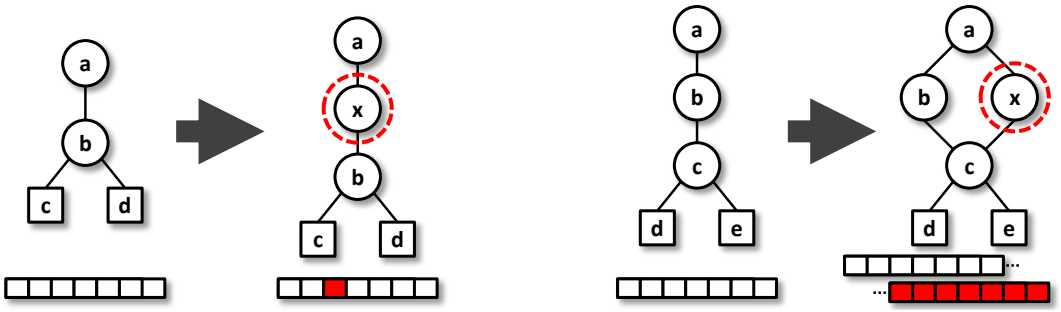


Figure 3.10: Additions of attribute nodes causing *instruction stream* mutations. In (a) the number of paths from root to leafs is the same before and after the change. In (b) there are two paths before the change—$\{(a, b, c, d), (a, b, c, e)\}$—and four paths after—$\{(a, b, c, d), (a, b, c, e), (a, x, c, d), (a, x, c, e)\}$

When adding the *transformation node* while also introducing a new path, for example by placing the node parallel to another one and connecting it to form a diamond shape with the existing node (Figure 3.10b) then the same rules apply as when adding new edges in any other way. This case is just a composite of (1) adding a new node, not introducing a new path, (2) adding a new edge, not introducing a new path, (3) adding another edge, this time also creating one or more new paths.

MUTATION TYPE	STRUCTURAL CHANGE	NODE CHANGES SIGNIFICANT
Add | Remove Geometry Leaf	Graph & Stream	Within Cache
Add | Remove Attribute Node	Graph	Within & Above Cache
Add | Remove Edge	Graph & Stream	Within & Above Cache
Change Attribute Value	None	Within & Above Cache

Table 3.2: Different changes to a scene graph and how they affect cache invalidation.

Table 3.2 gives a compact overview of the different cases and their characteristics. In summary, an *instruction stream* (and the resources it references) is very sensitive to changes in the scene graph it maps. The caching system as presented so far has therefore a very narrow set of use cases; mainly speeding up the rendering of completely *static* portions of a scene, such as terrain or buildings. While the system is not entirely useless, we can do better than this. This is where the *dependency system* presented in the next section comes into play and the section after the next—*Dependency-Aware Scene Graph Caching*—will show how it allows to significantly reduce the number of times the cache needs to be fully rebuilt.

3.4 Dependency System

The purpose of the *dependency system* is to provide a way of updating resources only when needed. This is facilitated by establishing a simple ontology that allows to model the logical dependencies of resource values and how they can be recomputed on demand.

At the beginning of this section the *Assumptions and Requirements* applying to the *dependency system* will be discussed. The rest of the section will then show the implementation that was chosen to fulfill the given requirements. First, the *Basic Concepts* the system is based on will be presented and then several implications arising from these concepts are described, such as *Resource Construction*, *Resource Composition*, *Resource Equality*, and the *Evaluation Order of Dependencies and Dependent Resources*.

3.4.1 Assumptions and Requirements

Arising from the scene graph environment the *dependency system* is developed for, there are some special assumptions that differ from the ones applying to a general purpose *incremental computing* solution:

- Updating a resource can be very costly as the majority of resources resides in graphics memory and changes will have to be transmitted over the *PCI Express* bus.
- There is always the possibility of (a large number of) objects being *culled* from the current view which should require as little computational resources as possible.
- Due to the scene graph environment the system is to be used in, there is no preexisting way to explicitly track changes made to the scene graph.
- The *dependency system* will be used to support the caching system.

Under these assumptions the following requirements for the *dependency system* were formulated:

1. The system must provide a way to update resources if and only if they actually need to be updated. For example, if an existing *constant buffer* stores the transformation matrix for some geometry, this *constant buffer* should not be mutated unless the transformation of the geometry actually changes.
2. Asserting whether a resource is not up-to-date anymore must be considerably cheaper than performing the resource update.
3. The system must not prevent employing a scene graph toolkit's frustrum and occlusion culling mechanisms, as these optimizations can be very important to rendering performance. Having to disable them can in many cases negate any positive effects of the caching system altogether. Ideally, the *dependency system* should even support the culling process and should itself profit from culling, if possible.

4. Since the whole caching system prototype is implemented in an existing scene graph environment actively used in production, the *dependency system*—same as the caching system—must avoid breaking any interfaces or cause any other unnecessary implementation overhead for the existing user base. Where possible, interfaces should be extended without changing their current meaning and the *dependency system*'s impact outside the caching system should be as small as possible.

5. As with the caching system in general, the *dependency system*—if actually used—is permitted to cause some additional computational effort at startup. For parts of a scene graph that do not make use of it, the system should not cause any significant performance overhead.

With these requirements laid down a concrete system has to be designed that implements them. The rest of this section will show our result to this challenge.

3.4.2 Basic Concepts

The *dependency system*, as we chose to implement it, has three basic concepts at its foundation. These concepts form an ontology that allows to assign roles and relations to the various scene graph entities, which in turn allows to implement the desired selective updating mechanism. These concepts are:

(1) *Dependencies* - A kind of metadata that allows for change tracking.

(2) *Value Sources* - Everything that provides input values needed for resource updates.

(3) *Dependent Resources* - All resources that should support smart, selective updating.

In the following paragraphs these three concepts are described in more detail:

Dependencies A *dependency* is a stateful predicate. A *dependency* provides a way of determining if some item has changed with respect to some former state of the item. It can be used to model predicates like «Is object X within range (a,b)» or «Has the camera moved more than 10 units since the last check». This functionality is implemented by a version number. Every time the observed object changes—according to the *dependency*'s definition of change—the version number is incremented. This way every distinct state the observed object takes during its lifetime has a unique version number. This version number can be used by other objects as an abstract reference to a specific state of an object. Consequently, other objects can determine if their view of the observed object is still up-to-date by a simple comparison of the *dependency*'s current version number and the version number the *dependency* had when last used by the other object. This ability to quickly check for changes is key to efficiently determine if a *dependent resource* needs to be updated or if its state is still valid.

A *dependency* instance is also allowed to store arbitrary state (although, the less the better). This can be used to store reference values to compare against when modeling conditions

such as «Has the position of object X changed more than Y units?» To evaluate such a condition, the object's former position must be known.

Arbitrary new kinds of *dependencies* can be created by implementing the *IDependency* interface shown in Listing 3.2.

```
interface IDependency
{
   // Returns the current version number of this dependency instance.
   DependencyVersion Version { get; }

   // Check whether the change criterion of the dependency is met
   // and if so, increments the dependency instance's version number
   void Evaluate(Traversal t);
}
```

Listing 3.2: The *IDependency* Interface

The implementing class can store any data it needs as instance fields and have an arbitrary constructor to initialize them. The *DependencyVersion* structure is just a thin wrapper around an integral value providing an abstraction of the underlying type (*unsigned int* in the current implementation) and some convenience methods easing its use. The *Evaluate()* method must implement the change tracking and increment the *dependency* instance's version number to indicate a change. It can use the *Traversal* object passed as argument to access environmental state, such as the current transformation stack or the current time.

input: Dependency d, Traversal t

1 **if** $distance(d.\text{RefPosition}, d.\text{ObservedObject.Position}) > d.\text{Epsilon}$ **then**
2 $\quad d.\text{RefPosition} \leftarrow d.\text{ObservedObject.Position}$
3 $\quad d.\text{Version} \leftarrow d.\text{Version} + 1$
4 **end**

Algorithm 3.1: Evaluating a *positional change* dependency.

Algorithm 3.1 shows an example implementation of a simple *dependency* that tracks positional changes of some object. The *dependency* must be supplied with an instance-specific epsilon value that allows to adjust the tracking sensitivity, and the object to be tracked. It is recommended to store such instance parameters in *readonly* fields of the concrete dependency class. The *dependency* instance also stores a mutable *reference position* to compare the object's current position against and increments the *dependency* version if the object has moved more than epsilon from the reference position. At the same time the reference position is set to the current value. The *Traversal* argument is not needed in performing this check.

Values Sources A *value source* is an item in the scene graph or global environment that is needed when computing the value of a *dependent resource*. An example for a *value source* would be a *transformation node*, the value of which is needed to calculate the model-transformation of a leaf some way down the scene graph.

A *value source* holds a set of *dependencies*. This set of *dependencies* must accurately reflect when the value of the *value source* changes. For example, if the *transformation node* mentioned above always changes whenever a key is pressed, then the node must have

a *dependency* that increases its version number whenever this event occurs. *Dependencies* are the designated means to get information about changes of *value sources*. Listing 3.3 shows how *value sources* a represented in code.

During traversal *value sources* are collected in a list in the traversal state so that all *value sources* on the path from the root to the current node are always available and can be used later to construct the *update action* for a *dependent resource* as explained below.

```
// Something that is needed by an IDependent
// to update its value.
interface IValueSource
{
   IEnumerable<IDependency> Dependencies { get; }
}
```

Listing 3.3: The *IValueSource* Interface

Since a *value source* does not just hold a single *dependency* but a set of *dependencies*, the semantics of *dependency* changes have to be clarified. What does it mean if one of n *dependencies* increases its version number while the others do not? What does it mean if a *dependency* set of a *value source* is empty? The answer to these questions should be consistent with intuition: If an object has two *dependencies*, it depends on both—a change (i.e. a version number increment) in one of the *dependencies* means that the object is out of date. Only as long as none of them changes the object is up-to-date. Correspondingly, if an object has no *dependencies* then that should mean that it never changes. Thus, for a set of *dependencies* «change» is defined disjunctive:

$$\text{changed}_{\text{depset}}(\{d_1, \ldots, d_n\}) = \exists i(\text{changed}_{\text{dep}}(d_i))$$

Dependent Resources A *dependent resource* has three defining properties:

(1) It has a *semantic*

(2) It can determine if it is up-to-date

(3) It knows how to bring itself up-to-date (the *update action*)

Property (1), the *semantic* of a *dependent resource* defines what the value of the resource represents.[8] Consequently, it also determines how the value can be (re-)calculated and therefore (together with the *value sources* of the *dependent resource*) determines the *update action* discussed below. Examples for different *semantics* are *ModelTransformation*, *ModelViewProjection*, or *TransparencyIndexBuffer*.

Like *value sources*, a *dependent resource* also has a set of *dependencies*.[9] To implement defining property (2) the *dependent resource* stores the set of its *dependencies* together with a corresponding set of version numbers. For every *dependency* of the resource this second set keeps the version number of the *dependency* that was effective at the time the resource was last updated. If there is a mismatch between one of these version numbers

[8] Instead of *semantic*, the term *type* could also have been used.

[9] How the *dependencies* of *value sources* and *dependent resources* relate to each other is explained below

and the current version number of the corresponding *dependency* then the resource is not up-to-date anymore. This set of version numbers can be seen as the composite version number of the *dependent resource*. More formally, the version of a *dependent resource* r from domain $\mathcal{R}$ with a set of *dependencies* $\{d_0, d_1, \ldots, d_n\}$ is given by the function $\text{version}_{\text{res}}$ which is defined as

$$\begin{aligned} \text{version}_{\text{res}}\colon\ & \mathcal{R} \to \mathbb{N}^n \\ & r \mapsto \big(\text{version}_{\text{dep}}(d_0), \text{version}_{\text{dep}}(d_1), \ldots, \text{version}_{\text{dep}}(d_n)\big) \end{aligned}$$

Property (3) of a *dependent resource* is that it must have a so-called *update action*. The *update action* is a procedure with the postcondition that the *dependent resource* is up-to-date afterwards. The concrete implementation of this procedure depends on a the resource's *semantic* but typically an *update action* stores references to a number of *value sources* and uses them to (re-)compute the value of the *dependent resource*. It is therefore more than just the code segment implementing the update logic: Two *update actions* using the same update code (because they stem from the same resource *semantic*) but different sets of *value sources* are not considered equal. [10] For example, the *update action* of a *ModelTransformation* resource would be to recompute the accumulated model-transformation from a fixed set of *transformation nodes* (implementing the *IValueSource* interface) on the path to the resource.

Apart from its postcondition, the *update action* must also fulfill another condition: Two subsequent invocations of the same *update action* must yield the same result in the form of setting equal state in the owning resource. That is, an *update action* should not set any external state that will have an effect on the next invocation. The result of computations in an *update action* must only depend on the *value sources* it references and state in the global environment that does not change during a render traversal.[11] This makes *update actions* predictable and easier to reason about.

```
// Something that is dependent on something else and knows if it is up-to-date
// and how to update itself. Any underlying objects or values managed by an IDependent
// must not be accessed if the IDependent is out-of-date. If unsure, call Update()
interface IDependent
{
   // Dependencies of this object
   IEnumerable<IDependency> Dependencies { get; }

   //Updates this object iff needed.
   void Update(Traversal traversal);
}

// A resource with dependencies
interface IDependent<out TResource> : IDependent
{
   // The underlying resource instance. Do not access unless the
   // dependent resource is up-to-date.
   TResource Resource { get; }
}
```

Listing 3.4: The *IDependent* Interfaces

[10] In practice this can be implemented by creating a *update action* class for each semantic and subsequently individual instances binding a specific set of *value sources*.

[11] This is similar to *uniform* and *varying* parameters in shader code, where *value sources* correspond to *varying* input (vertex data) and global state corresponds to *uniform* input.

In the source code *dependent resources* are implemented as two interfaces (Listing 3.4): *IDependent* and its generic specialization *IDependent<TResource>*. The base interface *IDependent* shows properties (2), the set of *dependencies*, and (3), the *update action*. Property (1), the *semantic*, is not explicitly represented in the interface but rather implicitly contained in the implementation of the *update action*.

The generic version of the interface also adds an accessor for the underlying resource, such as a *constant* or *index buffer*. The *dependent resource* concept is represented by two interfaces because the underlying resource is not needed for most purposes. Once a *dependent resource* and its wrapped resource instance are created, the base *IDependent* interface allows to manage it opaquely by providing a common supertype. The generic version of the interface is only used be the creator of the *dependent resource* which knows the type of the wrapped resource.

These three concepts—*dependencies*, *value sources*, and *dependent resources*—form the foundation of the *dependency system*. They define roles that can be taken by different objects in the scene graph by implementing the presented interfaces. Figure 3.11 shows a class diagram depicting the relationships in the system more formally.

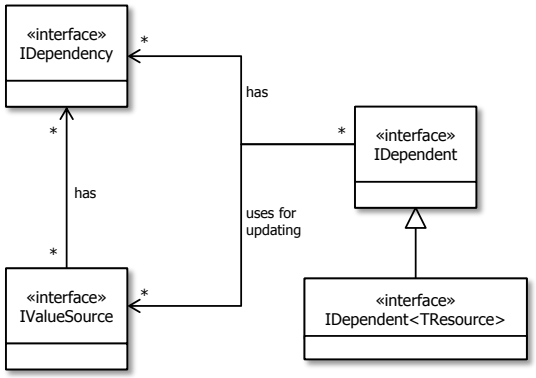
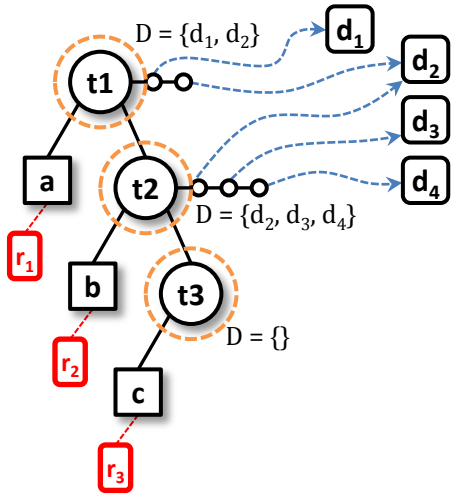


Figure 3.11: Class diagram showing the relationships between the base concepts of the dependency system.

To further illustrate the system, Figure 3.12 shows an example scene graph with dependency information. The scene graph consists of three *transformation nodes* (t_1, t_2, t_3) and three *geometry nodes* (a, b, c). The *transformation nodes* are *value sources*, each with a set of *dependencies*. Now for every *geometry node* a *dependent resource* r_i modeling its transformation can be constructed. The table shows the resulting sets of *value sources* and *dependencies* for each *dependent resource*.

Figure 3.12: A scene graph with dependency metadata



Dependent Resource	Geometry Node	Values Sources	Dependencies
r_1	a	(t_1)	$\{d_1, d_2\}$
r_2	b	(t_1, t_2)	$\{d_1, d_2, d_3, d_4\}$
r_3	c	(t_1, t_2, t_3)	$\{d_1, d_2, d_3, d_4\}$

The next chapter *Dependency-Aware Scene Graph Caching* will show in more detail how the *dependency system* can be integrated with the *scene graph caching system*. But first, some further properties of *dependent resources* and the system in general will be explored.

3.4.3 Resource Composition

With the properties of *dependent resources* given so far, it is possible to define a simple composition semantic for *dependent resources* with the result of such a composition again being a valid *dependent resource*. As described above, a *dependent resource* must have three properties: a *semantic*, a set of *dependencies*, and an *update action*. Consequently, the composite of several resources must have these properties too and they must behave as expected:

- The *semantic* must describe what the value of the composite resource represents.
- The set of *dependencies* must contain all *dependencies* of the value, meaning that
  1. the composite resource is up-to-date as long as none of its *dependencies* experiences a version change, and
  2. any version change renders the resource out-of-date, i.e. there are no *dependencies* that have no effect on the resource.
- The *update action* of the composite resource must bring the whole resource up-to-date.

As will be shown shortly, the above requirements can be fulfilled by combining a number of *dependent resources* in a *tuple-* or *record*-like manner, where every resource from the input set becomes an entry in the composite. Additionally, a function $\otimes_{\text{res}}$ can be defined that yields a valid composite from a set of input resources.

Let a *dependent resource* be defined as a tuple $(\sigma, D, \lambda) \in \mathcal{R}$, where σ is the *semantic*, D is the set of *dependencies*, and λ is the *update action* of the *dependent resource*. Then $\otimes_{\text{res}}$ is defined as: [12]

$$\begin{aligned} \otimes_{\text{res}}\colon \quad & \mathcal{R}^n \to \mathcal{R} \\ & ((\sigma_1, D_1, \lambda_1), \ldots, (\sigma_n, D_n, \lambda_n)) \mapsto \begin{pmatrix} \otimes_{\text{sem}}(\sigma_1, \ldots, \sigma_n) \\ \otimes_{\text{dep}}(D_1, \ldots, D_n) \\ \otimes_{\text{up}}(\lambda_1, \ldots, \lambda_n) \end{pmatrix} \end{aligned}$$

where the composition of *semantics* is a tuple of *semantics*

$$\otimes_{\text{sem}}(\sigma_1, \ldots, \sigma_n) = (\sigma_1, \ldots, \sigma_n)$$

defining a *product type* [24, p. 95] semantic.The composition of the *dependency* sets D_i is the union of the sets:

$$\otimes_{\text{dep}}(D_1, \ldots, D_n) = \bigcup_{1 \le i \le n} D_i$$

The composite resource is built from a set of sub-resources, each with its own set of *dependencies*. Since a resource as a whole is out-of-date if a part of the resource is out-of-date, the composite must inherit the *dependencies* of all of its parts:

$$\text{changed}(r_c) = \exists r_p(\text{changed}(r_p)), \quad r_p \in \text{parts}(r_c)$$

[12] The right side of the definition is given as a column vector only for formatting reasons. $\otimes_{\text{res}}$ yields a standard *dependent resource* of the form (σ, D, λ).

This is very similar to the change-semantics defined earlier for sets of *dependencies* (p. 41). The last function $\otimes_{up}$ combines the *update actions* of the sub-resources. An *update action* λ is defined here as a procedure that, using some computation rule with the *value sources* of a *dependent resource* as input, mutates the value of the resource to be consistent with the current state of the its *value sources*. The value of a composite resource can be seen as the tuple of the values of its sub-resources.

$$\text{value}(r_c) = (\text{value}(r_{p.1}), \ldots, \text{value}(r_{p.n})), \quad r_{p.i} \in \text{parts}(r_c)$$

Now, it is the task of the composite *update action* λ_c to bring r_c up-to-date. Being able to use the *update actions* $\lambda_{p.i}$ of the composite's sub-resources, $\lambda_c = \otimes_{up}(\lambda_{p.1}, \ldots, \lambda_{p.n})$ can be defined as

$$\lambda_c(r_c) = (\lambda_{p.1}(r_{p.1}), \ldots, \lambda_{p.n}(r_{p.n})), \quad r_{p.i} \in \text{parts}(r_c)$$

or in pseudo code notation which is better able to capture the impure, state mutating nature of *update actions*:

```
1 procedure Update(DependentResource r_c, TraversalState s_t)
2 begin
3 |   foreach r_p ∈ r_c.SubResources do
4 |   |   Update(r_p, s_t)
5 |   end
6 end
```

In the end $\otimes_{res}$ yields a composite *dependent resource* that exhibits all the properties of a regular *dependent resource*. Later in this work, it will become apparent how this is very useful for modeling *constant buffers* and other structured objects as *dependent resources*.

3.4.4 Resource Equality

What is a useful definition of *equality* for *dependent resources*? According to Leibniz' principle of the *identity of indiscernibles*, two objects are identical if they have the same properties. [20] More formally:

$$\forall F(Fx \iff Fy) \implies x = y \tag{3.1}$$

For two *dependent resources* $(\sigma_1, D_1, \lambda_1)$ and $(\sigma_2, D_2, \lambda_2)$ this would mean that they are equal iff all of their components are equal:

$$(\sigma_1, D_1, \lambda_1) = (\sigma_2, D_2, \lambda_2) \iff (\sigma_1 = \sigma_2) \wedge (D_1 = D_2) \wedge (\lambda_1 = \lambda_2) \tag{3.2}$$

In the context of the *dependency system*, however, the significant property of a *dependent resource* is its value, e.g. the state of the underlying *constant buffer*. In other words, the set of properties F which have to satisfy (3.1) can be reduced to properties pertaining to the state of the underlying resource; whereas the set of *dependencies* and the actual *update action* are mere bookkeeping appendage. Consequently, a more practical definition is that, from the users point of view, two *dependent resources* can be considered equal if their values are always perceived equal. Or, formulated in analogy to the *Liskov substitution principle* [29]:

Definition 3.1. A *dependent resource* r_1 is equal to a *dependent resource* r_2 if every occurrence of r_2 can be replaced with r_1 without changing the semantic of the program.

This definition is of practical impact, as will be shown at the end of this section. It is not immediately clear, however, whether two objects equal according to Definition (3.2) satisfy this desired condition of substitutability.

To prove that equality as defined in (3.2) is an instance of equality as given in definition 3.1 the contract established by the *IDependent* interface (Listing 3.4) has to be taken into account: The concept of *dependent resources* forbids to use its value unless it is up-to-date. Thus, a client using a *dependent resource* may only ever access the latest version of its value. Anything else is a breach of the usage protocol and can therefore be ignored as far as resource equality is concerned. In other words, only the current, most up-to-date state of two *dependent resources* has to be equal in order for the resources to be able to replace each other.

Theorem 3.1. *Two dependent resources* $r_1 = (\sigma_1, D_1, \lambda_1)$ *and* $r_2 = (\sigma_2, D_2, \lambda_2)$ *satisfy the substitutability condition as given by definition* (3.1) *if they are equal according to definition* (3.2)*.*

Proof.

(1) As both resource instances have exactly the same set of *dependencies* ($D_1 = D_2$) their *up-to-date* respectively *out-of-date* state are always equal at any given point in time. If r_1 becomes outdated so does r_2.

(2) Because λ_1 and λ_2 are equal it can be taken for granted that they are implemented by the same program code and reference the same *value sources*.

(3) A *value source* cannot not change state during an equality comparison of two *dependent resources*. If a *value source* changed stated, both resources referencing it would by definition be out-of-date and thus the equality comparison would be illegal in the first place.

(4) The computation of an *update action* must only depend on the referenced *values sources* as defined earlier (p. 42)

(5) From (2) $\wedge$ (3) $\wedge$ (4) it follows that the same code operating on the same constant *value source* will have the same effect on the respective *dependent resource*.

(6) As a consequence of (1) $\wedge$ (5), if both r_1 and r_2 are up-to-date their legally accessible state must be equal, which proves the proposition.

□

Having this concept of equality is very valuable because it allows for an important optimization: All equal *dependent resources* can be folded into a single instance. For example, a *dependent constant buffer* containing widely visible data such as view and projection matrices can be shared by a large quantity of *render jobs* in a completely automated fashion. Once the *dependency* metadata is in place, the dependency system can take care of reusing resource instances if it is legal

to do so. This optimization is reminiscent of *common subexpression elimination* [9] in programming language compilers. The next section will show how it can be directly incorporated in the construction process of *dependent resources*.

3.4.5 Resource Construction

In order to easily facilitate the creation of new *dependent resource* instances the *dependency system* provides an extensible resource factory that already implements the above mentioned resource sharing. To create a *dependent resource* the triple (σ, D, λ) is needed. The *semantic* σ has to be specified by the creator but the *dependency* set D and the *update action* λ can be extracted from the traversal state s_t once the desired resource *semantic* is known. The resource factory implements thus a partial function ν

$$\begin{array}{rccc} \nu\colon & \mathbb{S} \times S_T & \rightarrow & \mathbb{S} \times \mathcal{D} \times \mathcal{U} \\ & (\sigma, s_t) & \mapsto & (\sigma, D, \lambda) \end{array}$$

where D and λ can be computed as follows:

- The resource factory contains an internal registry of functions that can create an *update action* from the current traversal state s_t. These functions—from now on called *type-info factory*—are indexed by resource semantic. When the factory receives a request to create a new *dependent resource*, it uses the semantic σ it received as argument to select the correct *type-info factory*. It then invokes this function with traversal state s_t as argument. The *type-info factory* uses s_t to collect all *value sources* needed. All *value sources* encountered on the current traversal path are available in a list in s_t. Depending on the *semantic* this list is filtered to only the relevant sources. Having retrieved these, the *type-info factory* can create and return the *update action*.

  The *dependency system* already provides default *type-info factories* for common resource *semantics* such as *ModelTransformation* or *ModelViewProjection* but the registry can be extended with additional factories in order to support new resource *semantics*.

- The set of *dependencies* D of a *dependent resource* is—like the *update action*—determined by *semantic* and *value sources*. As the *type-info factory* already is specific to the given *semantic* and already has collected the relevant *value sources*, it can easily be extended to also compute the *dependency* set D which is the union of all sets of *dependencies* of its *value sources*:

  $$\mathrm{deps}_{\mathrm{res}}(r) = \bigcup_{v \in V} \mathrm{deps}_{\mathrm{src}}(v), \quad V = \mathrm{sources}_{\mathrm{res}}(r)$$

  This is because whenever one of the *value source* changes, also any resource dependent on it changes. Thus the relationship between a *dependent resource* and its *value sources* is the same as the relationship between a composite *dependent resource* to its component resources.

  $$\mathrm{changed}_{\mathrm{res}}(r) = \exists s(\mathrm{changed}_{\mathrm{src}}(s)), \quad s \in \mathrm{sources}_{\mathrm{res}}(r)$$

Having gathered all the information needed, the resource factory could now create a new *dependent resource* instance. Yet, as already mentioned, the factory implements automated resource sharing too avoid unnecessary redundancies. Using the now available triple (σ, D, λ) as resource identifier, the factory queries an internal cache for an instance equal to the requested resource. If there is a match then the cached resource is returned; else a new instance is created, added to the cache and then returned.

Note that *update actions* are compared indirectly, as there is no readily available way of determining whether two *update actions*, represented as C# *delegates* have equal implementations. Yet, since the *update action* is chosen based on resource *semantic* and may only use *value sources* as inputs, it suffices to compare those: If two *dependent resources* have equal *semantic* and equal sets of *value sources* then their *update actions* are also equal. Since the combination of *semantic* and *value sources* also determines the set of *dependencies* of a resource, the two of them being equal not only means that *update actions* are equal but the whole resource. Consequently, the information needed for implementing a cache lookup is resource *semantic* and the set of *value sources*. This pair $(\sigma, \mathrm{sources}_{\mathrm{res}}(r))$ can be considered the *signature* of the *dependent resource*. The *type-info factory* has already been extended to also return D in addition to λ and it is only consistent to let it also return $\mathrm{sources}_{\mathrm{res}}(r)$ which it already must have computed anyway. The final resource construction process is shown in Algorithm 3.2.

input : Semantic σ, TraversalState s_t
output: DependentResource r

1 $typeInfoFactory \leftarrow \mathrm{FactoryLookup}(\sigma)$
2 $typeInfo \leftarrow typeInfoFactory(s_t)$
3 $signature \leftarrow (\sigma, typeInfo.ValueSources)$
4 **if** $\neg(r \leftarrow \mathrm{CacheLookup}(signature))$ **then**
5 $\quad r \leftarrow (\sigma, typeInfo.D, typeInfo.\lambda)$
6 $\quad \mathrm{CacheInsert}(signature, r)$
7 **end**
8 **return** r

Algorithm 3.2: Creating a *dependent resource* with automated resource sharing.

3.4.6 Evaluation Order of Dependencies and Dependent Resources

In general, the update order of interdependent objects is determined by the structure of dependencies between the objects. If object a depends on object b—that is, if the value of object b is needed to compute the value of object a—then b must be updated before a.

$$update(a) \text{ uses } b \implies a \text{ depends on } b \implies update(b) \text{ before } update(a) \tag{3.3}$$

Otherwise, if a is updated before b then the value of a might grow stale when b is updated. This general principle also applies to the *dependency system*, however it is not immediately clear how this translates into a concrete implementation. This is what will be investigated in this section.

In the *dependency system* the three types of entities that can change state are:

Dependencies have a state consisting at least of their version number.

Value Sources have their value, the changes of which are indicated by their *dependencies*.

Dependent Resources have a value that is computed from their *value sources* and they also have *dependencies* which are used to determine whether the *dependent resource* needs to be updated.

There is one obvious consequence from this: *Dependent resource* must be updated after *value sources*, as the *value sources* are used to re-compute resource values. But how do *dependencies* fit into the picture? To answer this question, one needs to look at how and when the state of *dependencies* (i.e. their version number) is used and modified:

- The only time the version number of a *dependencies* can be modified is during the *dependency*'s *Eval()* method. Consequently, the *Eval()* method constitutes the «update» of the *dependency* and therefore—in accordance to (3.3) with the *dependency* being a here—any inputs used by *Eval()* must already have been updated. The input which can be used in an *Eval()* implementation is (1) state which is external to the scene graph (such as time or input device state) and (2) the *value source* the *dependency* describes. Concerning case (1) external state can be considered constant during a scene graph traversal. As a result, any external object or value used by *Eval()* can be considered up-to-date because there is no way for these objects or values to become «stale». For case (2) the situation is different: If *value sources* are given the chance to mutate their state when they are traversed they are not constant. Thus, in order not to violate (3.3), the *Eval()* method of a *dependency* must be executed after the *value source* it describes has been traversed (and possibly changed). This gives a lower bound on the point in time when *Eval()* has be executed.

  $$traverse(s_d) \text{ before } eval(d)$$

  where s_d is the *value source* described by d.

- Looking at when the version number of a *dependency* is used will provide an upper bound for the interval the version number must be updated in. Again (3.3) is used but this time the *dependency* takes the place of b. The role of a is taken by any *dependent resource* that contains the *dependency* in its set of *dependencies*. This is the case because the conditional *Update()* method of the *IDependent* interface is the only place where the *dependency* version number is used. Therefore, when a *dependent resource* r is updated, the version numbers of all *dependencies* in $\mathrm{deps}_{\mathrm{res}}(r)$ must be up-to-date. In other words, *dependency* version numbers, alongside *value sources*, are an input for *Update()*.

  $$eval(d) \text{ before } update(r_d)$$

  where r_d is the first *dependent resource* where $d \in \mathrm{deps}_{\mathrm{res}}(r_d)$

In summary, *dependencies* have to be evaluated between the traversing the *value source* they describe and updating the first *dependent resource* using this *value source* (and therefore containing the dependency in its *dependency* set).

In the prototype presented in this work, this is implemented by giving *value sources* the responsibility to evaluate all of their *dependencies*. This has the following implications:

- Any *dependencies* used by a *dependent resource* r will have been evaluated when r is encountered by the traversal. This is the case because of the way *dependent resources* are created:

  Proof.

  (1) A *dependent resource* r can only be created from *value sources* which are on the current traversal path at creation time.

  (2) When a *dependent resource* r is reached by a traversal, the same *value sources* have been traversed as when r was created, as every path gets its own *dependent resource* instance.[13] This is a consequence of resource sharing in the *dependent resource factory* (see *Resource Construction*).

  (3) The *dependency* set of r is the union of the *dependency* sets of the *value sources* of r.

  (4) As a consequence of (1) $\wedge$ (2) $\wedge$ (3) the *dependency* set of r can only contain *dependencies* that have already been evaluated by a *value source* when r is reached by the traversal.

  □

- If a *dependency* is shared by n *value sources* it will redundantly be evaluated n times. This should not be a problem for correctness, as a redundant evaluation should leave the version number untouched. If the evaluation was very costly, its implementer can add an «early exit» guard condition that checks whether the *dependency* has already been evaluated for this traversal.

- *Value sources* must not change during the traversal after its *dependencies* have been evaluated. If *value source* changes are only allowed while traversal is at the *value source* this cannot happen, as the *value source* has to re-evaluate its *dependencies* immediately after, even if it is traversed for the n-th time. The only possibility this can be a problem is when *value sources* are modified concurrently. This can lead to missed resource updates if the *dependency* evaluation still sees the old state of the *value source* but at the time of the resource update the new state of the *value source* is not recognized as the *dependency* does not indicate it.

[13] Note that this only holds as long as there are no structural changes which introduce new paths after creating *dependent resources*.

In conclusion, assigning the responsibility of evaluating *dependencies* to *value sources* fulfills the upper and lower bound conditions given above. The condition that *value sources* will always be up-to-date when *dependent resources* referencing them are encountered is also always met because of point (2) of the proof given above.

As a corollary of all of the above it can be stated that there are two equivalent ways of implementing the *Evaluate()* method in *IDependency* which could be termed «a priori» and «a posteriori» implementations.

A Posteriori The *dependency* evaluation is performed by comparing the just traversed and possibly changed *value source* to some reference value stored in an instance field of the *dependency*. This condition is reliant on the *value source's* state, and it can only yield the correct result *after* the *value source* has been updated.

$$\begin{aligned} P &\implies \text{changed}(s) \\ \text{changed}(s) &\implies \text{inc}(d.\text{Version}) \end{aligned}$$

A Priori The *dependency* evaluation does not use the *value source* directly but rather tests the external conditions that are known to always cause a change in the *value source*. The *Evaluate()* method never even has to look at the *value source* instance it describes.

$$\begin{aligned} P &\implies \text{changed}(s) \\ P &\implies \text{inc}(d.\text{Version}) \end{aligned}$$

Given the *dependency* and resource evaluation scheme described above both strategies will yield correct results. The *a posteriori* approach at first is more intuitive and may be easier to implement. However, the *a priori* approach will often allow more *dependency* instances to be shared between *value sources* and there can always be a *a priori* version of a *dependency* as there are always external causes to *value source* modifications (such as the passing of time or user input) which can be captured by the *dependency*.

3.5 Dependency-Aware Scene Graph Caching

This section will show how the *Dependency System* can be integrated with the *Caching Architecture*. The first section will show how using *Dependent Resources instead of Regular Resources* proves to be very beneficial for cache invalidation. The next section is on *Resource Management* and will describe various characteristics of different (dependent) resource types. The last section, *Updating Dependent Resources: The Dependency Index*, will show how a special data structure can help in updating *dependent resources* efficiently.

3.5.1 Dependent Resources instead of Regular Resources

As described earlier in *Cache Nodes & Render Caches* (p. 28), a *render cache* typically consists of an *instruction stream* and the resources this *instruction stream* references. Later, in the section on *Cache Invalidation* (p. 35) it has been shown that different kinds of scene graph modifications have different kinds of consequences on the *instruction stream* representing the scene graph. Most notably—as listed in the last line of Table 3.2—changing the value of an attribute node does not necessitate a structural change of the *instruction stream*: just the value stored in a resource referenced by the stream has to be updated.

As it turns out, these kinds of modifications are the most frequent. Examples range from camera movement and object transformations to material and lighting changes. Even direct modifications of index data (e.g. sorting primitives in transparent geometry) and vertex data (e.g. object morphing) fall into this category. Only structural modifications to the scene graph cannot be handled by simple resource content updating.

Not entirely coincidentally, smart resource updating is exactly what the *dependency system* was developed for—and conceptually it can be integrated with *render caches* rather easily. By using *dependent resources* instead of regular ones when building *instruction streams*, a whole class of scene changes does not invalidate the cache anymore. Instead—by bringing all *dependent resources* up-to-date before executing the *instruction stream*—the *render cache* stays valid. Updating the *dependent resources* can be considered an «incremental update» of the *render cache* and, if implemented correctly, can be much more efficient than rebuilding the cache from scratch. The next section will elaborate on the various resource types in the scene graph system.

3.5.2 Resource Management

Managing resources is always a very important part of any performance oriented system. For the *scene graph caching system* this is no different. The following list of graphics *API* resources have to be handled by it:

1. Vertex & Index Buffers
2. Vertex & Index Buffer Bindings
3. Shader Input Layouts
4. *GPU* Programs

5. Constant Buffers

These resource types play very different roles in the graphics pipeline and not all of them represent 3D content. Buffer bindings and shader input layouts are better characterized as bookkeeping data and can be generated automatically. These resources merely being implementation details also do not need to be modeled as *dependent resources* as will be shown below. What follows is a short description of each resource type together with an analysis of its interaction with the *dependency system*, its ownership and lifetime, and how an instance of the resource is constructed in the context of the caching system.

Vertex & Index Buffers

In terms of size these resources hold the majority of the scene data. They contain a representation of geometry information described in *geometry nodes* in a vertex-based format natively consumable by the graphics hardware. Since these buffers contain such a large amount of data it is highly desirable to share them between the *immediate mode* renderer and the caching system. Yet, the *immediate mode* renderer does not have any concept of change. It just uses buffer data as is when it encounters it during the *render traversal*. Changes, such as morphing and sorting of indices, are applied without the renderer knowing about it. This can lead to two problems:

- When some entity modifies buffer contents for the *immediate mode* renderer an inconsistency between buffer contents and the dependency information describing it is introduced. As a consequence, a *render cache* using the same buffer will wrongly and redundantly execute the buffer's possibly quite expensive *update action*.

- The logic for updating buffers has to be duplicated and coordinated between the *immediate mode* renderer and *render caches*, adding another source of inefficiencies and inconsistencies.

In order to avoid these pitfalls the prototypical implementation always represents *index* and *vertex buffers* as *dependent resources*. Both, the *immediate mode* renderer and the caching system use *IDependent.Update* to handle buffer modifications. For static geometry no performance overhead is introduced, as it will have an empty *dependency set* and hence never has to be checked for modifications.

Dependence In the AARDVARK scene graph environment *geometry nodes* reference *VertexGeometry* objects which contain vertex data in a separate array for each attribute (such as position, normal, or color). Each of these has a string key. For the prototypical implementation of the caching system *VertexGeometries* have been extended to contain a mapping of attribute names to *dependent resource semantics*. This mapping specifies which kind of *dependent resource* is created for a given attribute array. From the viewpoint of the *dependent resource* the attribute array represents a *value source*. Additional *value sources* will depend on the given *semantic*. For example, a *TransparencyIndexBuffer* will reference the camera as another *value source*.

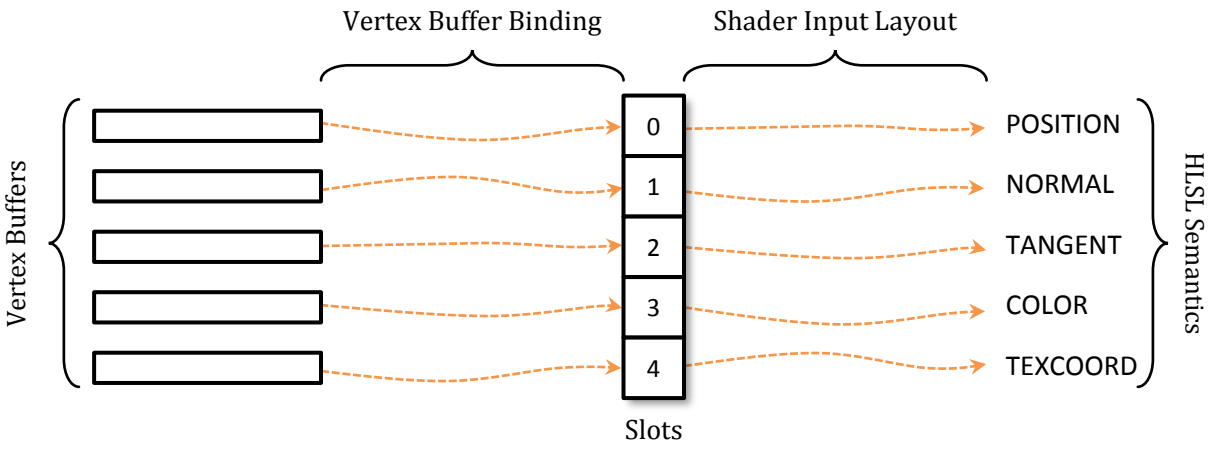


Figure 3.13: Vertex buffer binding, slots, and shader input layout

Ownership, Lifetime & Construction Because ownership of an *index* or *vertex buffer* can be shared between several *geometry nodes* and *render caches* a reference counting scheme is employed to manage resource lifetime. As described above (p. 49) a *dependent resource* can be identified by its *signature* $(\sigma, \text{sources}_{\text{res}}(r))$. Whenever a *dependent vertex buffer* needs to be created the *dependent resource factory* is invoked which creates the *signature* from the given traversal state and the *semantic*. It will then use the *signature* to query its internal cache for an equivalent resource. If it finds one, it will increment the resource's reference count. Else it will add a new instance to its cache. *Render caches* and *geometry nodes* use the factory to acquire and release references to *dependent index* and *vertex buffers*.

Shader Input Layouts, Index & Vertex Buffer Bindings

A *vertex buffer binding*, in analogy to *Direct3D 11*, maps certain vertex buffers to so-called «slots» (with a slot just being a non-negative index). The other half of the mapping are *shader input layouts* which map slots to *HLSL semantics* [13]. This way, the graphics driver knows which vertex buffer to bind to which input parameter in the shader code. Figure 3.13 illustrates the concept.

Index and *vertex buffer bindings* are represented by one interface in the graphics backend used for the prototypical implementation: *IVertexBufferBinding*. An instance of *IVertexBufferBinding* may also contain an optional reference to a *index buffer*. In the AARDVARK scene graph toolkit the shaders to be used for a subgraph are specified using so-called *surface nodes*. A *surface* is a pair of vertex and pixel shader, plus an optional geometry shader. In other words, the *surface* defines active *GPU* programs. Additionally it also provides some reflection data about itself. Among other things it can be queried for the varying (~*vertex buffers*) and uniform inputs (~*constant buffers*) it requires. This information is extracted from the *HLSL* source file of the shaders assigned to the *surface*. For varying inputs a mapping from *HLSL* semantic to a string key has to be provided at the beginning of the *HLSL* file. This key must specify under which

name the *vertex buffer* data is stored in the geometry data structure in the scene graph. AARDVARK defines an extensive set of standard keys (such as «Positions», «Normals», «Colors», etc) to ease using this system. With this metadata available, *vertex buffer bindings* and *shader input layouts* can be created automatically by the scene graph system.

Dependence The *shader input layout* depends on the *surface* it was created for. If the number or order of varying inputs of the vertex shader changes, the input layout has to be adapted. *Vertex buffer bindings* depend on the input layout and thus, by extension, also an the *surface*. If a changed input layout maps a given semantic to a different slot, the *vertex buffer binding* must remap the buffer in question to the new slot.

In theory buffer bindings and input layouts could be modeled as *dependent resources*. Their only *value source* would be the *surface* active for them and whenever the *surface* changes, their respective *update actions* could re-run the automated mapping algorithm described below. However, as these kinds of changes are rather infrequent at runtime, this functionality is not implemented in the prototype. As a result, *vertex buffer bindings* and *shader input layouts* are treated as being constant and changes to the *vertex shader* which change the input mapping are not supported at runtime.

Ownership & Lifetime The ownership of *vertex buffer bindings* and *shader input layouts* depends on whether they are used within a *render cache* or not. In the cached scenario all buffer bindings and input layouts are created when the cache is built and subsequently they are also owned by the containing *render cache*. Every *render cache* has its own set of buffer bindings and input layouts which does not intersect with the resource set of any other *render cache* or the resources used by *immediate mode* rendering. This makes lifetime management rather easy: The bindings and layouts live as long as their owning cache does; disposing the cache will also dispose these resources.

As a side effect of this ownership setup and the resources' immutability, it is also easy to share input layouts and buffer bindings *within* the cache. If two *geometry nodes* happen to reference the same geometry data under the same *surface* they can use the same *vertex buffer binding*.

For *immediate mode* rendering, *vertex buffer bindings* are owned by the *geometry node* the vertex data of which they reference. Every *geometry node* holds a small data structure specific to each *surface* the node is reachable from. This data structure stores the *vertex buffer binding* the content of which depends on the pair $(\textit{surface}, \textit{geometry})$. The binding with key (s_1, g_1) is disposed when geometry node g_1 becomes unreachable from *surface node* s_1.

For *shader input layouts* it is different: There is always only one *shader input layout* for each surface.[14] The same input layout may even be reused for two different *surfaces*. As a consequence, the overall number of input layouts needed for the whole scene graph is likely to be rather small. For this reason, *shader input layouts* are considered globally shared resources, owned by the scene graph as a whole. They live as long as the scene graph does.

[14] In theory, there could be as many input layouts per *surface* as there are slot permutations. However, normally, there is little reason to generate different input layouts for the same surface.

Construction *Vertex buffer bindings* and *shader input layouts* are created together by an instance of the *BindingFactory* class. On creation the factory can be configured to reuse binding and input layout instances if possible. The factory instance thus serves as a resource sharing context. See Algorithm 3.3 for a pseudocode listing of the combined buffer binding and input layout creation procedure.

Every *render cache* has its own factory instance (with reuse of both bindings and input layouts enabled) and consequently all *vertex buffer bindings* and *shader input layouts* are optimally reused within a cache automatically. The *BindingFactory* also has a *Finish()* method which closes the factory instance and returns a special *disposer* object that allows to dispose all resources created by this factory instance at once. This way the *render cache* can take ownership of bindings and input layouts when it is completely built. When the cache is disposed, it will use the disposer object to also free these resources.

GPU Programs

GPU programs such as *vertex, pixel* and *geometry shaders* are held by *surface nodes* in the scene graph. For the rendering backend to actually use them they must be instantiated as *API* resources.

Dependence In the current implementation, *GPU* programs are immutable once they are loaded. Thus they have no *dependencies* and are not implemented as *dependent resources*. Yet, in the future they might be modeled as such in order to enable automatically reloading *GPU* programs when they are changed on the disk. However, the implications of changing shaders at runtime can be complicated, especially when the number or types of varying and uniform inputs are modified. Such changes may necessitate adapting quite a few other resource instances, such as *vertex* and *constant buffers*. The exact implications of such modifications have not been investigated in detail.

Ownership & Lifetime & Construction *GPU programs* are managed by a reference counting scheme to facilitate shared ownership semantics. *Geometry nodes* and *render caches* alike register their usage of a surface node which automatically instantiates and disposes *GPU program* resources as needed. See also the following description of *constant buffers* for additional information.

input : Surface s, GeometryData g
output: IVertexBufferBinding vbb, IShaderInputLayout il

```
1  begin
2      ilDef ← new Dictionary[string → int]
3      vbbDef ← new Dictionary[int → string]
4      for slot ← 0 to s.Inputs.Count do
5          input ← s.Inputs[slot]
6          ilDef[input.HlslSemantic] ← slot
7          vbbDef[slot] ← g.VertexBuffers[input.GeometryKey]
8      end
9      il ← GetInputLayout(ilDef, s)
10     vbb ← GetVertexBufferBinding(vbbDef)
11 end
12 function GetInputLayout(Dictionary[string → int] def, Surface s)
13 begin
14     if (reuse shader input layouts) then
15         cache ← GetCacheFor(s)
16         if ¬(cachedInputLayout ← cache.LookUp(def)) then
17             cachedInputLayout ← new ShaderInputLayout(def)
               cache[def] ← cachedInputLayout
18         end
19         return cachedInputLayout
20     else
21         return new ShaderInputLayout(def)
22     end
23 end
24 function GetVertexBufferBinding(Dictionary[int → string] def)
25 begin
26     if (reuse vertex buffer bindings) then
27         hash ← CreateHashFor(def)
28         if ¬(cachedBinding ← CacheLookUp(hash)) then
29             cachedBinding ← new VertexBufferBinding(def)
               CacheInsert(hash, cachedBinding)
30         end
31         return cachedBinding
32     else
33         return new VertexBufferBinding(def)
34     end
35 end
```

Algorithm 3.3: Creating a *vertex buffer binding* with matching *shader input layout*

Constant Buffers

Constant buffers are the most frequently changing resources. Every camera movement means that at least one *constant buffer* instance needs to be changed (because of *view*, *view-projection*, *model-view-projection* matrices), every transformation change of an object is normally reflected by different values in a *constant buffer*. The same is true for changes in lighting and many material properties. As a consequence it is important that *constant buffer* handling is implemented efficiently.

Constant buffers can be seen as a record with a number of fields. From the viewpoint of the *dependency system* each field is a *dependent resource* with a given *semantic* (like *ModelTransformation*). The *constant buffer* as a whole is a composite *dependent resource* as described in *3.4.3 Resource Composition*.

Dependence Two usage cases of *constant buffers* have to be distinguished here. Their usage within a *render cache* and their usage by the *immediate mode* renderer. From the viewpoint of the later there is no *dependency system*. It only uses regular *constant buffer*.

Within the caching system *constant buffers* are treated as *dependent resources* and all the rules described in *3.4 Dependency System* apply. The *dependency* set of each field is the union of the *dependency sets* of all *value sources* used by the *update action* of the field (see p. 41). The *dependency set* of the *constant buffer* is the union of the *dependency* sets of all its fields (see p. 45).

Ownership & Lifetime Again the distinction between cached and non-cached usage has to be made. In the non-cached case, *surface nodes* in the scene graph hold *constant buffer* instances needed by their GPU programs. For every *constant buffer* that is contained in a *vertex*, *fragment*, or *geometry shader* of the *surface*, the node keeps exactly one *constant buffer* instance with the required layout. This instances are reference counted together with the GPU programs. When the first reference to the *surface* is registered *constant buffers* and GPU programs are allocated and when the last reference to the *surface* is cleared they are deallocated.

In the cached usage case every *render cache* gets its own set of *constant buffers*. The buffers are not shared between two caches and are not shared between a cache and the *immediate mode* renderer. The *constant buffers* in a *render cache* are created when the *SlimSG* is created by the *ExtractCachingDataTraversal* and are then owned by the cache that is created from the *SlimSG*. When the *render cache* is disposed, so are its *constant buffers*.

Construction *Constant buffers* use the construction process described in *Resource Construction*. They are created in *ICacheable.ExtractCachingData()* using a *dependent resource factory* instance owned by the *ExtractCachingDataTraversal* object. As this factory implements the resource sharing technique described before every distinct *dependent resource* gets its own instance while *constant buffers* which are equal according to Definition (3.1) are represented by the same instance. With the right factoring of data into different buffers this can dramatically reduce the number of buffer instances needed.[15]

[15]The «right factoring» in this case means to group uniform data according to their update frequency. This is already advocated in general, for example in the NVIDIA GPU Programming Guide [16, p. 46]

In order to efficiently check whether there already is an equal *constant buffer* instance available to be reused, the factory uses *dependent resource signatures* (p. 49). These signatures act as key into a dictionary of already created resources. By creating a hash value for signatures searching for an equivalent *constant buffer* can be performed in approximately $\mathcal{O}(1)$.

Like with the *BindingFactory*, once the construction of all resources for a *render cache* is finished, the *dependent resource factory* can be closed. This provides a disposer object which allows the *render cache* to take ownership of all *dependent resources* created by the factory.

Updating Because *constant buffers* are potentially updated very often, special attention should be given to the implementation of this process. Because re-computing values of single fields can be quite expensive (like accumulating a current model transformation) redundant updates should be avoided. For composite *dependent resources* a single sub-resource being out-of-date implies that the whole composite is out-of-date. But the opposite is not true: The composite as a whole being out-of-date does not imply that all of its sub-resources are out-of-date. Just (at least) one of them. Consequently a possible optimization for updating *constant buffers* is to only re-compute value of fields which are not up-to-date. This means that each field must be check individually, which is easily possible because the *dependency set* of each field is known at resource construction time and can be retained with the buffer. The naive implementation is shown in Algorithm 3.4.

input : DependentConstantBuffer b

```
1  for i ← 0 to b.FieldCount do
2  |  updateNeeded ← false
3  |  field ← b.Field[i]
4  |  for j ← 0 to field.DependencyCount do
5  |  |  if field.Dependency[j].Version ≠ field.RefVersion[j] then
6  |  |  |  updateNeeded ← true
7  |  |  end
8  |  end
9  |  if updateNeeded then
10 |  |  field.Update()
11 |  end
12 end
```

Algorithm 3.4: Naive implementation of updating a *dependent constant buffer*

However, this approach would necessitate storing the *dependency set* and corresponding version numbers for each field individually. Moreover the nested loop in the algorithm leads to a complexity of $\Theta(n \cdot m)$ and the version comparisons cause a lot of possibly redundant indirection. In practice none of this is a crippling problem but a more efficient implementation can be provided quite easily.

When creating a *constant buffer*, create a union of the *dependency sets* of its fields. This is in accordance to the rules for composite *dependent resources.* Then for each *dependency* in the united set store a bit mask that indicates which fields the *dependency* affects. Fields have a fixed index within a *constant buffer*. If *dependency* d contained in the *dependency set* of field

f_i then set bit i in the mask of d. Now it is known which fields need to be updated if d's version number has changed. These bit mask for every *dependency* is stored with the *constant buffer* when it is created.

To get the complete set of fields that need to be updated at a given point in time simply iterate over all *dependencies* and—if a version has changed—combine their bit masks with a *bitwise or* operation. The result is a bit mask containing the indices of all fields that need to be updated. Iterate over fields and, if their bit is set, update them. See Algorithm 3.5 for a full listing.

input: DependentConstantBuffer b

```
1  for i ← 0 to b.DependencyCount do
2  |   fieldMask ← 0
3  |   if field.Dependency[i].Version ≠ b.RefVersion[i] then
4  |   |   fieldMask ← fieldMask | b.FieldMask[i]
5  |   end
6  end
7  for i ← 0 to b.FieldCount do
8  |   if bit i is set in fieldMask then
9  |   |   b.field[i].Update()
10 |   end
11 end
```

Algorithm 3.5: More efficient implementation of updating a *dependent constant buffer*

Commonly, the number of fields in a *constant buffer* is not greater than the number of bits in a machine word. In this case bit masks can be represented by a single integer variable which allows for efficient bitwise operations and costs little memory. For this reason, the prototype contains a specialized implementation of *dependent constant buffers* with less than 32 fields. For larger buffers—which are rare—the algorithm is still better than the naive implementation but a bit less efficient.

Overall, this alternative approach has many benefits. *Dependency sets* and corresponding version numbers do not need to be stored for each individual field without field values having to be recomputed unnecessarily. The number of indirections is reduced as each *dependency* instance is only accessed once per update. And the nested loop has been replaced by two regular loops leading to a complexity of $\Theta(n + m)$.

3.5.3 Updating Dependent Resources: The Dependency Index

As has been described before, the *Render()* method of a *render cache* must update all *dependent resources* used in its *instruction stream* before executing the stream. In order do make this functionality easily available to implementers of *IRenderCache* it has been encapsulated in a standardized data structure: the *DependencyIndex*.

The straightforward implementation of updating a set of *dependent resources* is very similar to Algorithm 3.4 for updating a *constant buffer*: Iterate over all resources and call their *Update()* method. As the *IDependent* interface requires the *Update()* method to only update the *dependent resource* if needed there will be no unnecessary resource value computations. However, many

resources that may not need to be updated will still have to check their *dependencies*. The question is: Can the number of up-to-date checks be reduced? To answer this question, we analyze the nature of the given problem. Within a cache there is a fixed number of *dependent resources*, each with a fixed set of *dependencies*. When the cache is rendered, it wants to know the set of *dependent resources* that really need an update. This set is specified as those resources the *dependency set* of which contains a *dependency* with a changed version number. This situation is very reminiscent of the setup of *constant buffers* and in fact the set of *dependent resources* within one *render cache* can again be considered a composite *dependent resource* as described in *Resource Composition*.

In contrast to *constant buffers*, this time the total number of sub-resources can go well into the thousands. The total number of *dependencies*, however, is not expected to be that high as many *dependent resources* have an empty set of *dependencies* and one *dependency* instance is expected to be shared by many resources. It is our estimation that in a *render cache* with n *dependent resources* there are (in the average case) $\log(n)$ *dependency* instances.[16]

The query to be performed is checking for overlaps in sets of *dependencies*: First, find the set of *dependencies* which have changed since the last update. Second, find all resources the *dependency* set of which overlaps with the change set. The first step can be implemented by creating the union set of all *dependencies* in the *render cache* and keeping a corresponding version for each. When the cache is to be rendered, iterate over the union set and collect all *dependencies* the version number of which does not match the corresponding version number stored. For the second step an *inverted index* [7] of *dependencies* to their resources can be built for efficient set intersection testing.

The algorithm for finding all *dependent resources* that need to be updated can then be specified as follows:

input : List of *Dependencies* ds, List of *Dependency reference versions* vs, InvertedIndex $index$
output: Set of *dependent resources* to be updated rs

```
1 rs ← ∅
2 for i ← 0 to ds.Length do
3     if ds[i].Version ≠ vs[i] then
4         resourcesWithThisDependency ← index[ds[i]]
5         merge resourcesWithThisDependency into rs
6         vs[i] ← ds[i].Version
7     end
8 end
```

3.5.4 A Note on Resource Update Order

Section 3.4.6 laid out the rules for the update order of *value sources*, *dependencies* and *dependent resources*. Moving along the normal scene graph traversal, these rules are met. Yet, the

[16] At the moment this is just a guess which may have to be revised after testing the system with more real world data.

caching system changes the traversal order: A cached subgraph is not traversed anymore. As a consequence, care has to be taken to keep update order correct. Before the *instruction stream* and *dependent resources* in a *render cache* (representing the leaves of the cached subgraph) are used, the *render cache* also has to update/evaluate the *value sources* and *dependencies* between *cache node* and leaves.

For this reason, *render caches* «pull up» *dependencies* and *value sources* from the cached subgraph and update them before using the *DependencyIndex* to update *dependent resources*. However, *value sources* do not have a standardized way of being updated; unlike *dependent resources* they have no *Update()* method known to the dependency system. Changes to *value sources* are initiated (1) before the render traversal or—conceptually rather uncleanly—(2) in the *Render()* method, called by the traversal when encountering them. As a cached *value source* is excluded from traversal the caching system introduces the restriction that only option (1) is used for mutating *value sources*. This keeps the update order intact in any case by lifting the lower bound on *dependency* evaluation described on page 50.

The best way, however, to solve this problem would be to unify the concept of *value sources* and *dependent resources*. *IValueSource* is already a super type of *IDependent* and *dependent resources* can act as *value sources* without a problem.[17] Having only *dependent resources* would make data dependencies even more explicit and evaluation order would always automatically be correct. This change in concept would mean moving from a predominantly imperative scene graph evaluation model to an entirely declarative, incremental evaluation model. For the current implementation this was too radical a shift because of the additional integration costs with the existing code base. However, a more fundamental approach might prove interesting for future work.

[17] This is a fact that will be exploited by some of the optimizations presented in the next section

3.6 Optimizations

The *scene graph caching system* as described so far—using *instruction streams* and the *dependency system*—can be augmented by quite a few optimizations in various areas. This section will describe each of them in detail.

3.6.1 Removal of Redundant Instructions

The simplest way of creating an *instruction stream* from a list of *render jobs* is to iterate over the list and for each job create a sequence of instructions which sets the full pipeline state, from *primitive topology*, GPU programs, *shader input layout*, and textures to *vertex*, *index*, and *constant buffer bindings*. Although semantically correct, an *instruction stream* generated this way can contain a significant percentage of redundant instructions. Most of the time the correct *primitive topology*, GPU programs, et cetera are already set from a previous instruction in the stream and setting it to the same value again poses a waste of computational resources.

Fortunately, it is rather easy to omit redundant instructions already while generating the *instruction stream*. Every *render job* contains the full render state that needs be set before its *draw call* can be executed. As a consequence the active render state at the current end of the *instruction stream* is known to be equal to the render state specified in the *render job* that was last processed by the code generator. Thus, the code generator can determine which instructions need to be generated for the current *render job* by simply looking at its immediate predecessor. This is implemented by simple conditional statements of the form

if *current.PrimitiveTopology* $\neq$ *previous.PrimitiveTopology* **then**
  | **yield new** SetPrimitiveTopologyInstruction($current$.PrimitiveTopology)
end
if *current.VertexShader* $\neq$ *previous.VertexShader* **then**
  | **yield new** SetVertexShaderInstruction($current$.VertexShader)
end
...

This kind of optimizations has already been described in the *Related Work* section. Durbin et al use it on their *streamlined arrays* [18] where they call it a «peephole optimization». Also, many scene graph toolkits like *Performer* [40, Section 2.2.3] and *OpenSceneGraph* [39] will do a similar filtering of *API* calls at runtime.

3.6.2 Super Instructions

Superoperators [36] or *superinstructions* [19, p. 20] are an optimization originating from byte-code interpreters. Common sequences of atomic byte-code instructions are folded into a larger, semantically equivalent *superinstruction*. When used in an interpreter, this has various benefits, such as smaller code size, less instruction dispatch overhead and improved branch prediction. For *instruction streams* used by the caching system, there are similar benefits, mainly the reduction of the «instruction dispatch» *INativeInstruction.Execute()* calls.

3.6.3 State Sorting

Sorting *render jobs* in order to reduce the number of render state changes is often mentioned as an important optimization. In the *Related Work* section it has already been described as *Pull up costly state changes* (p. 6), the *NVIDIA GPU Programming Guide* recommends it [16, p. 20], and various scene graph toolkits such as *Performer*, *OpenSceneGraph*, and *OpenSG* support it as a runtime feature (see Rohlf and Helman [40, Section 3.1.3], Osfield [39], and Reiners [38]).

In this context, the optimization is implemented by sorting the list of *render jobs* before generating an *instruction stream* from it. The sorting is performed in two phases:

1. The list of *render jobs* is grouped by *render target*, then by the tuple $(vertex\ shader, fragment\ shader, geometry\ shader)$. This yields a number of shorter, disjunct lists of *render jobs* within which the most expensive states are not changed anymore. The grouping can be implemented rather efficiently by using hashing techniques.

2. Each of the groups generated in the previous step are now sorted to further reduce the number of state changes. This is done by defining a distance metric between *render jobs* and then solving the *traveling salesman* problem within each group. The more state changes are required for moving between two *render jobs* the greater the distance assigned by the metric will be. If the render state of two jobs is equal, the distance is zero. If two *constant buffer* bindings need to be changed it will be greater than if only one needs to be changed, and so on. Following the shortest path connecting all *render jobs* will result in the least state changes. Mind, however, that any path will yield correct results and solving the *TSP* only approximately is sufficient to reduce the number of state changes significantly.

3.6.4 Overdraw Sorting

Most modern graphics hardware supports an optimization called *Z culling*. [16, p. 43] If a pixel is determined not to be visible because of its depth value, it is discarded from the rendering pipeline before the *fragment shader* stage. This way the GPU does not need to compute the color value of the pixel (done by the *fragment shader*) and does not need to blend it into the *render target*. The performance gain of this optimization will depend on the complexity of the *fragment shader*, the size of the *render target*, and the percentage of pixels that do not need to be computed because of the optimization. Only the last factor can be influenced by the caching system. By drawing geometry closer to the camera first, the chance increases that there are already pixels in the *depth buffer* which occlude later drawn geometry. This is called *front-to-back rendering*. *NVIDIA's GPU Programming Guide* recommends to prioritize *front-to-back rendering* over sorting by state. [16, p. 20] To draw front-to-back the geometry must be sorted by distance to the camera position. However, there are two problems which need to be solved when implementing this:

(1) Geometry data is hard to sort. A geometry node can represent arbitrary shapes, intersecting themselves and shapes of other nodes. Often shapes would have to be split to allow an accurate sorting, which is a time-consuming process.

(2) Camera and object positions can change freely every frame. There is no static sort order that can be determined beforehand. An *instruction stream* in a *render cache* must be dynamically adjusted to the current scene configuration.

This problems can make *front-to-back rendering* seem unfeasible in the context of the caching system. Fortunately, both problems can be solved by relaxing the constraints a bit. As long as the geometry rendered is not transparent, the *depth buffer* will always guarantee a correct output image, independent of the drawing order. That means, drawing the geometry in arbitrary order will have the same result as drawing accurately sorted or just roughly sorted. The only difference is the number of pixels that can be culled early from the pipeline. This provides some leeway because a completely correct sort order does not have be guaranteed at any time.

As a result problem (1) can be solved by partly ignoring it. Facilitating an entirely accurate spatial sorting would be too expensive while an approximated sorting can also bring the desired outcome. For this reason, the caching system will sort geometries by the distance from the camera to the centroid of the geometries vertices. The centroid is modeled as a *dependent resource* so it can be kept up-to-date automatically. This heuristic will work well for small, compact geometries where the centroid poses a good approximation of geometries' spatial configuration. For larger shapes the sorting will be more random. This can negatively affect the number of pixels profiting from *Z culling* but—as already mentioned above—will not compromise the output image.

The solution for problem (2) is more complicated and consists of several parts:

(a) Determining when to re-sort the *instruction stream*,

(b) integrating the re-sorting process with the render traversal, and

(c) developing an algorithm that can actually re-sort an *instruction stream* without breaking its semantic.

The solution to (a) again profits from the relaxed accuracy constraints. Taking into account every movement of every object and the camera would very probably result in having to re-sort the *instruction stream* every frame. At the same time the sort order might not even change due to the temporal coherence of many scenes. Also, we expect the benefit of an accurate sort order over a roughly correct sort order to be negligible. As the geometries drawn are distributed over the two dimensional *render target*, sorting just by camera distance can produce arbitrarily different sort orders for objects not overlapping in screen space without changing the number of pixels profiting from *Z culling*.

As a consequence we use a heuristic again: Only the camera position is used to determine when re-sorting is advisable and only if the camera moves by a certain distance will re-sorting be triggered. This predicate is implemented as a *dependency*, looking very much like Algorithm 3.1 (p. 40). The sort order of an *instruction stream*, same as geometry centroids, is implemented as a *dependent resource* with the camera as its *value source*. When the render traversal reaches the *render cache* containing a re-sortable *instruction stream* the sort order will be updated by the *dependency index*. If the *dependency* indicates that the order is out-of-date, the sort order resource's *update action* will trigger re-sorting the *instruction stream*.

This also already illustrates the solution for task (b). For large *instruction streams*, however, sorting can take a long enough time to cause a visible stuttering of an animated scene. To rectify this issue, re-sortable *instruction streams* are implemented by a specialized class: *ReSortable-InstructionStream*. This class stores two copies of the original *instruction stream*. One always contains the original sorting, while the second is front-to-back sorted. When the sort order is out-of-date, the second stream is sorted by a worker thread in the background. If the render traversal reaches the cache while the second stream is locked by the sorting process it can just fall back to executing the first stream. This effectively eliminates the stuttering while providing front-to-back rendering most of the time.

Task (c), making *instruction streams* sortable, is the hardest part of this optimization. To find a starting point for this undertaking, one needs to understand the semantic and internal structure of *instruction streams*. The purpose of an *instruction stream* is to issue commands to the rendering backend, which ultimately draws an image. This image is what matters and any transformation done to an *instruction stream* which does not alter the output image will in this context be considered to yield an equivalent *instruction stream*. The re-sorting algorithm to be developed here must be such an *equivalence transformation*.

To ensure that a transformation does not change the effect of an *instruction stream* its inner workings must be analyzed. The rendering backend can be configured by setting different states (already referred to previously as the *render state* or *pipeline state*). Also, *draw calls* can be issued to the rendering backend which initiate the actual drawing process using the current state. A *draw call* plus the state that is active when it is issued has previously been called a *render job*. A *render job* is self-contained, meaning that it does not dependent on any *render jobs* previously processed by the rendering backend. This is the reason why the list of *render jobs* can be arbitrarily ordered by the *state sorting* optimization.[18]

Yet, once the list of *render jobs* is transformed into an *instruction stream* things are more complicated. A state set by an instruction stays active as long as another instruction overwrites it. Thus, the «scope» of an instruction—the range within the *instruction stream* which is influenced by the instruction—can span until the end of the whole stream and over multiple *draw calls*. With the *remove redundant instructions* optimization activated this is the common case, as this optimization uses exactly this property to shrink the stream. What previously was a self-contained *render job* is now one *draw call* and all the instructions that define the render state at the time when the *draw call* is executed. These state-setting instructions can be anywhere in the stream as long as they are in front of the *draw call*. As a consequence instructions cannot be freely moved around. Figure 3.14 shows an example of *render jobs* mapped to an *instruction stream* and the scopes of the various instructions.

In order to still be able to re-sort the stream the algorithm proposed here will segment the stream into ranges which represent as much of a *render job* as possible. Then it will determine which of these segments can be moved around without changing the stream's semantic. These segments are called *atomic sections* as the instructions within them will always be moved together when the stream is sorted. An *atomic section* always ends with a *draw call*. An *atomic section* has a *local state* which is defined by the state-setting instructions within it. Commonly

[18] Of course this is only true for non-transparent geometry where the *depth buffer* will take care correct visibility. For transparent geometry the drawing order is fixed and neither *state* nor front-to-back sorting can be done.

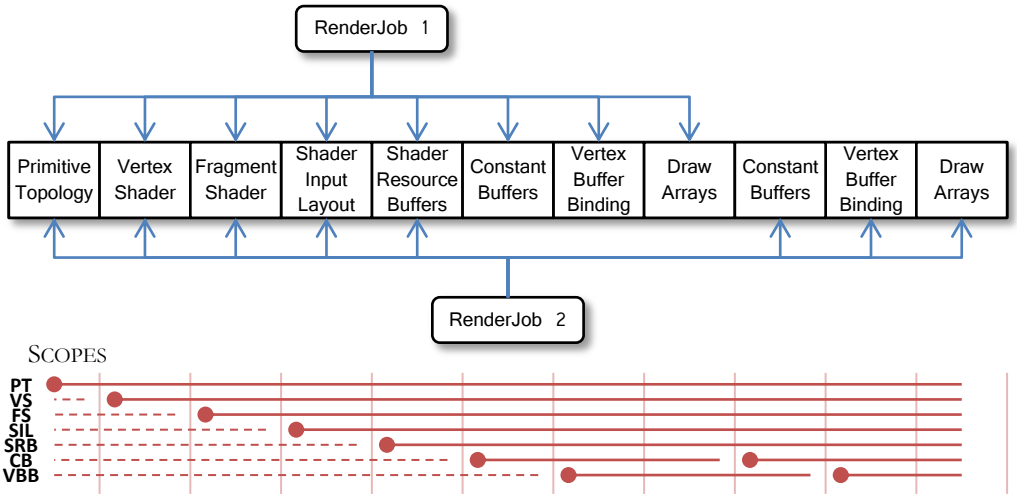


Figure 3.14: Representation of render jobs in an *instruction stream* with instruction scopes

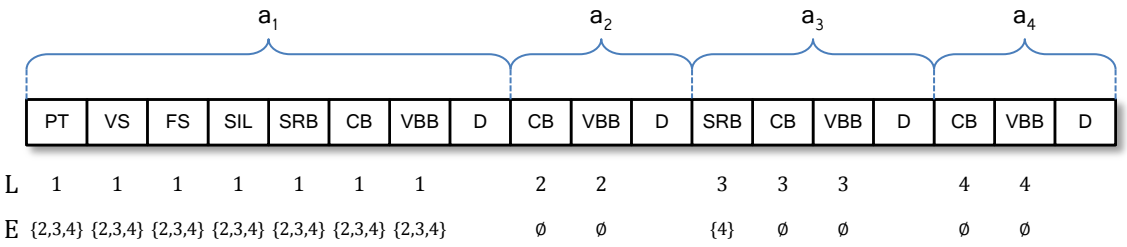


Figure 3.15: *Instruction stream* with *atomic sections*. The prefixed with L shows the index of the *atomic section* the local scope of which the instruction is part of. The line prefixed with E shows the set of *atomic sections* the external scope of which the instruction is part of.

this *local state* will not define the whole render state active at the *draw call* of the *atomic section*. Mind, however, that *local state* does not mean the actual values applied by the instructions, just which kind of state is set. For example, rather than $\{vertex\ shader = x, buffer\ binding = y\}$ the *local state* would just be $\{vertex\ shader, buffer\ binding\}$, meaning that these state-fields are defined, never mind the actual value. Typically *local state* comprises «per-geometry attributes» such as *index* and *vertex buffer bindings*. The rest of the render state (e.g. GPU programs) is set *externally*, i.e. before the *atomic section*. State external to one *atomic section* is always the *local state* of a previous *atomic section*. An *atomic section* can thus be seen as a partially defined *render job*, like a logical expression containing some bound variables ($\simeq$ *local state*) and some free variables ($\simeq$ *external state*). The free variables are later bound depending on where the *atomic section* is placed in the stream. Figure 3.15 shows an example of *atomic sections* and their scopes within an *instruction stream*.

Initially, the *instruction stream* can be segmented into its *atomic sections* with a simple $\mathcal{O}(n)$ algorithm:

input : InstructionStream s, CentroidMapping cm
output: List of AtomicSections as

```
1  as ← ∅
2  localState ← ∅
3  sectionBegin ← 0
4  for i ← 0 to s.Length do
5      if s[i] is state-setting instruction then
6          localState ← localState ∪ {state set by s[i]}
7      else                                  /* s[i] is a draw call ending the atomic section */
8          sectionEnd ← i
9          centroid ← cm[s[i]]
10         a ← new AtomicSection(sectionBegin, sectionEnd, localState, centroid)
11         Append a to as
12         localState ← ∅
13         sectionBegin ← i
14     end
15 end
```

Every *draw call* signifies the end of an *atomic section* at which point the algorithm will create an *atomic section* object storing the *local state* which has been accumulated since the last draw call. Every sections also get assigned a reference to the *dependent centroid* approximating the position of the geometry the section draws. The distance from centroid to camera is later used as sorting criterion as described above.

The next task is to find ranges within this list of partial *render jobs* which can be sorted without changing the stream's semantic. The conditions for two consecutive *atomic sections* a_1 and a_2 to be allowed to exchange positions are:

1. They must have the same *local state* which also implies that they have the same *external state*. Because of this they are independent of each other. Every state-field set by a_1 will be overwritten in time by a_2 and vice versa if they exchanged position because they set exactly the same state-fields ($\simeq$ *local state*) and leave exactly the same state-fields unchanged ($\simeq$ *external state*).

   And because they are consecutive, their «free variables» will be bound to the same values. a_1 will not set a state-field which is external to a_2 as their *external state* is equal. Of course this also applies to the ordering a_2, a_1. So the same *external state* values that were active when the first section is executed are still active for the second one. Figure 3.16 shows a visualization of three re-sortable ranges within a list of *atomic sections*. A re-sortable range appears as an uninterrupted sequence of equal columns.

2. The first criterion asserts that the *draw call* of the *atomic section* being moved keeps its valid state. However, as said before, the scope of state-setting instructions can span across multiple *draw calls* and consequently across multiple *atomic sections*. This introduces

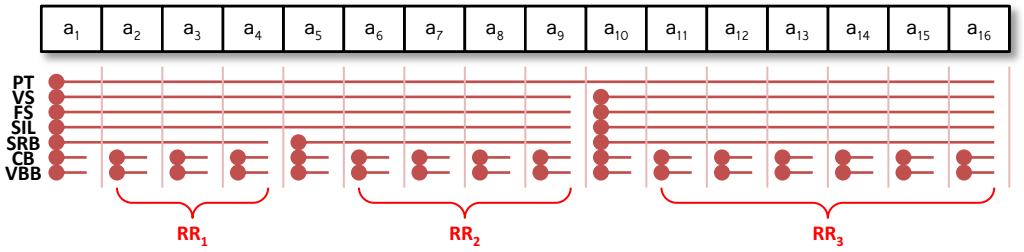


Figure 3.16: Re-sortable ranges of *atomic sections* with instruction scopes

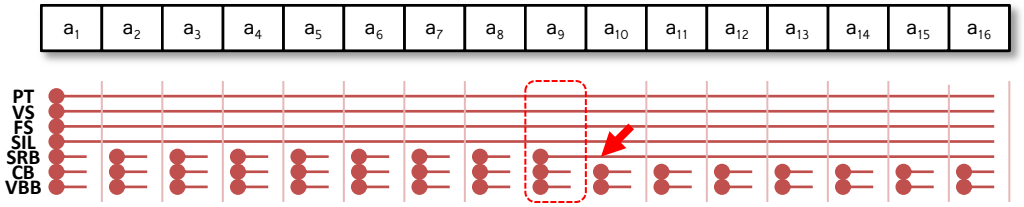


Figure 3.17: *Atomic section* a_9 must not be moved because a_{10} to a_{16} depend on it

dependencies among *atomic sections*, in particular, an *atomic section* a depends on all *atomic sections* which contain a free variable for which a provides a binding. Thus, when moving an *atomic section*, the validity of dependent sections has to be guaranteed: No free variable of any sections dependent on a moved section must be bound to a different (wrong) value. This is the second criterion, and the algorithm adheres to it by regarding all *atomic sections* which bind free variables in other sections as affixed—sections with dependencies will not be moved.

Fortunately it is rather easy to detect whether an *atomic section* has dependencies. As the scope of an instruction ends when another instruction of the same type sets a new value, the scope of an *atomic section* in a certain state-field does never «jump» over another section, it always extends to the right consecutively. Thus, for an *atomic section* a_i to have dependent sections, already the first following section a_{i+1} must be dependent. If the first following section is not dependent, no other section can be dependent—they could only be dependent on a_{i+1} which must have overwritten any *local state* of a_i to be independent of it. a_{i+1} can act as a kind of «scope barrier». As a result determining whether an *atomic section* must stay fixed is the same as determining whether the *local state* of its successor is a real subset of its own *local state*. Figure 3.17 shows an example configuration containing a fixed *atomic section*.

Having these two criteria an algorithm finding re-sortable ranges within a list of *atomic section* can be defined as follows:

input : List of AtomicSections as
output: Set of re-sortable ranges rs

```
1  rs ← ∅
2  rangeBegin ← 0
3  for i ← 1 to as.Length do
4  |  if (s[i] is affixed) ∨ (localstate(s[i]) ≠ localstate(s[i − 1])) then
5  |  |  newRange ← [rangeBegin, i[
6  |  |  if newRange.Length ≥ MINSIZE then
7  |  |  |  rs ← rs ∪ {newRange}
8  |  |  end
9  |  |  rangeBegin ← i
10 |  end
11 end
12 lastRange ← [rangeBegin, as.Length[
13 if lastRange.Length ≥ MINSIZE then
14 |  rs ← rs ∪ {lastRange}
15 end
```

All in all a *ReSortableInstructionStream* stores the list of *atomic sections*, the list of re-sortable ranges, and two copies of the *instruction stream*, one for sorting, one for reference. This data is all that is needed to dynamically keep the stream roughly in front-to-back drawing order.

3.6.5 Interleaved Resource Upload

Normally, the *DependencyIndex* will update the values of *dependent resources* which are out-of-date and then immediately trigger the value to be uploaded to graphics memory. As a consequence all resource values are uploaded before the first rendering command is issued from the instruction stream (Figure 3.18).

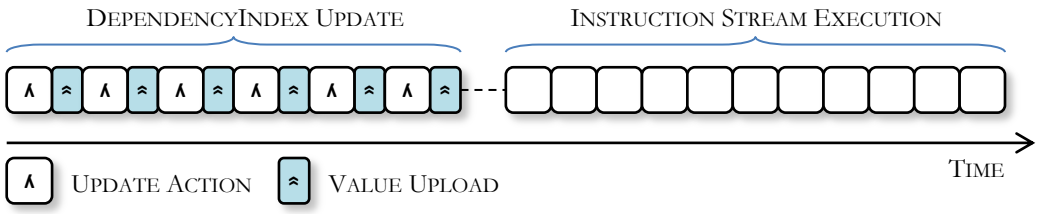


Figure 3.18: Resource update protocol where values are uploaded to immediately

For some reason, this protocol seems to be detrimental to rendering performance. In order to alleviate this problem, *render caches* support an alternative resource update protocol. As before, the *DependencyIndex* will update *dependent resources* values where necessary; but it will not automatically trigger the upload of the new values. Instead uploading gets interleaved into the execution of the *instruction stream*. This is easily facilitated by introducing *upload instructions* holding a reference to the resource to be uploaded (Figure 3.19).

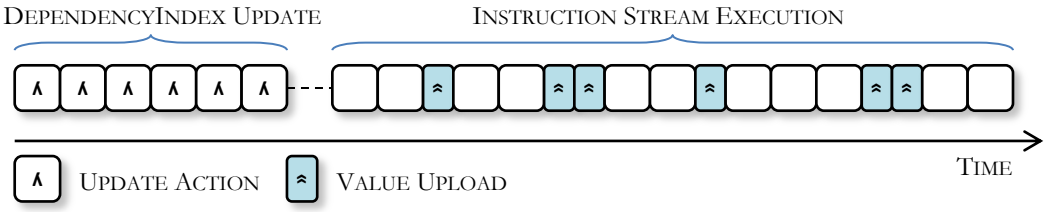


Figure 3.19: Resource update protocol with interleaved values uploads

As all resources maintain a «dirty flag» which indicates whether its value has changed since the last upload, the *upload instructions* can be inserted into the stream at build-time without impairing the streams semantic. Inserting the upload instructions is implemented by a simple augmentation of the process translating *semantic instructions* to *native instructions*. Each time before a *semantic instruction* is translated it is queried for all *dependent resources* it uses.[19]Then, if the resource has not already been updated for a previous instruction in the stream, an *upload instruction* is inserted in front of the instruction using the resource. This way every resource gets uploaded once right before its first usage. See Algorithm 3.6 for a full listing of the process.

[19]Mind that *dependent resources* with zero *dependencies* can safely be ignored because their value will never change and thus never needs to be uploaded during execution.

input : List of semanticinstructions *semantic*
output: List of nativeinstructions *native*

```
1  native ← ∅
2  uploaded ← ∅
3  for i ← 0 to semantic.Length do
4  |  foreach Resource r used by semantic[i] do
5  |  |  if (r has dependencies) ∧ (r ∉ uploaded) then
6  |  |  |  Append new UploadInstruction(r) to native
7  |  |  |  uploaded ← uploaded ∪ {r}
8  |  |  end
9  |  |  Append (compile (semantic[i])) to native
10 |  end
11 end
```

Algorithm 3.6: Injecting *resource upload instructions* into an *instruction stream*

3.6.6 Concurrent Resource Value Computation

This optimization only works in conjunction with the previous optimization *Interleaved Resource Upload*. The following principle is used to leverage multiple processor cores concurrently: The *DependencyIndex*, after determining which *dependent resources* need to be updated, delegates the task of updating to a number of worker threads and gives back control to its owning *render cache*. The *render cache* then immediately starts executing the *instruction stream*. Since computing resource values often takes a large percentage of the time needed to render a frame, this can result in major performance gains.

However, there is a race condition between the resource value computations and the interleaved resource uploads which needs to be handled by proper synchronization. In the prototype this is implemented by allocating a *ManualResetEventSlim* (referred to as *events* from here on) for each *dependent resource* that needs synchronization. After the *DependencyIndex* has determined the set of resources to update, it locks these resources by *resetting* their corresponding *events*. Then it starts the worker threads which pick the resources one by one from a queue, update them and then, after each individual update, unlock the resource by *signaling* its *event*. At the same time the *instruction stream* is executed by the main thread. To facilitate synchronization here, *upload instructions* are replaced by *synchronized upload instructions*. This instructions will always *wait* on the *event* associated with the resource to be uploaded. This way no resource upload is triggered before the value of the resource is update-to-date. Figure 3.20 shows an exemplary *instruction stream* execution with concurrent value computations.

The optimization can be further aided by partitioning the stream into two section when it is created. The first section only contains instructions using *dependent resources* with zero *dependencies*. As their values are constant they never have to be synchronized. The second partition contains resources which will probably need updates. In the end there is still just one *instruction stream* but the worker threads will have a little more time before stream execution blocks for the first time.

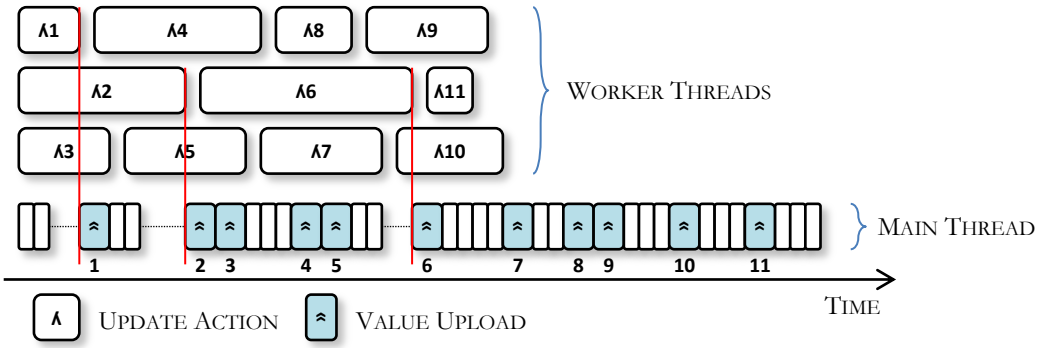


Figure 3.20: Resource update protocol with interleaved values uploads and concurrent value computations

Another improvement is to put the resources into the update queue in the same order they occur in the stream. This avoids the situation that stream execution is blocked early by a resource which is updated late, leading effectively to the case that most or all resource values are computed before the stream is executed.

3.6.7 Memoization

One of the most expensive tasks to perform for rendering a frame is the computation of transformation matrices during traversal. Although the *dependency system* can optimize recomputations of constant transformations away, for frequently changing transformations the matrices of all *transformation nodes* on the path to the resource have to be accumulated anew in the *update action*. There are two sources of inefficiencies in this setup:

(1) Because *update actions* do not interact with each other by default they cannot share common intermediate results. Yet, these occur quite often due to the tree structure of most scene graphs where many *transformation nodes* are shared by a multitude of paths. In contrast, regular scene graph traversals do re-use intermediate results by using a stack for computing transformations.

(2) The *update action* must multiply all transformation nodes along the path; even if only one matrix actually has a different value. Yet, if intermediate multiplication results along the path where available it would only be necessary to re-multiply matrices starting at the changed one while re-using the intermediate result from before the change.

For these reasons it becomes obvious that there is some optimization potential. At the very least point (1) should be solved as it makes transformation computations in very dynamic settings more expensive when using the caching system than when doing regular traversals. The following paragraphs will show how a variant of memoization can—at the expense of some storage space—solve both problems.

Normally, memoization designates caching the results of function calls for different sets of parameters. This is easily implemented when function are assumed to be free of side effects. In the context of transformations within a scene graph this would mean that a function $\mathrm{transformation}$ from *traversal path* to transformation matrix would always yield the same matrix for the same path. Unfortunately this is not the case. Matrices in a *transformation node* can change any time, which also changes the accumulated transformation of all paths containing the node. However, the *dependency system* allows to still implement a form memoization that exploits the information about when the value of a *transformation node* has changed. Whenever a *transformation node* has changed, all memoized transformation values depending on it can correctly be updated.

The $\mathrm{transformation}$ function of a path p can be (recursively) defined as

$$\mathrm{transformation}(p) = \begin{cases} \mathrm{value}(p) & \text{if } p \text{ is a single node} \\ \mathrm{transformation}(\mathrm{prefix}(p)) * \mathrm{value}(last(p)) & \text{else} \end{cases} \tag{3.4}$$

where $\mathrm{prefix}(p)$ is p without its last node. Executions of this function for a number of paths can be memoized in a *forest* structure where each node holds the function result for a complete path p and the parent of the node holds the result of the recursive function call for $\mathrm{prefix}(p)$ (see Figure 3.21 for an example).

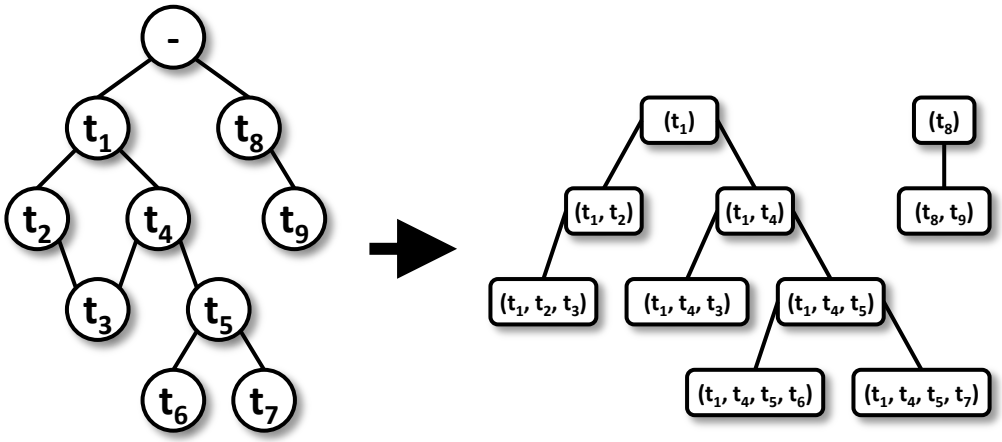


Figure 3.21: A scene graph with transformations and its memoization forest

The nodes are called *memo nodes* and for every path in the scene graph which starts at the root the forest contains one such *memo node* storing the path's transformation matrix. Another way of looking at it is that there is exactly one *memo node* for every state the *transformation stack* takes during a render traversal. *Dependent resources* using the transformation (e.g. *ModelViewProjection*) can store a reference to the *memo node* for the needed path (i.e. the path to the node the *dependent resource* is created at) and query it for its value. In the prototype, the whole forest is wrapped in a data structure that indexes every *memo node* by a hash value of its path, which allows to efficiently check whether there already is an entry for a given path/prefix.

The major task in the setup—and the reason why *memo nodes* keep a reference to the *memo node* representing their prefix—is to keep the memoized matrices up-to-date. As long as there are no changes, the memoized values stay correct. However, if a *transformation node* does change, all its corresponding *memo nodes*[20] and their descendants need to be updated. That is, changes always need to be propagated downward in the tree.

How can the system know about changes that happened? The answer is: Through the *dependency system* which is already in place and already tracks changes of *transformation nodes*. On closer examination, a *memo node* can be considered a *dependent resource* and a *memo node* with ancestors is a *composite dependent resource*. However, in this case a parent node is at the same time a *value source* and a *dependent resource*. This poses no problem but has some implications:

- Because the *value source* is also a *dependent resource*, it must always be up-to-date before its value may be used.
- Same as with regular *composite dependent resources*, the *dependency set* of a *memo node* is the union of the *dependency sets* of its *value sources*. In this case the union of the *dependency set* of its corresponding *transformation node* (=*value source* 1) and the *dependency set* of the parent node (=*value source* 2).
- The *update action* of the *memo node* has to bring the transformation matrix for its path up-to-date. As shown in (3.4), this computation is recursive and uses the accumulated value the prefix of the path in question. This prefix value can simply be read from the parent node, as long as the parent node is guaranteed to be up-to-date.
- If the value of the parent node is indeed up-to-date no further recursion is required.
- Similar to *dependent constant buffers*, *dependencies* in a *memo node* can be encoded in a bit mask. And since the *dependency set* of a parent node is always a subset of the child node's *dependency set*, a bit mask generated for a child node is also compatible with the parent node's bit mask.
- Also the *dependency set* of the *transformation node* can be encoded in a bit mask of the same format. This allows to use the same bit mask for both *value sources* of a node.
- As *memo nodes* might be accessed simultaneously by several threads, they must be properly synchronized. The prototype uses light-weight per-node *spin locks* to this end. As nodes are always locked in the same order (from child to parent) there cannot be any deadlocks.

The final algorithm for safely reading the memoized value from a *memo node* then looks as shown in Algorithm 3.7.

Creating a *memo node* node is a simple matter of creating the *dependency set* bit mask, storing a reference to its *transformation node* and linking to the right parent node (identified by the prefix of the node's full path). To utilize the memoization system, the *update action* of resources using the transformation just needs to use the right *memo node* as their *value source*.

[20] A *transformation node* can have more than one corresponding *memo node*. If a *transformation node* is reachable via n paths it will have n corresponding *memo nodes*.

```
input: MemoNode memoNode
1  acquire memoNode.Lock
   // build dependency change mask
2  changeMask ← 0
3  for i ← 0 to memoNode.DependencyCount do
4  |  if memoNode.Dependency[i].Version ≠ memoNode.RefVersion[i] then
5  |  |  memoNode.RefVersion[i] ← memoNode.Dependency[i].Version
6  |  |  changeMask ← changeMask | (1«i)
7  |  end
8  end
   // check own value
9  if (((changeMask & memoNode.ValueSourceMask) ≠ 0)) then
10 |  thisValue ← memoNode.ValueSource.Value
11 end
   // build aggregate value for whole path
12 if memoNode has parent node then
   |  // check parent value
13 |  if ((changeMask & memoNode.ParentMask) ≠ 0) then
14 |  |  parentValue ← GetMemoValue(memoNode.Parent) ;      /* recursive call */
15 |  else
16 |  |  parentValue ← memoNode.Parent.AggregateValue ;     /* no recursion */
17 |  end
18 |  memoNode.AggregateValue ← thisValue * parentValue
19 else
20 |  memoNode.AggregateValue ← thisValue
21 end
22 release memoNode.Lock
23 return memoNode.AggregateValue
```

Algorithm 3.7: The *GetMemoValue()* algorithm

3.6.8 Disposing Cached Subgraphs

Once a *render cache* is built it can take over any rendering responsibilities of the subgraph it is an representation of. Hence, assuming the cached subgraph will not undergo any changes in the future, it can be discarded just below the *cache node*. This will free up main memory resources, especially the main memory representation of *vertex buffers*, which often constitutes a substantial part of the scene graph's memory usage. Only object's which are referenced by the *render cache* through *update actions* will be retained. The reference counting scheme described above will take care of keeping *GPU* resources such as *vertex buffers* and *GPU programs* alive.

3.6.9 Cached Culling Hierarchy

View frustrum culling is one of the most important optimizations in graphical applications and it is often augmented by using a *bounding volume hierarchy* (*BVH*). It would be counter productive to the purpose of the *scene graph caching system* if it precluded view frustrum culling. This section will show how the *dependency system* can even aid at keeping a *BVH* up-to-date with information already available.

We set the following goals for the integration of a *BVH*-based culling solution with the caching system:

(1) Obviously, parts of the scene graph not intersecting the view frustrum should not be rendered.

(2) Not quite as straight forward, *dependent resources* within culled parts of the scene graph should not be updated.

(3) Bounding boxes within the *BVH* should be kept up-to-date through changes with minimal effort.[21]

These goals are achieved by the following algorithm: A special node type, the *CachedCullingRoot* is placed at the root of the *BVH*. The leaves of the *BVH* are constituted by *cache node*—consequently a *cache node* is the smalled unit of culling in this setup.[22] The internal nodes of the *BVH* are *BoundingBoxAttribute* nodes which the scene graph environment already uses for hierarchical culling in non-cached scenarios.

Using a *BuildCachedCullingHierarchyTraversal*, the *CachedCullingRoot* creates an internal, light-weight representation of the *BVH* (Figure 3.22). As expected, this representation has a tree structure and every node stores the bounding volume of its subtree. Additionally, every node also stores a flat list of all *render caches* in subtree. When *render traversal* reaches the *CachedCullingRoot*, this internal tree is used for rendering, the actual subgraph is not touched.

Processing the *BVH* for rendering follows a straightforward algorithm:

[21] This does not encompass changing the structure of the hierarchy by moving nodes to different subtrees. Hence, the algorithm described later on cannot keep a hierarchy from degenerating when objects within the same node drift to far apart.

[22] Although it would be possible to implement some kind of intra-*render cache* frustrum culling support if the need arose.

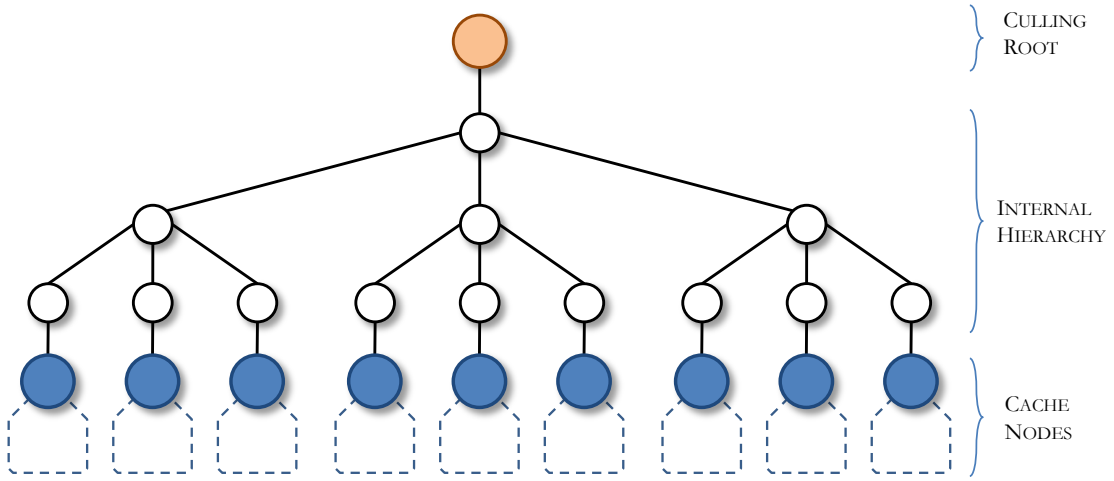


Figure 3.22: A *bounding volume hierarchy* with a *CachedCullingRoot* at the top and *cache nodes* as leaves.

```
1  procedure RenderNodeBVH(node, viewFrustrum)
2  begin
3      if node is a leaf then
4          node.RenderCache.Render()
5      else
6          visibleChildren ← {c | c ∈ node.Children, c intersects viewFrustrum}
7          if visibleChildren.Count = node.Children.Count then
               // Render complete subtree, no further recursion
8              foreach cache ∈ node.RenderCaches do
9                  cache.Render()
10             end
11         else
               // Recursive decent into child nodes
12             foreach child ∈ visibeChildren do
13                 RenderNodeBVH(child, viewFrustrum)
14             end
15         end
16     end
17 end
```

Algorithm 3.8: Algorithm for rendering a *bounding volume hierarchy* (*BVH*)

This algorithm satisfies goals (1) and (2). *Render caches* containing no geometry intersecting the view frustrum will not receive a call to their *Render()* method and consequently their *dependent resources* will not be touched.

However, there is still the question of keeping bounding volumes up-to-date. As has already

been the case a few times when dealing with change, the *dependency system* provides the facilities to implement this efficiently. *Render caches* can be configured to keep their bounding volume as a *dependent resource*. The bounding volume of a *render cache* is the union of the bounding volumes of all contained geometries which are also modeled as *dependent resources*. As transformations and morphing of geometries are already tracked by the *dependency system*, all the information needed for keeping geometry and *render cache* bounding volumes up-to-date is available. Similar to *memo nodes* (p. 75), bounding volumes are composite *dependent resources* having other *dependent resources* as *value sources*: Vertex data and transformations comprise the *value sources* of geometry bounding volumes, which in turn are the *value sources* of *render cache* bounding volumes. Going further up the culling hierarchy, the bounding volume of an internal node is a *dependent resource* with the bounding volume of its children as *value sources*. Following this pattern, *dependent bounding volumes* can be constructed for each node in the *BVH* up until the root.

Modeling bounding volumes as *dependent resources* has a couple of beneficial side effects:

- A *dependent bounding volume* having an empty *dependency set* is guaranteed to never change and hence never has to be updated.
- Some of the child volumes of a *dependent bounding volume* might have *dependencies* while others do not. Those child volumes without *dependencies* are also guaranteed to never change and hence it suffices to created their union once at the beginning. This union can then be re-used for updating without having to recurse into the static child volumes.
- Similarly, *dependency*-less bounding volume will always act as a recursion bound, even if it is high up in the hierarchy.

As a result of these properties very little work is needed to keep the *BVH* up-to-date.

3.6.10 Optimization Composability

This section will investigate whether the different proposed optimizations can be enabled at the same time. Table 3.3 shows an overview of optimization composability with cases that need special attention explained in more detail. As can be seen in the table all optimizations can be enabled at once.

	SI	SS	OS	IRU	CVC	MEM	SD	CULL
RR	✓	✓	✓	✓	✓	✓	✓	✓
SI		✓	(1)	(2)	✓	✓	✓	✓
SS			✓	✓	✓	✓	✓	✓
OS				(3)	✓	✓	✓	✓
IRU					✓	✓	✓	✓
CVC						✓	✓	✓
MEM							✓	✓
SD								✓

RR	...	Redundant Instruction Removal	CVC	...	Concurrent Value Computation
SI	...	Super Instructions	MEM	...	Memoization
SS	...	State Sorting	SD	...	Subgraph Disposal
OS	...	Overdraw Sorting	CULL	...	Culling
IRU	...	Interleaved Resource Uploads			

(1) As beginning and end of *atomic sections* are defined by the position of instructions in the *instruction stream* they may need to be adapted when the introduction of a *super instruction* shifts instruction indices. Also, a *super instruction* cannot be inserted if it crossed the bounds of an *atomic section*.

(2) If a *super instruction* replaces a range of instructions which need resource uploads then any *upload instructions* necessitated by a replaced instruction must be moved in front of the *super instruction*.

(3) *Upload instructions* must always be in front of their resource's first usage in the stream. To prevent stream re-sorting from breaking this condition, any *upload instructions* contained within a *re-sortable range* must be moved in front of the range. This way, they are not moved by sorting.

Table 3.3: Optimization composability

3.7 Default Cache Types

The prototypical implementation of the *scene graph caching system* supports two types of *render caches* for the most common use cases. This section will describe them in more detail, especially how they relate to the concepts and optimizations presented above.

3.7.1 SolidPassCache

The *SolidPassCache* is used to store non-transparent geometry and it is where all of the above comes together. It uses a single *instruction stream* to render all of its geometry. It uses the *DependencyIndex* class to keep the *dependent resources* it contains up-to-date. Finally, it can use all of the optimizations described in the previous section. The complete process for building a solid pass cache is as follows:

(1) The *cache node* decides that it needs to build a *SolidPassCache*.

(2) The *cache node* calls *RenderCacheFactory.Create(pass, traversal, cache node)*.

(3) The *RenderCacheFactory* selects the factory function registered for the render pass (the constructor of *SolidPassCache* in this case).

(4) The *RenderCacheFactory* starts an *ExtractCachingDataTraversal* below the *cache node*:

- The *ExtractCachingDataTraversal* builds the *SlimSG* for the subgraph using the *ICacheable* interface implementations in the scene graph.
- This will create all *(dependent) resources* needed.
- At this point the *dependent resources* may have already been set up to use *Memoization* (p. 74) during their creation process.
- OUTPUT: *SlimSG*

(5) The *RenderCacheFactory* packs the *SlimSG*, a *cache node* reference, and a *ResourceDisposer*[23] into a *CachingInfo* object. OUTPUT: *CachingInfo* object

(6) The *RenderCacheFactory* calls the previously selected factory function ($\simeq$*SolidPassCache* constructor) with *CachingInfo* and traversal as arguments.

(1) The *SolidPassCache* constructor reads the cache configuration from the *cache node*

(2) IF configured for interleaved resource uploads THEN create a *DependencyMapping*[24]

[23] The *ResourceDisposer* object allows to dispose all resources created when the *SlimSg* was built. This disposer is used later by the *render cache* to cleanly dispose all resources it owns.

[24] The *DependencyMapping* object is a means of communication between *instruction stream* factories and the *DependencyIndex* constructor. Both need to use the same *ManualResetEventSlim* instances for synchronization. The *DependencyMapping* provides a mapping of resources to their *ManualResetEventSlim* instances. The *instruction stream* factory can also provide a hint on the position of a resource's first usage in the *instruction stream*. This allows the *DependencyIndex* to order resource updates roughly the same as they occur in the *instruction stream*.

(3) The *SolidPassCache* constructor builds the *semantic instruction stream*, which consists of the following steps:

   (1) A *RenderJobBuilder* is created

   (2) The *RenderJobBuilder* is sent through the *SlimSG*. OUTPUT: List of *RenderJobs*

   (3) At this point RenderJobs can optionally be sorted to minimize state changes (*State Sorting*, p. 65).

   (4) The code generator creates a list of *SemanticInstructions*, not generating redundant instructions. (*Removal of Redundant Instructions*, p. 64)

   (5) OUTPUT: List of *SemanticInstructions*

(4) The *SolidPassCache* constructor selects the appropriate of *IInstructionStreamFactory* based on configuration. This can be one of:

   - SimpleInstructionStreamFactory
   - InterleavedInstructionStreamFactory
   - ResortableInstructionStream.Factory

   Factories are supplied with the *DependencyMapping* created earlier.

(5) The *SolidPassCache* constructor calls *IInstructionStreamFactory.Create()*

   SimpleInstructionStreamFactory Simply translates each *semantic instruction* to its *native instruction* counterpart.

   InterleavedInstructionStreamFactory Same as the above but will also inject (possibly synchronized) *resource upload instructions* for *dependent resources*. (*Interleaved Resource Upload*, p. 72 and *Concurrent Resource Value Computation*, p. 73)

   ResortableInstructionStream.Factory Analyzes the *semantic instruction stream* and creates metadata needed for re-sorting the stream as described in *Overdraw Sorting* (p. 65). May also inject *resource upload instructions* if configured to do so.

   All of the above may also apply the *Super Instructions* (p. 64) optimization on the translated stream.
   OUTPUT: a native *instruction stream* (implementing *IInstructionStream*)

(6) The *instruction stream* is stored in an instance field of the *SolidPassCache*.

(7) The *SolidPassCache* constructor collects all *dependencies* from the *SlimSG* and stores them in an instance field.

(8) The *SolidPassCache* constructor collects all *dependent resources* from the *SlimSG* and builds two *DependencyIndices* from them, a «pre cull» and a «post cull» index, using *dependent resources* and the *DependencyMapping* as arguments:

   - The *DependencyIndex* constructor builds an *inverted index* structure for fast queries.

- The *DependencyIndex* constructor uses the *DependencyMapping* to build synchronization bookkeeping data for the *dependent resources*.

(9) The *SolidPassCache* constructor stores *pre* and *post cull DependencyIndices* in instance fields.

(10) The *SolidPassCache* constructor optionally creates a dependent bounding box used for culling and stores it.

(11) Now the *SolidPassCache* is completely built. OUTPUT: the *SolidPassCache* instance.

(7) The *RenderCacheFactory* returns the *SolidPassCache* to the *cache node*.

(8) The *cache node* stores the *SolidPassCache* instance in its dictionary of caches with *Pass.Solid* as key

The process for rendering a *SolidPassCache* is shown in Algorithm 3.9.

input: SolidPassCache c, Boolean $doPreCull$

```
1  if doPreCull then
       // If the pre-cull tasks have not already been executed by the culling hierarchy
2      foreach d ∈ c.DependenciesToEvaluate do
3          d.Evaluate()
4      end
       // Update pre-cull dependent resources containing the dependent bounding box
5      c.PreCullIndex.Update()
6      if ¬(c.BoundingBox intersects view frustrum) then
7          return
8
9  end
10 c.PostCullIndex.Update()
11 c.InstructionStream.Execute()
```

Algorithm 3.9: Algorithm for rendering a *SolidPassCache*

3.7.2 TransparencyPassCache

The *TransparencyPassCache* is different from the *SolidPassCache* because transparent geometry always has to be rendered back-to-front in order to get correct visual results. This has a number of consequences:

- The sorting algorithm must be able to freely apply a render order to all transparent geometry. No preexisting grouping (such as assignment to a *render cache*) must impede this. A simple, general solution to this problem is to place all transparent geometry in a single *render cache*.
- There cannot be a static *instruction stream* because the render order has to be adapted dynamically depending on the camera position.
- Some optimizations are not possible anymore such as *State Sorting* or *Overdraw Sorting*. Others, such as *Removal of Redundant Instructions* have to be applied after every change of the *instruction stream*, which makes them a lot less attractive.

Sorting

Geometry sorting is provided by a two-level BSP-Tree algorithm (kindly provided by Robert F. Tobler). All transparent geometry is subdivided and regrouped into non-intersecting *geometry nodes* of a single material. For each of these nodes, a BSP-Tree is created which allows to quickly sort the individual primitives back-to-front. This way the *index buffer* of each *geometry node* can be updated before rendering. On top of these nodes sits another BSP-Tree which is used to sort the *geometry nodes* back-to-front too. This way all transparent primitives in the scene can be sorted. In the scene graph this algorithm is represented by a *TransparentSceneGraph* node, below which all transparent *geometry nodes* must be contained. This node is used by the *immediate mode* renderer as well as by the caching system. The *TransparentSceneGraph* node type implements the *ICacheable* interface and the caching system can use the *ExtractCachingDataTraversal* to retrieve all necessary data from it.

Cache Implementation

The *TransparencyPassCache* in the prototype is implemented as follows:

1. Instead of creating a single, big *instruction stream* as in the *SolidPassCache* a small *instruction stream* segment is created for each *render job* that is identified when building the cache. This segment contains all instructions needed to perform the rendering of one transparent *geometry node*.
2. The render order of the *instruction stream* segments is extracted from the *TransparentSceneGraph* node as a *dependent resource* internally referencing the top-level BSP-Tree mentioned above. Executing the *update action* of this *dependent resource* will bring the *instruction stream* segments in the desired order.

3. The *index buffers* of the single geometries are modeled as special *TransparencyIndexBuffer dependent resources*, each referencing the BSP-Tree used for sorting the primitives of the geometry. Executing the *update action* of such a *TransparencyIndexBuffer* will sort it in a way that primitives are back-to-front with respect to the current camera position.

As a result, updating all *dependent resources* of the cache—including *TransparencyIndexBuffers* and the top-level render order resource—will bring it in a state that is ready for rendering. After that the individual *instruction stream* segments just have to be executed according to the render order resource.

Figure 3.23 shows a usage example of the *TransparencyPassCache* within a larger scene graph.

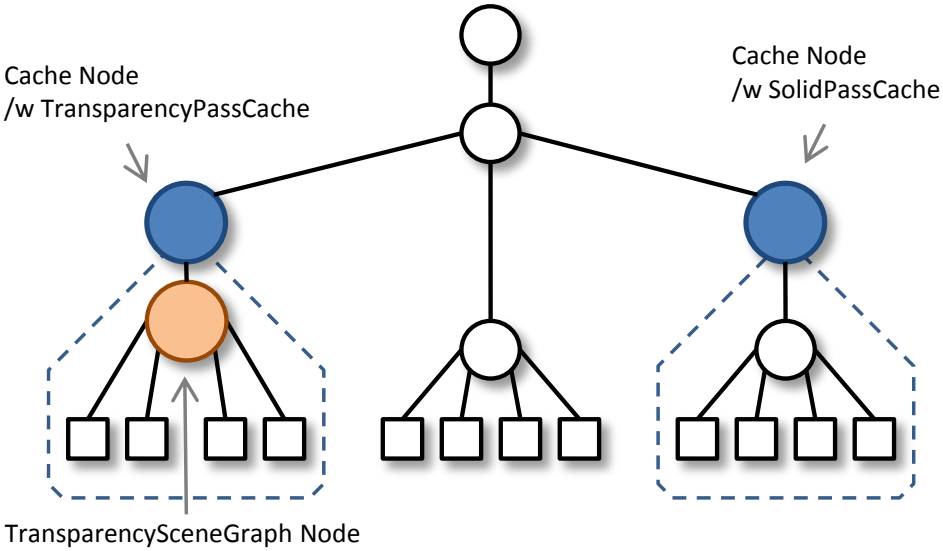


Figure 3.23: A *cache node* with *TransparencyPassCache* within a larger scene graph.

3.8 Evaluation

The last section of this chapter will investigate how the prototypical implementation of the caching system fares in relation to the goals set at the beginning of the chapter (p. 21).

Reduce Traversal Cost. As described in *3.3.2 Cache Nodes & Render Caches* the render traversal does not need to descend into cached subgraphs. Instead the *instruction stream* located at the *cache node* at the root of the subgraph is executed. The performance analysis in the next chapter will show that this can result in significant performance improvements.

Allow Partial Caching. Section *3.3.2 Cache Nodes & Render Caches* shows how *cache nodes* can be placed very flexibly at selected positions within the scene graph, allowing for fine-grained control over which scene graph parts will be cached and which will not be.

Allow Modification. Section *3.3.4 Cache Invalidation* describes various classes of scene graph modifications. Section *3.5 Dependency-Aware Scene Graph Caching* shows how dependency-aware *render caches* are able to incrementally adapt to the most common modification class encompassing non-structural changes like positional updates of objects or cameras.

Easy Configuration. Marking a certain subgraph for caching can simply be achieved by placing a *cache node* at its root. The *cache node* can be configured via a set of boolean flags enabling or disabling any of the optimizations described above.

Extensibility. The caching system can be extended at various points. It is possible to implement new types of *dependencies* (*IDependency*), *dependent resources* (*IDependent<T>*), scene graph nodes (*ICacheable*), and *render caches* (*IRenderCache*) by implementing the corresponding interfaces. This allows to exploit application specific optimization opportunities.

Optionally Replace Cached Subgraph. Section *3.6.8 Disposing Cached Subgraphs* shows how a cached subgraph can be discarded with the *render cache* taking over all of its rendering responsibilities.

Use the Graphics Hardware Efficiently. The optimizations *State Sorting*, *Overdraw Sorting*, and *Interleaved Resource Upload* strive to use the graphics hardware in a way that allows it to process commands and data in an efficient way. The performance analysis section investigates how much of a speedup this yields (if any).

In conclusion, the caching system achieves all the goals set out at the beginning. How well this translates into actual performance gains will be investigated in the next chapter *Results*.

CHAPTER 4

Results

This chapter will investigate how the *scene graph caching system* performs under various test conditions by putting the system through a number of synthetic test scenes with different scene and cache configurations.

4.1 Test Setup

Two synthetic scenes are used which can be parameterized for varying degrees of geometry complexity, dynamicity, and number of used GPU programs.

- The *SPHERES* scene (Figure 4.1) consists of an $n \times n \times n$ grid of spheres where n can be configured, as can be the number of triangles used to model a single sphere. *Vertex* and *index buffers* are not shared between geometries, each sphere has its own set of buffers. By default, the only moving object is the camera circling the grid. However, a percentage can be given to designate the number of rotating spheres, necessitating *constant buffer* updates. Each sphere uses one of up to eight *surfaces* with normal and environment mapping.

- The *MENGER SPONGE* scene (Figure 4.2) uses a Menger sponge structure [45] the recursion level of which can be configured, resulting in geometry counts of 400 (level 2), 8000 (level 3), and 160000 (level 4) cubes. Each recursive sponge structure is nested within a *transformation node* and *group node*, resulting in a comparatively deep transformation hierarchy. All cubes share the same *vertex* and *index buffers*. They also all share the same surface with normal mapped lighting with 3 light sources. Again, by default the only moving object is the camera, but the lowest level of *transformation nodes* can be configured to apply a time-dependent rotation to the cubes.

Each scene is rendered for a number of frames while the average time needed to render a frame is recorded. Also, the time needed to build the scene, the graphics and main memory used,

and the number of *instructions* is measured.[1]

Each test configuration is run for 5 iterations. The final result is calculated as the average of the results of iterations 2 to 5 in order to offset startup costs caused by the Just-In-Time compiler of the *.NET Framework*.

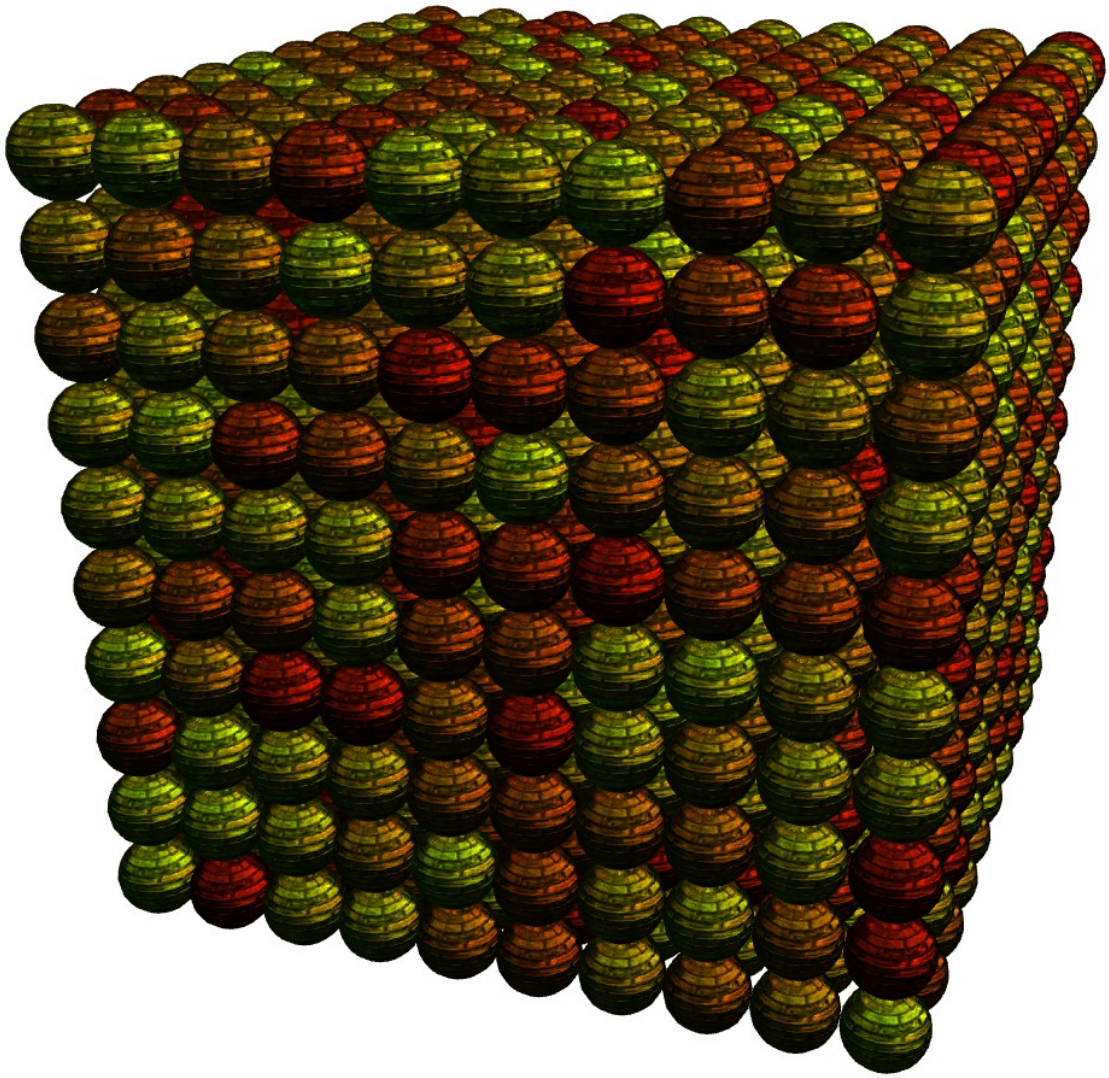


Figure 4.1: A screenshot from the *SPHERES* test scene.

[1]The main memory used by the scene is measured by calling *GC.GetTotalMemory(true)* before and after building the scene. This function provided by the *.NET Framework* will only give an estimate of the number of bytes allocated on the heap. Consequently the heap memory consumption numbers should only be considered as a general hint.

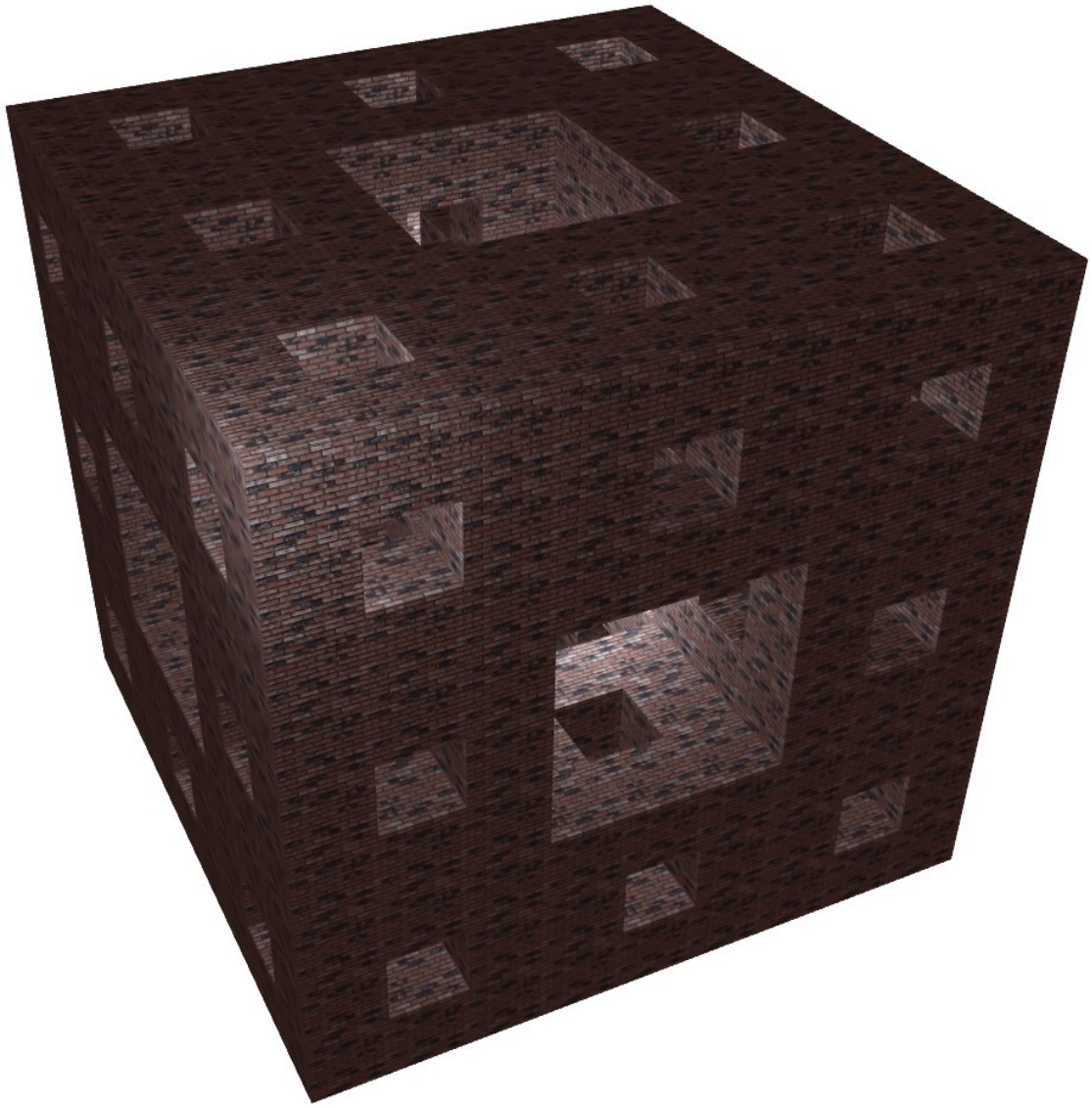


Figure 4.2: A screenshot from the *MENGER SPONGE* test scene.

4.2 Result Analysis

4.2.1 Caching Baseline - Static Scenes

The first set of tests will compare the performance of immediate mode rendering with cached mode with varying numbers of geometries, and varying numbers of vertices per geometry to find out for which types of scenes the caching system can bring improvements. The only moving object in the scene is the camera. These tests are run with no optimizations enabled. Any speedups are therefore a result of using *instruction streams* and persistent *constant buffers* modeled as *dependent resources*.

Frame Time

Table 4.1 shows the absolute and relative frame times of the *SPHERES* scene with varying combinations of geometry count and geometry complexity. For both, cached and immediate mode, more complex scenes mean longer frame times. The relative frame times, however, show that cached mode is consistently faster than immediate mode, ranging from 2.17 to 3.86 times as fast. This suggests that scene graph traversal (which is replaced with *instruction stream* execution in cached mode) is indeed the performance bottleneck when rendering these scenes.

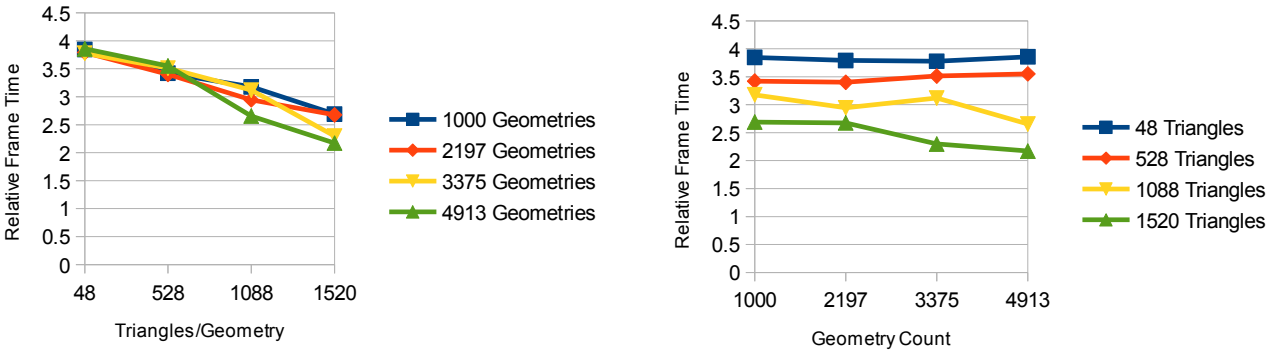
Frame Times (ms) IMMEDIATE	1000	2197	3375	4913	← Geometry Counts
48	16.18	34.97	53.81	79.31	
528	16.42	35.26	54.85	80.32	
1088	16.43	35.45	54.95	80.22	
1520	16.56	35.86	55.69	80.97	
↑ Triangles/Geometry					

Frame Times (ms) CACHED	1000	2197	3375	4913	← Geometry Counts
48	4.21	9.23	14.25	20.57	
528	4.8	10.37	15.62	22.62	
1088	5.17	12.04	17.63	30.22	
1520	6.16	13.4	24.25	37.29	
↑ Triangles/Geometry					

Relative Frame Time	1000	2197	3375	4913	← Geometry Counts
48	3.84	3.79	3.78	3.86	
528	3.42	3.4	3.51	3.55	
1088	3.18	2.94	3.12	2.65	
1520	2.69	2.68	2.3	2.17	
↑ Triangles/Geometry					

Table 4.1: Absolute and relative frame times with cached and immediate mode rendering in various static configurations of the *SPHERES* scene using eight surfaces.

This hypothesis is further backed up when examining the relative frame rates along fixed axis for geometry count (4.3a) and geometry complexity (4.3b): While the relative performance gain stays roughly constant with varying geometry count, it drops with increasing geometry complexity. In other words, the caching system is able to optimize the more complex scene graph traversal caused by a higher geometry count, but it cannot optimize the increasing influence of higher complexity per geometry.



(a) (b)

Figure 4.3: Relative frame times in the *SPHERES* scene with fixed geometry counts and complexity.

The frame times for the MENGER SPONGE scene a given in table 4.2. Again, frame times are consistently smaller with caching enabled. This time even up to 14.24 times faster for the medium sized scene and still 5.56 times faster for the small scene. The higher performance gain can be explained by the deeper transformation hierarchy in the *MENGER SPONGE* scene. The caching system only needs to calculate the accumulated transformation matrices once while the immediate mode renderer has to do this every frame.

Frame Times (ms)	400	8000	160000 ← Geometry Counts
IMMEDIATE	5.84	116.21	2383.03
CACHED	1.05	8.16	188.66
RELATIVE	5.56	14.24	12.63
↑ Render Mode			

Table 4.2: Absolute and relative frame times with cached and immediate mode rendering in static configurations of the *MENGER SPONGE* scene.

In conclusion, frame times in both scenes were significantly reduced by using the caching system without any further optimizations. The next section will compare the durations of scene loading to analyze how much this performance gain costs at startup.

Startup Time

Table 4.3 shows absolute and relative startup times for the *SPHERES* scene with the same configuration as above. The startup time is measured as the duration it takes for loading the scene and performing an initial render traversal, in order to make sure that any resources asynchronously loaded by the graphics driver have actually been uploaded to the *GPU*. The table containing relative duration shows that startup times in immediate mode are always within 90 percent of cached mode when no further cache optimizations are enabled.

Startup Times (s) IMMEDIATE	1000	2197	3375	4913	← Geometry Counts
48	6.19	6.93	7.68	8.53	
528	6.68	8.21	9.76	11.82	
1088	7.37	9.75	12.33	15.83	
1520	7.94	10.94	14.3	18.73	
↑ Triangles/Geometry					

Startup Times (s) CACHED	1000	2197	3375	4913	← Geometry Counts
48	6.27	7.13	8.14	9.11	
528	6.86	8.43	10.25	12.37	
1088	7.5	9.96	12.78	17.3	
1520	8.12	11.53	15.5	20.28	
↑ Triangles/Geometry					

Relative Startup Times	1000	2197	3375	4913	← Geometry Counts
48	0.99	0.97	0.94	0.94	
528	0.97	0.97	0.95	0.96	
1088	0.98	0.98	0.96	0.91	
1520	0.98	0.95	0.92	0.92	
↑ Triangles/Geometry					

Table 4.3: Absolute and relative startup times with cached and immediate mode rendering in various static configurations of the *SPHERES* scene.

The results are a little different for the *MENGER SPONGE* scene as shown in Table 4.4. For the small scene, startup times are still similar, for the medium scene building the cache already takes three times as long as just loading the scene graph, and for the large scene creating the scene with cache takes more than ten times as long because the cache building process completely dominates loading times. Note, however, that cache creation times increase roughly linearly with the size of the scene.

In summary, building the *render cache* will take more time than using the scene graph directly. Yet, depending on the scene this additional cost is only a fraction of the time needed to create and upload other graphics resource such as *vertex buffers* and textures.

Startup Times (s)	400	8000	160000	← Geometry Counts
IMMEDIATE	0.6	0.75	3.71	
CACHED	0.68	2.15	47.08	
RELATIVE	0.88	0.35	0.08	
↑ Render Mode				

Table 4.4: Absolute and relative frame times with cached and immediate mode rendering in static configurations of the *MENGER SPONGE* scene.

Memory Consumption

Table 4.5 shows absolute and relative heap memory consumptions in the *SPHERES* scene. It can be observed that in scenes like these—with large amount of geometry and texture data—the amount of memory needed by the caching system is insignificant in comparison to overall memory usage. The same is true for graphics memory: The caching system will allocate additional *constant buffers* but—as shown in Table 4.6—these contribute very little to graphics memory consumption. In all these cases the immediate-to-cached ratio is very close to one.

Heap Memory (MiB) IMMEDIATE	1000	2197	3375	4913	← Geometry Counts
48	15.76	34.34	52.88	76.2	
528	77.68	170.8	263.57	381.45	
1088	149.99	327.82	503.97	732.67	
1520	204.76	448.8	689.82	1003.22	
↑ Triangles/Geometry					

Heap Memory (MiB) CACHED	1000	2197	3375	4913	← Geometry Counts
48	15.82	35.16	54.17	78.02	
528	77.99	171.08	264	385	
1088	149.63	329.22	507.91	735.59	
1520	204.7	449.45	692.38	1010.62	
↑ Triangles/Geometry					

Relative Heap Memory	1000	2197	3375	4913	← Geometry Counts
48	0.996	0.977	0.976	0.977	
528	0.996	0.998	0.998	0.991	
1088	1.002	0.996	0.992	0.996	
1520	1.000	0.999	0.996	0.993	
↑ Triangles/Geometry					

Table 4.5: Absolute and relative heap memory consumption with cached and immediate mode rendering in various static configurations of the *SPHERES* scene.

Graphics Memory (MiB) IMMEDIATE	1000	2197	3375	4913	← Geometry Counts
48	6.53	14.35	22.04	32.09	
528	65	142.81	219.39	319.36	
1088	132.3	290.65	446.49	649.96	
1520	184.02	404.3	621.07	904.1	
↑ Triangles/Geometry					

Graphics Memory (MiB) CACHED	1000	2197	3375	4913	← Geometry Counts
48	6.65	14.62	22.45	32.69	
528	65.12	143.08	219.8	319.96	
1088	132.42	290.92	446.9	650.56	
1520	184.14	404.56	621.48	904.7	
↑ Triangles/Geometry					

Relative Graphics Memory	1000	2197	3375	4913	← Geometry Counts
48	0.982	0.982	0.982	0.982	
528	0.998	0.998	0.998	0.998	
1088	0.999	0.999	0.999	0.999	
1520	0.999	0.999	0.999	0.999	
↑ Triangles/Geometry					

Table 4.6: Absolute and relative graphics memory consumption with cached and immediate mode rendering in various static configurations of the *SPHERES* scene.

In the *MENGER SPONGE* scene—where the bulk of the data is the scene graph structure while geometry data is just a simple cube shared by all geometry nodes—the caching data makes up for a greater part of the overall memory consumption (see Tables 4.7 and 4.8).

Heap Memory (MiB)	400	8000	160000	← Geometry Counts
IMMEDIATE	0.34	5.5	108.77	
CACHED	0.67	11.99	244.41	
RELATIVE	0.5	0.46	0.45	
↑ Render Mode				

Table 4.7: Absolute and relative heap memory consumption with cached and immediate mode rendering in static configurations of the *MENGER SPONGE* scene.

Overall, the total memory consumption of the caching system can be characterized as modest and is at most $\mathcal{O}(n)$ where n is the number of *render jobs* described by the scene graph.

Graphics Memory (MiB)	400	8000	160000	← Geometry Counts
IMMEDIATE	0.002	0.002	0.002	
CACHED	0.051	0.978	19.533	
RELATIVE	0.036	0.002	0.000	
↑ Render Mode				

Table 4.8: Absolute and relative graphics memory consumption with cached and immediate mode rendering in static configurations of the *MENGER SPONGE* scene.

4.2.2 Caching Baseline - Dynamic Scenes

The previous section has examined the possible speed ups through *scene graph caching* in completely static scenes. This section shows how the caching system holds up as more and more objects in the scene change their position every frame.

Frame Time

First frame times will be investigated in the *SPHERES* scene with different percentages of moving geometries. The results shown in this section are for a scene with 3375 *geometry nodes*. The number of triangles per geometry is fixed at 360. Eight different surfaces are distributed uniformly over the geometries. Table 4.9 shows frame times with and without caching at differ-

Change Rate (%)	IMMEDIATE	CACHED
0	56,64	15,24
20	59,09	16,00
40	60,97	16,03
60	63,19	20,25
80	65,39	25,67
100	67,46	31,17

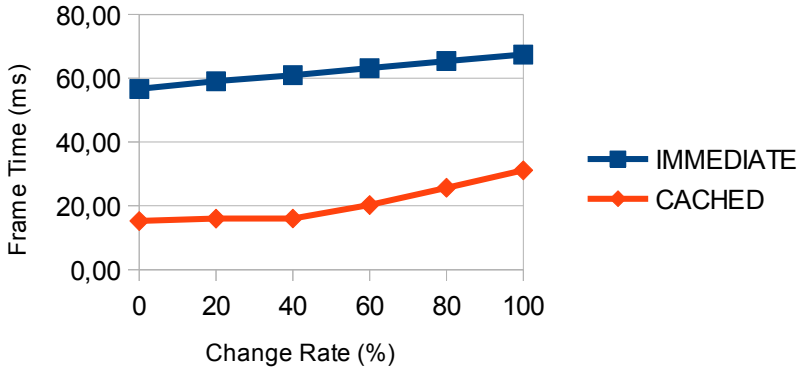


Table 4.9: Frame times (in ms) in the *SPHERES* scene with 3375 geometries at different percentage of rotating spheres.

ent rates of change. It can be observed that frame times with caching enabled are consistently lower than immediate mode rendering at equal change rates. Moreover, even with 100% moving objects the caching system is still twice as fast as the immediate mode renderer at 0% change rate. This shows that for non-structural changes in the scene graph (such as object movement) the *dependency system* allows the caching system to remain an useful optimization.

Frame times from the *MENGER SPONGE* scene show similar results. As can be seen in Table 4.10 frame times in the cached case are consistently lower than in *immediate mode*. And even at its worst the caching system outperforms the competitor at its best by 75%.

Change Rate (%)	IMMEDIATE	CACHED
0	122.355	8.1375
20	129.405	20.53
40	137.6325	32.77
60	143.5025	44.1625
80	149.2725	56.4075
100	155.8225	70.0225

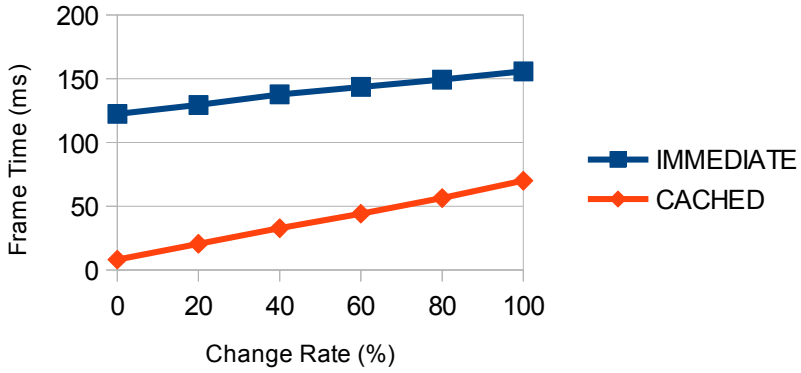


Table 4.10: Frame times (in ms) in the *MENGER SPONGE* scene with 8000 geometries at different percentage of rotating cubes.

Startup Time

Startup time does not increase by much in the *MENGER SPONGE* scene (see Table 4.12) or even not at all for the *SPHERES* scene (see Table 4.11) as more and more objects in the scene are dynamic. This is true for both the cached and the non-cached case.

Change Rate (%)	IMMEDIATE	CACHED
0	8,92	9,46
20	9,00	9,63
40	8,98	9,89
60	9,08	9,54
80	9,00	9,49
100	9,02	9,50

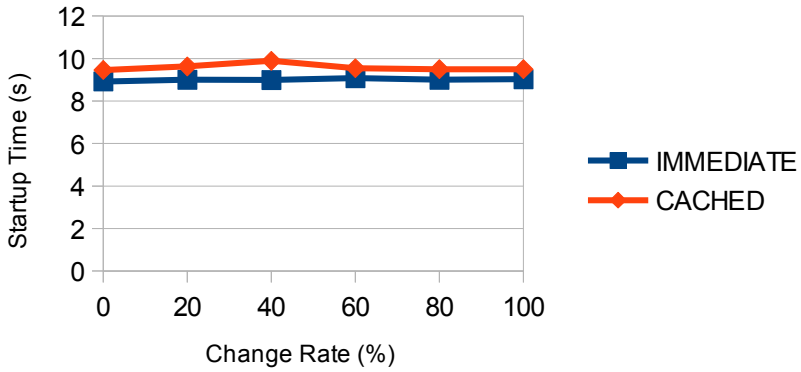


Table 4.11: Startup times (in s) in the *SPHERES* scene with 3375 geometries at different rates of dynamicity.

The difference between the *SPHERES* and the *MENGER SPONGE* scenes can be explained by their structure. The *SPHERES* scene graph is very flat—all leaves having only a single *transformation node* and a single *surface node* above them—while the *MENGER SPONGE* scene has a much deeper hierarchy. Especially the multiple, stacked *transformation nodes* lead to more complex dependency information being allocated. Because *transformation nodes* with no *dependencies* do not contribute to this additional cost, startup time at lower change rates is relatively smaller. In the *SPHERES* scene, resource uploads dominate the whole startup process while the *dependency system* does not contribute a significant part.

Change Rate (%)	IMMEDIATE	CACHED
0	0.76	2.19
20	0.79	2.28
40	0.82	2.39
60	0.87	2.47
80	0.89	2.58
100	0.92	2.67

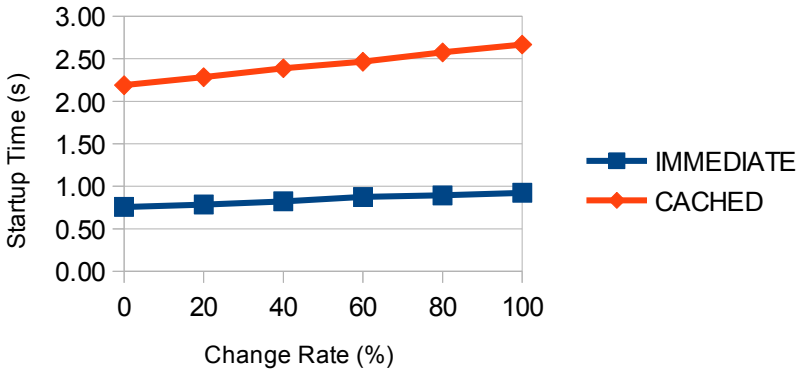


Table 4.12: Startup times (in s) in the *MENGER SPONGE* scene with 8000 geometries at different percentage of rotating cubes.

Memory Consumption

As can be seen in Table 4.13 and Table 4.14 the caching system takes up a little more memory then the immediate mode renderer—as has already been shown in the static case—but increasing the change rate has the same effect on memory consumption in both cached and immediate mode: The relative distance stays the same at different change rates. The caching system does not use disproportionally more memory when dynamicity is involved.

Change Rate (%)	IMMEDIATE	CACHED
0	190,07	191,37
20	191,50	192,26
40	190,80	192,52
60	192,95	193,49
80	192,56	200,94
100	193,26	196,00

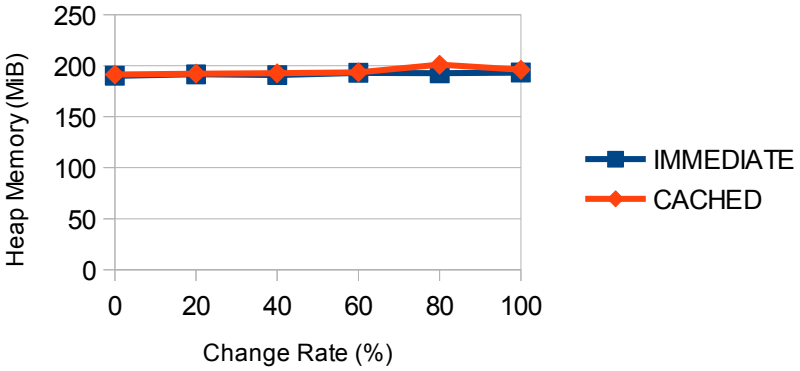


Table 4.13: Heap memory consumption in (MiB) in the *SPHERES* scene with 3375 geometries at different rates of dynamicity.

Change Rate (%)	IMMEDIATE	CACHED
0	5.50	11.99
20	8.10	15.45
40	10.82	19.38
60	12.80	23.11
80	14.86	25.07
100	19.82	28.12

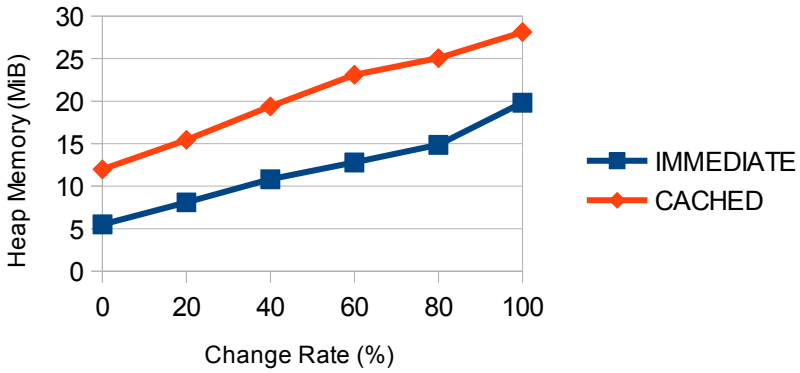


Table 4.14: Heap memory consumption (in MiB) in the *MENGER SPONGE* scene with 8000 geometries at different rates of dynamicity.

Graphics memory stays completely unaffected by the change-rate. In conclusion it can be said that the caching system continues to provide significant performance gains in the presence of non-structural scene graph changes at little to no additional cost compared to the *immediate mode* renderer.

4.2.3 Redundancy Removal

The *Removal of Redundant Instructions* optimization will remove instructions from the stream that do not have any effect. For example, an instruction setting the *pixel shader* to the one that is already set, can be omitted without changing the *instruction streams* semantic. The expected results of this optimization are shorter *instruction streams* and consequently a smaller memory footprint and decreased frame times. Table 4.15 shows frame times with and without *Redundancy Removal* enabled. The *SPHERES* scene is configured to have 3375 geometries with 360 triangles each and 8 different surfaces. The *MENGER SPONGE* scene contains 8000 cubes. In both scenes the camera is the only moving object. It can be observed that rendering is indeed speed up by the optimization, albeit very little in the *SPHERES* scene with only 1.7%. In the *MENGER SPONGE* scene the performance gain is more pronounced at 26.3%.

Frame Times (ms)	SPHERES	MENGER
RR Disabled	15.40	7.56
RR Enabled	15.15	5.99
Relative Speedup	1,7%	26,3%

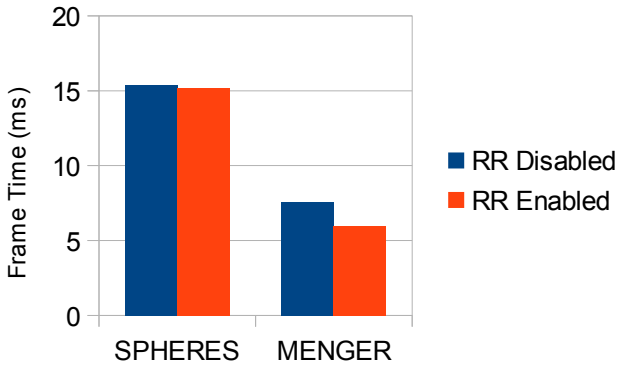


Table 4.15: Frame times with and without the *Redundancy Removal* optimization enabled.

Looking at the startup times in Table 4.16, enabling the optimization somewhat unexpectedly reduces load times. This could be explained by the reduced pressure on the garbage collector, as redundant instructions will be filtered out in the translation step from *semantic* to *native instructions*. Therefore fewer *native instruction* instances are created in the first place. The reduction, however, is not significant. Table 4.17 shows that, as predicted, a smaller memory consumption can be measured. As *instruction streams* do not take up much memory in the first place, this reduction will probably not make much of a difference in practice.

Nonetheless, the absolute numbers of instructions the optimization is able to omit are quite impressive, as Table 4.18 shows: More than eleven thousand for the *SPHERES* scene and nearly fifty-six thousand for the *MENGER SPONGE* scene. In conclusion, there is no reason not to enable this optimization. It will never increase frame times or memory consumption and—as has been shown—does not increase startup time as well.

Startup Times (s)	SPHERES	MENGER
RR Disabled	9.47	2.21
RR Enabled	9.40	2.16

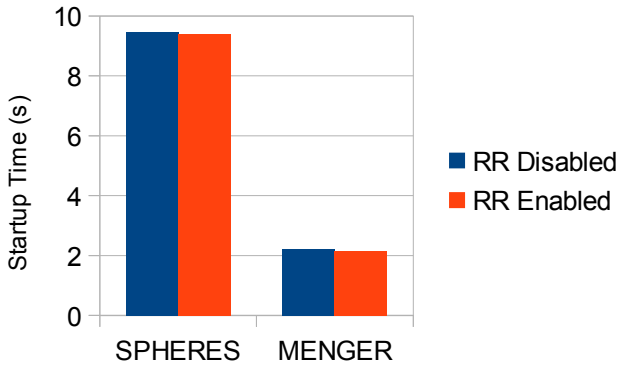


Table 4.16: Startup times with and without the *Redundancy Removal* optimization enabled.

Heap Memory (MiB)	SPHERES	MENGER
RR Disabled	191.36	11.99
RR Enabled	190.78	10.22

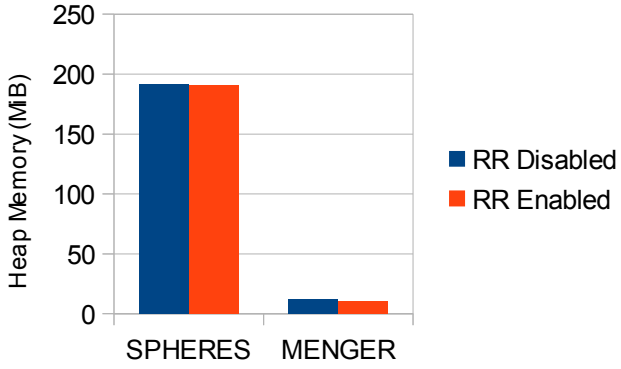


Table 4.17: Heap memory consumption with and without the *Redundancy Removal* optimization enabled.

Instruction Count	SPHERES	MENGER
RR Disabled	37125	88000
RR Enabled	25671	32007
Difference	11454	55993

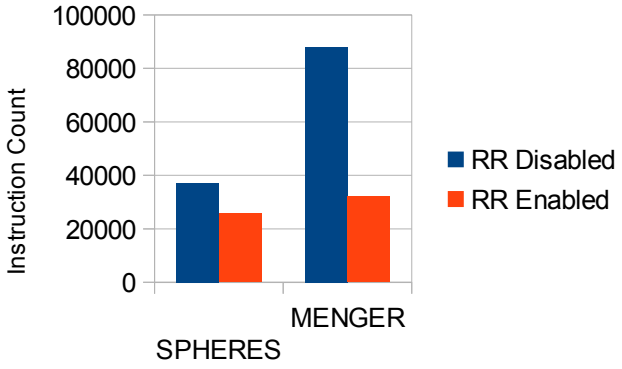


Table 4.18: Number of *native instructions* with and without the *Redundancy Removal* optimization enabled.

4.2.4 State Sorting

The performance gained by the *State Sorting* optimization very much depends on how many state changes there are in the first place and how many of them are semantically unnecessary.

Frame Time

Table 4.19 shows frame times with different numbers of surfaces in the *SPHERES* scene with 3375 geometries and 8 different textures. It can be observed that frame times increase with the number of surfaces in the unsorted case while in the sorted case they stay essentially stable.

Surface Count	SORTED	UNSORTED
1	9.39	8.86
2	9.69	11.96
4	9.60	13.14
8	8.74	13.40

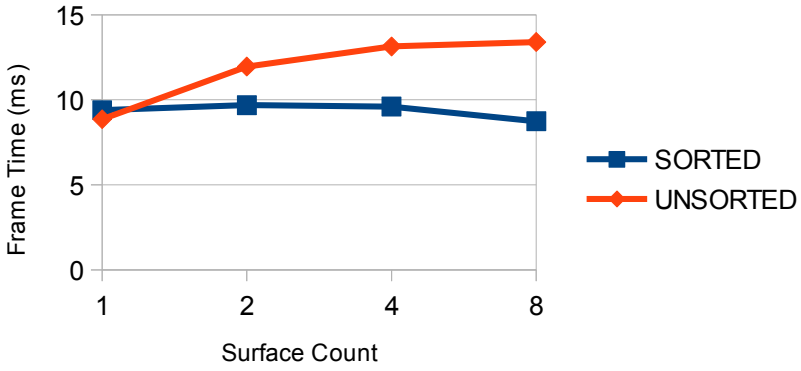


Table 4.19: Frame times (in ms) in the *SPHERES* scene with 3375 geometries with different surface counts.

This can also be seen when varying the number of textures and surfaces at the same time (Table 4.20). The more textures, the longer frame times, the more surfaces, the longer frame times—in the unsorted case. In the sorted case frame times again are stable independent of texture and surface counts.

SORTED	1	2	4	8	← Textures
1	9.39	9.44	9.46	9.72	
2	9.69	8.65	9.56	9.72	
4	9.60	8.66	8.61	9.65	
8	8.74	8.64	8.57	9.58	
↑Surfaces					

UNSORTED	1	2	4	8	← Textures
1	8.86	9.51	9.79	10.32	
2	11.96	11.68	12.97	13.45	
4	13.14	13.10	14.18	14.64	
8	13.40	14.30	14.59	15.47	
↑Surfaces					

Table 4.20: Frame times (in ms) in the *SPHERES* scene with 3375 geometries with different surface and texture counts.

In the case of the *MENGER SPONGE* scene the picture is quite different. In this scene there is only one surface and one set of textures available. Accordingly, state sorting cannot reduce any state changes. As a result, the optmization does have no significant effect on frame times as can be seen in Table 4.21.

	SORTED	UNSORTED
Frame Time (ms)	7,49	7,6075

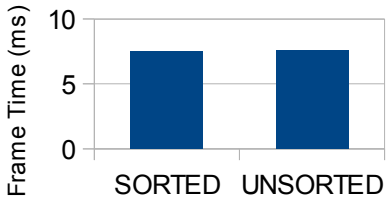


Table 4.21: Frame times in the *MENGER SPONGE* scene with 8000 geometries with and without state sorting.

Startup Time

Overall state sorting does not contribute much to startup times. Table 4.22 shows that there is little difference for the *SPHERES* scene with 3375 geometries, 360 triangles per geometry, 8 textures and varying surface counts.

Surface Count	SORTED	UNSORTED
1	9.07	9.00
2	9.24	9.08
4	9.37	9.36
8	9.77	9.62

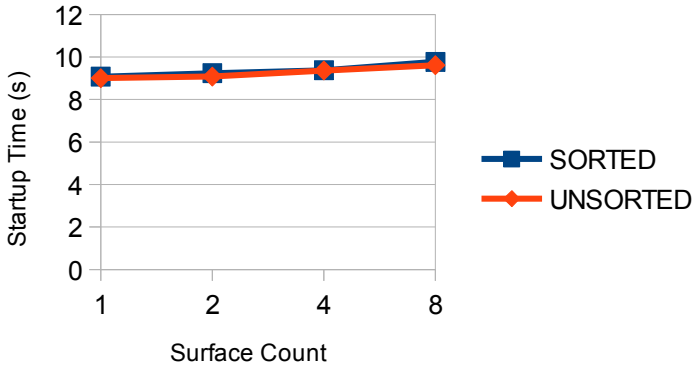


Table 4.22: Startup times (in s) in the *SPHERES* scene with 3375 geometries with different surface counts.

	SORTED	UNSORTED
Startup Time (s)	2,56	2,23

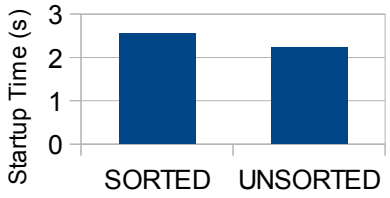


Table 4.23: Startup times in the *MENGER SPONGE* scene with 8000 geometries with and without state sorting.

In the *MENGER SPONGE* scene the results are similar. Table 4.23 shows that state sorting costs some additional time when building the cache but for a one-time penalty the difference is rather small at 330ms.

Instruction Count

Another interesting feature of state sorting is its effect on the *Removal of Redundant Instructions* optimization. Table 4.24 shows a comparison of instruction counts with different combinations of the two optimizations. Sorting *render jobs* by state leads to an *instruction stream* with much more redundancy, thus providing the redundancy removal optimization with better input data.

	Instruction Count
No Opt	37125
RR	25671
State | RR	16902

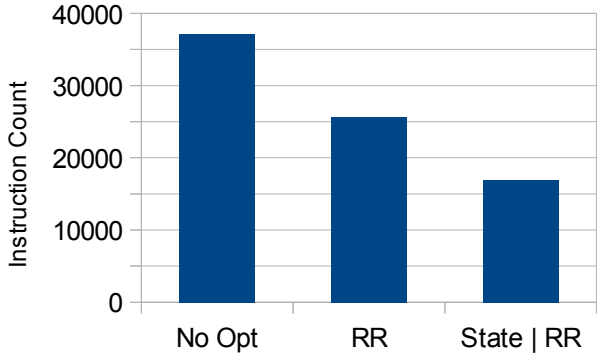


Table 4.24: Instruction counts in the *SPHERES* scene at 3375 geometries with 8 surfaces and textures with different optimizations enabled.

4.2.5 Super Instructions

The *Super Instructions* optimization combines a sequence of instructions into a single one, thus reducing instruction dispatch overhead, such as interface calls and parameter casting. However, as Table 4.25 shows the effect is small and not always positive. In the *SPHERES* scene with 3375 geometries and 360 triangles per geometry, using *super instruction* even seems to make the rendering process slower. In the *MENGER SPONGE* scene with 8000 geometries rendering is faster, although only marginally. The reduction of heap memory consumption (Table 4.26) also is very small. Consequently, we do not recommend to enable this optimization by default.

Frame Times (ms)	SPHERES	MENGER
SI Enabled	15,72	8,73
SI Disabled	15,34	9,10

Table 4.25: Frame Times in both synthetic test scenes with the *Super Instructions* optimization enabled and disabled.

Heap Memory (MiB)	SPHERES	MENGER
SI Enabled	190,90	10,89
SI Disabled	191,13	11,99

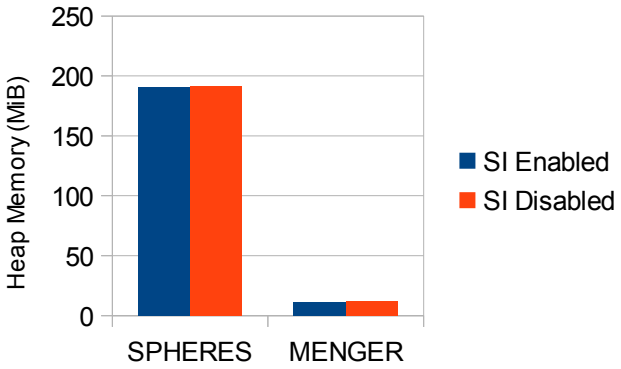


Table 4.26: Heap memory consumption in both synthetic test scenes with the *Super Instructions* optimization enabled and disabled.

4.2.6 Overdraw Sorting

Even though the *NVIDIA GPU Programming Guide* recommends to prioritize *front-to-back rendering* over sorting by state [16, p. 20], the *Overdraw Sorting* optimization does not seem to bring any performance gains. As Tables 4.27 and 4.28 show, enabling the optimization even causes a slight performance drop in both the *SPHERES* and the *MENGER SPONGE* scene. This suggests that the CPU overhead of sorting geometries front-to-back is greater than any time savings on the GPU side. It is still possible that the optimization might be useful at higher resolutions,[2] or with more expensive *pixel shaders* but in the general case, we do not recommend to enable *overdraw sorting*.

Geometry Count	SORTED	UNSORTED
1000	3,01	2,86
2197	6,36	5,67
3375	9,75	9,29

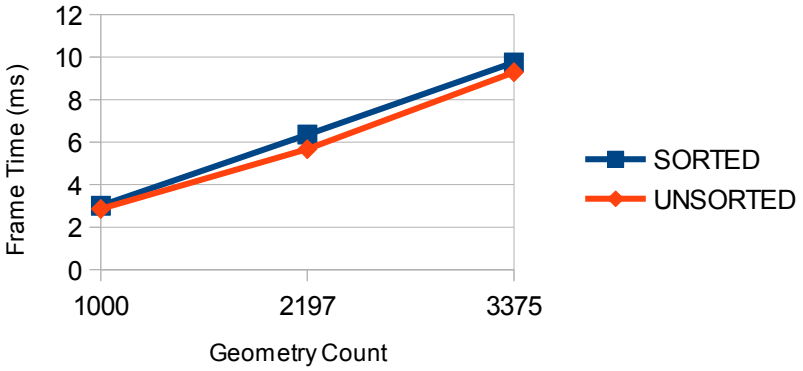


Table 4.27: Frame times (in ms) in the *SPHERES* scene using the *Overdraw Sorting* optimization at different geometry counts.

Geometry Count	SORTED	UNSORTED
400	1,36	1,06
8000	6,64	6,00
160000	211,47	187,51

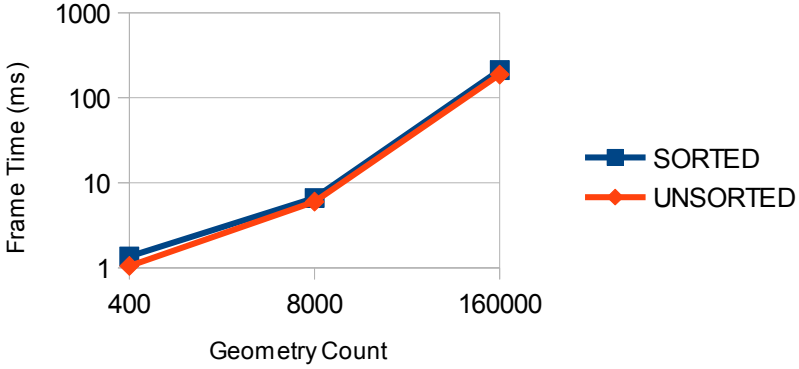


Table 4.28: Frame times (in ms) in the *MENGER SPONGE* scene using the *Overdraw Sorting* optimization at different geometry counts.

[2] These tests where performed at 1920 by 1200 pixels without any super sampling.

4.2.7 Memoization

This optimization works by memoizing intermediate results of transformation matrix multiplications and it shows its strengths in dynamic settings.

Frame Time

Table 4.29 shows that this optimization can keep frame times more stable as more and more objects in the *SPHERES* scene (3375 geometries, 360 triangles per geometry, 8 surfaces, 8 textures) are moving. Table 4.30 shows similar results for the *MENGER SPONGE* scene (8000 cubes). Starting out at equal frame times with 0% change rate, the memoized version is 30% faster at 100% change rate.

Change Rate (%)	MEMOIZED	COMPUTED
0	15,71	15,53
20	16,33	16,28
40	16,86	16,94
60	17,30	21,11
80	21,14	27,00
100	25,30	35,38

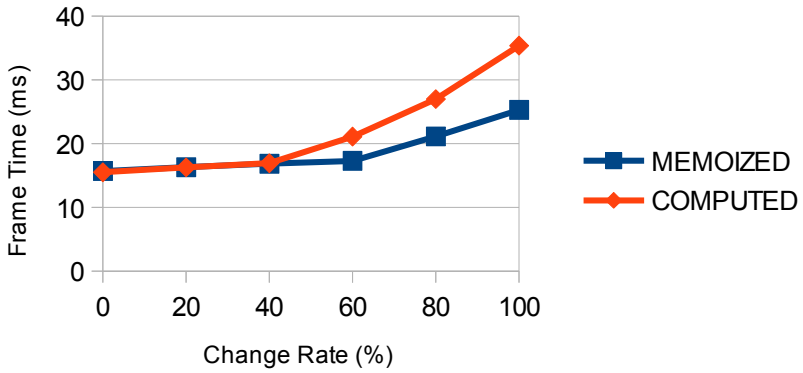


Table 4.29: Frame times (in ms) in the *SPHERES* scene using the transformation matrix *Memoization* at different rates of dynamicity.

Change Rate (%)	MEMOIZED	COMPUTED
0	7,56	7,57
20	14,97	18,92
40	22,55	30,76
60	29,53	41,38
80	36,89	53,04
100	44,73	64,75

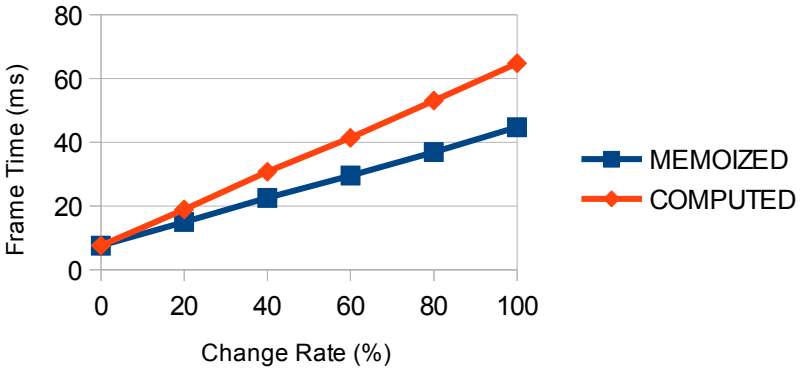


Table 4.30: Frame times (in ms) in the *MENGER SPONGE* scene using the transformation matrix *Memoization* at different rates of dynamicity.

Startup Time

The time needed to build the scene is increased by enabling the optimization, albeit the difference is very small in the *SPHERES* scene (Table 4.31) and does not clearly correlate with the change rate (which directly influences the size of the memoization table that needs to be created) in the *MENGER SPONGE* scene (Table 4.32).

Change Rate (%)	MEMOIZED	COMPUTED
0	9,98	9,42
20	10,37	10,14
40	10,82	10,47
60	10,90	10,59
80	11,20	11,19
100	13,81	12,10

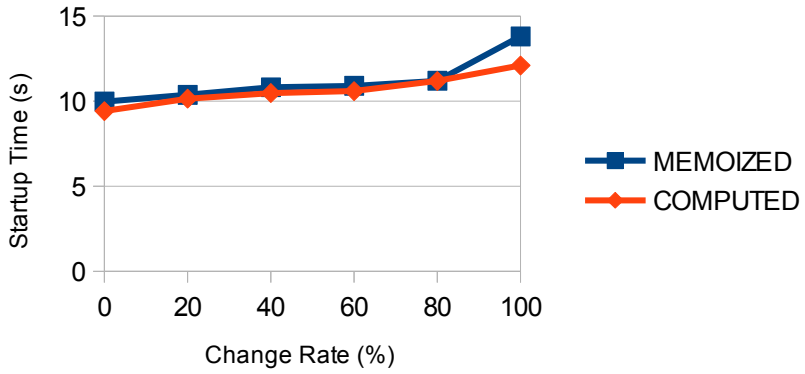


Table 4.31: Startup times (in s) in the *SPHERES* scene using the transformation matrix *Memoization* at different rates of dynamicity.

Change Rate (%)	MEMOIZED	COMPUTED
0	2,26	2,20
20	2,85	2,26
40	2,56	2,40
60	2,57	2,54
80	2,89	2,57
100	2,77	2,70

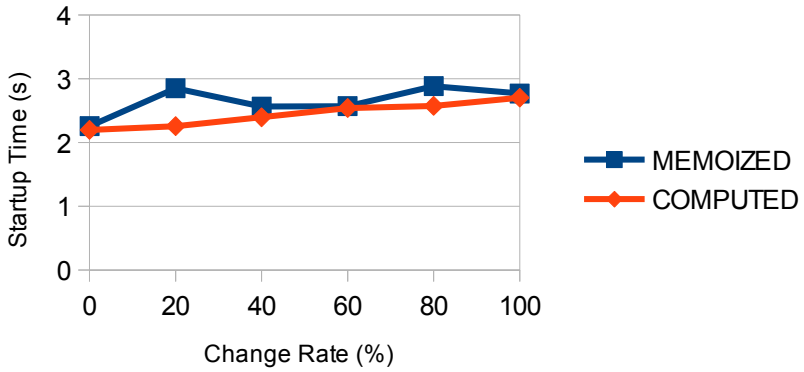


Table 4.32: Startup times (in s) in the *MENGER SPONGE* scene using the transformation matrix *Memoization* at different rates of dynamicity.

Memory Consumption

A particularly interesting question concerning this optimization is how much addition memory the memozation table costs. For the *SPHERES* scene, where *transformation nodes* are not stacked, the additional memory consumption is rather small: around 2 MiB in most cases which is under 1% of the total memory consumption (Table 4.33). In the *MENGER SPONGE* scene with its deeper *transformation node* hierarchy, the difference is more pronounced at 6 to 8 MiB (Table 4.34)

Change Rate (%)	MEMOIZED	COMPUTED
0	193,19	191,13
20	193,98	192,89
40	194,22	192,52
60	195,17	194,99
80	202,54	194,81
100	197,62	195,71

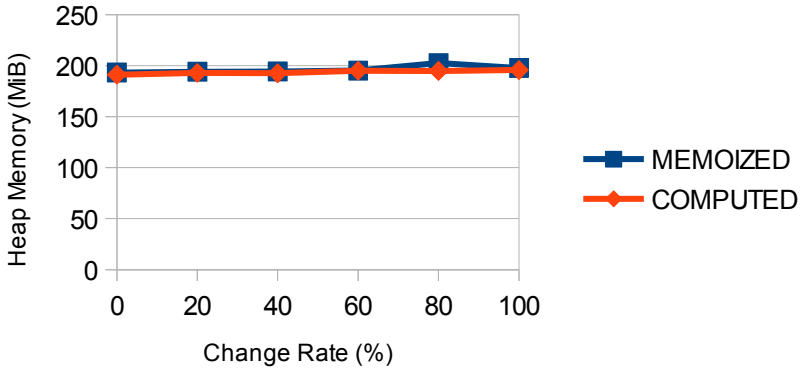


Table 4.33: Heap memory consumption (in MiB) in the *SPHERES* scene using the transformation matrix *Memoization* at different rates of dynamicity.

Change Rate (%)	MEMOIZED	COMPUTED
0	17,34	11,99
20	21,63	15,55
40	26,39	19,23
60	30,70	22,09
80	33,13	25,07
100	36,81	30,83

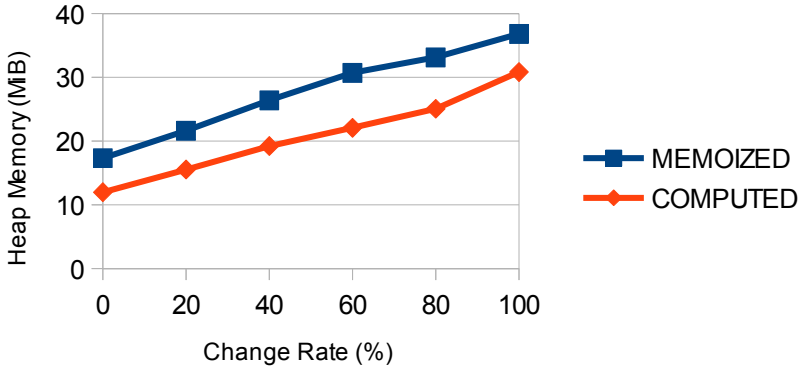


Table 4.34: Heap memory consumption (in MiB) in the *MENGER SPONGE* scene using the transformation matrix *Memoization* at different rates of dynamicity.

In conclusion, this optimization can bring noticeable drops in frame time when many objects in the scene are moving. At the same time, the additional cost in startup time and memory consumption is not significant on modern hardware.

4.2.8 Interleaved Resource Uploads

Instead of issuing them all at once, this optimization distributes resource uploads (which are mostly for *constant buffers*) evenly over the execution of the *instruction stream*. As Tables 4.35 and 4.36 show, this results in a small but consistent drop in frame times. The optimization was mostly implemented to enable *Concurrent Resource Value Computation*. However, since it does not cost additional memory or startup time, it can be enabled by default.

Change Rate (%)	INTERLEAVED	PRELOADED
0	15,96	15,74
20	15,78	16,52
40	16,11	16,51
60	20,24	21,31
80	25,74	27,22
100	30,99	33,17

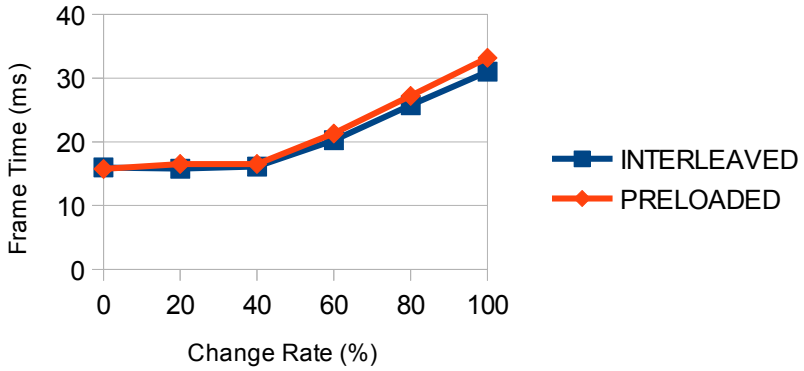


Table 4.35: Frame times (in ms) in the *SPHERES* scene (3375 geometries, 360 triangles per geometry) using the *Interleaved Resource Upload* optimization at different rates of dynamicity.

Change Rate (%)	INTERLEAVED	PRELOADED
0	7,60	7,62
20	18,86	19,25
40	29,56	30,94
60	40,46	42,56
80	50,39	54,23
100	61,99	66,46

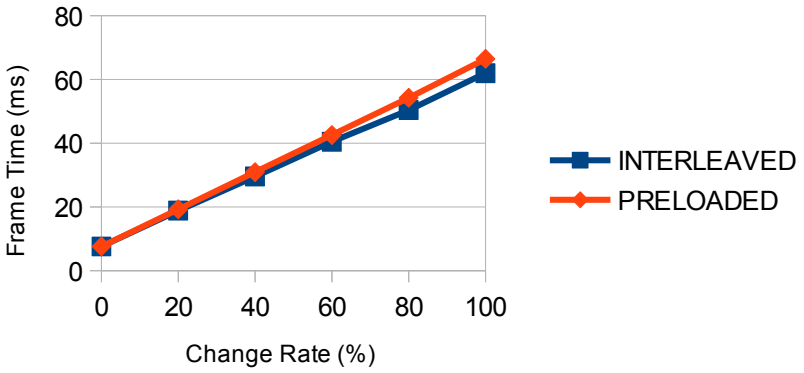


Table 4.36: Frame times (in ms) in the *MENGER SPONGE* scene (8000 cubes) using the *Interleaved Resource Upload* optimization at different rates of dynamicity.

4.2.9 Concurrent Resource Value Computation

This optimization uses a number of parallel worker threads to execute the *update actions* of those *dependent resources* that are out of date. Tables 4.37 and 4.38 show that one worker thread computing resource values—while the main thread starts executing the instruction stream—cannot provide any significant performance gain over executing *update actions* and the *instruction stream* sequentially. The additional synchronization overhead caused by multi-threaded execution seems to cancel out any benefits provided by the additional core. Adding a second and a third thread, however, does incur a frame time drop: In both cases the frame time is reduced to around a half of the original value. In settings with high dynamicity this is one of the most useful optimizations—provided appropriate hardware is available.[3]

Thread Count →	0	1	2	3
0	15,05	15,09	15,09	15,19
20	15,52	15,31	15,72	15,62
40	15,97	15,50	16,08	16,07
60	21,04	19,79	16,05	16,24
80	26,54	25,10	17,11	16,70
100	32,00	30,06	20,15	17,90
↑ Change Rate (%)				

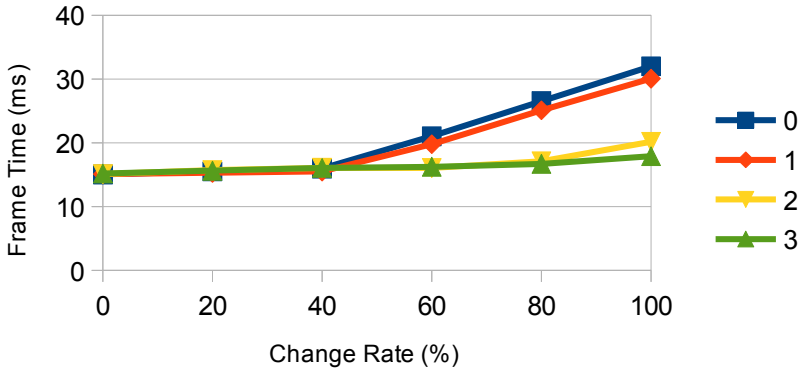


Table 4.37: Frame times (in ms) in the *SPHERES* scene (3375 geometries, 360 triangles per geometry) using the *Concurrent Resource Value Computation* optimization at different rates of dynamicity.

Thread Count →	0	1	2	3
0	7,92	7,61	7,46	8,03
20	18,89	18,53	14,41	13,69
40	29,19	29,15	21,01	18,90
60	38,56	38,88	26,91	23,52
80	48,31	48,93	32,90	28,10
100	58,29	59,36	38,88	32,72
↑ Change Rate (%)				

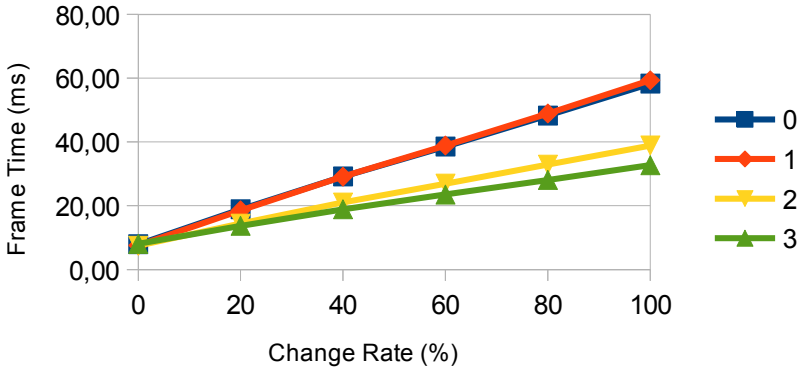


Table 4.38: Frame times (in ms) in the *MENGER SPONGE* scene (8000 cubes) using the *Concurrent Resource Value Computation* optimization at different rates of dynamicity.

[3] These test cases were run on an *Intel Core i5 2400* with 4 CPU cores.

4.2.10 Transparent Geometry

For transparent geometry it turns out that (in the current implementation) the immediate mode renderer is just as fast as the *TransparencyPassCache*. Table 4.39 shows only marginal differences at varying geometry counts in the *SPHERES* scene (360 triangles per geometry, 1 surface, 8 textures).

Geometry Count	CACHED	UNCACHED
125	4,76	5,52
512	20,31	23,92
1000	53,48	53,33

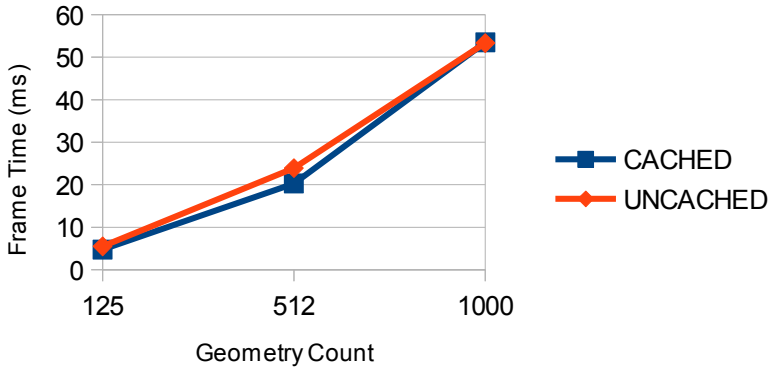


Table 4.39: Frame times (in ms) in the *SPHERES* scene (360 triangles per geometry, 1 surface, 8 textures) using transparency sorting.

Closer profiling shows that sorting and updating the *index buffers*—which has to be done for cached and immediate rendering alike—clearly dominates the rendering process, not allowing the faster *instruction stream* execution to give any performance advantage. Both versions have to wait for the *index buffers* to be uploaded.

4.2.11 Optimizations Combined

This last section will compare the recommended set of default optimizations to immediate mode and cached rendering without any optimizations. Comparisons will be done for the *SPHERES* scene at 3375 geometries, 360 triangles per geometry, and 8 surfaces using 8 different textures. The *MENGER SPONGE* scene is configured to have 8000 cubes. All tests were run at different percentages of objects moving every frame.

Frame Time

Table 4.40 and Figure 4.4 show absolute frame times in the *SPHERES* scene. At zero percent dynamicity, cached rendering is about 4 to 5 times faster than the *immediate mode* renderer. Also note, that the enabled optimizations (*Memoization*, *State Sorting*, *Removal of Redundant Instructions*, and *Interleaved Resource Upload*) can decrease the frame time by a third relative to unoptimized caching. Looking at the previous test results for the various optimizations, this performance gain is most probably solely due to *state sorting*, as the other optimizations do not influence frame time at 0% dynamicity (*memoization*), or only have a very small effect on it in general (*redundancy removal* and *interleaved resource uploads*). The *Concurrent Resource Value Computation* optimization too does not decrease frame time for the non-dynamic setup, as no resource values (apart from the camera matrices) need to be computed.

SPHERES SCENE (Frame Times in ms)

Change Rate (%)	NO CACHING	JUST STREAM	MEMO | RR | SS | IL (1 Core)	MEMO | RR | SS | IL (4 Cores)
0	54,58	15,40	9,58	10,24
20	56,63	15,82	8,95	10,61
40	58,51	16,26	10,27	10,65
60	59,81	20,77	13,38	10,85
80	61,80	26,50	16,78	11,05
100	63,50	32,04	20,02	12,60

MEMO ... Memoization RR ... Redundancy Removal
SS ... State Sorting IL ... Interleaved Uploads

Table 4.40: Frame times (in ms) in the *SPHERES* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

At increasing rates of dynamicity it is the *concurrent resource value computation* optimization that has the most prominent effect. It can keep the frame time almost constant across all rates of dynamicity. *Memoization*, on the other hand, does not seem to provide any performance gain in this scene: The speedup relative to unoptimized caching stays around 1.6 across the board.

The picture is similar for the *MENGER SPONGE* scene (Table 4.41 and Figure 4.5) although there are some notable differences. Cached rendering again is considerably faster than the immediate mode renderer: about 14 times as fast for unoptimized caching and about 20 times as

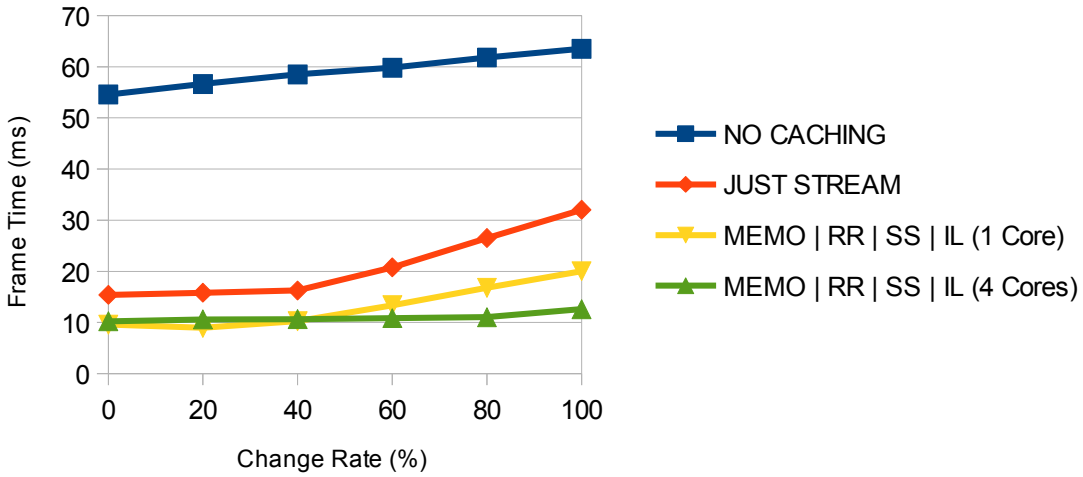


Figure 4.4: Frame times (in ms) in the *SPHERES* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

fast with fully optimized caching. This time *state sorting* can reduce frame time by about 25% for cached rendering.

MENGER SPONGE SCENE (Frame Times in ms)

Change Rate (%)	NO CACHING	JUST STREAM	MEMO | RR | SS | IL (1 Core)	MEMO | RR | SS | IL (4 Cores)
0	118,89	8,16	6,00	5,91
20	125,81	19,55	9,74	6,79
40	133,57	30,73	15,28	9,84
60	138,39	41,02	20,19	12,46
80	146,15	51,72	25,40	15,27
100	150,82	62,62	29,45	17,35

MEMO ... Memoization
SS ... State Sorting
RR ... Redundancy Removal
IL ... Interleaved Uploads

Table 4.41: Frame times (in ms) in the *MENGER SPONGE* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

In the *MENGER SPONGE* scene with its deep transformation hierarchy the *memoization* optimization brings more of a performance gain than in the *SPHERES* scene. Starting at 136% percent relative speed at 0% dynamicity, the relative speed grows 213% at 100% dynamicity for the single core configuration. Again, *redundancy removal* and *interleaved resource uploads* do not change with different rates of dynamicity. The *concurrent resource value computation*

optimization can bring another drop in frame times at higher change rates.

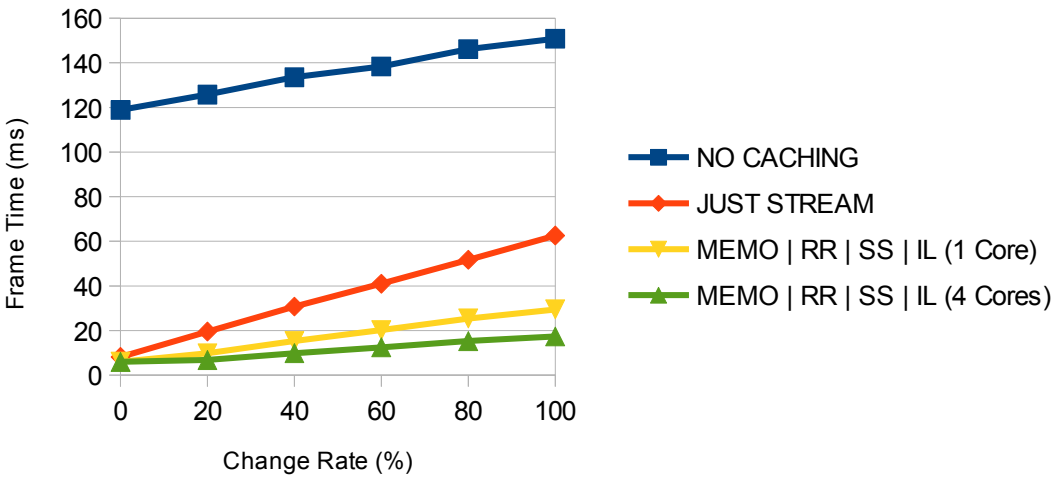


Figure 4.5: Frame times (in ms) in the *MENGER SPONGE* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

In conclusion, for scenes with a high number of individual geometry nodes, the biggest performance gain is achieved by using *instruction streams* instead of a regular scene graph traversal. This could increase the frame rate from 4 times as fast in the *SPHERES* scene to 14 times as fast in the *MENGER SPONGE* scene. Adding the *state sorting* optimization could bring another 25% to 33%. As the percentage of moving objects in the scene increased, the other optimizations too start to play a role. Most notably *memoization* for the *MENGER SPONGE* scene, and *concurrent resource value computation* for both scenes.

With all objects moving cached rendering (all optimizations enabled) was still 5 times as fast as the immediate mode renderer for the *SPHERES* scene and more than 8 times as fast in the *MENGER SPONGE* scene. This shows that *scene graph caching*, as proposed in this work, is a viable solution also for dynamic scenes, as long as there are no structural changes in the scene graph. The next section will examine the costs of these relative speed ups.

Startup Time

As already shown in the previous tests, building *render caches* will consume some additional time when loading a scene. Table 4.42 and Figure 4.6 show startup times for the *SPHERES* scene. There is no clear correlation between startup times and dynamicity. Enabling optimizations for the cache will cost additional time. In absolute numbers, however, creating the cache does only cost between 1 second (unoptimized) and 3 seconds (fully optimized), with cache creation taking up 15% to 25% of the total startup time.

SPHERES SCENE (Startup Times in s)

Change Rate (%)	NO CACHING	JUST STREAM	MEMO | RR | SS | IL (1 Core)	MEMO | RR | SS | IL (4 Cores)
0	8,92	9,44	12,39	11,58
20	8,95	9,68	11,50	11,20
40	8,97	9,86	11,08	11,05
60	8,92	9,75	11,24	11,11
80	8,93	9,63	11,05	10,87
100	8,95	9,69	10,59	10,59

MEMO ... Memoization
SS ... State Sorting
RR ... Redundancy Removal
IL ... Interleaved Uploads

Table 4.42: Startup times (in s) in the *SPHERES* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

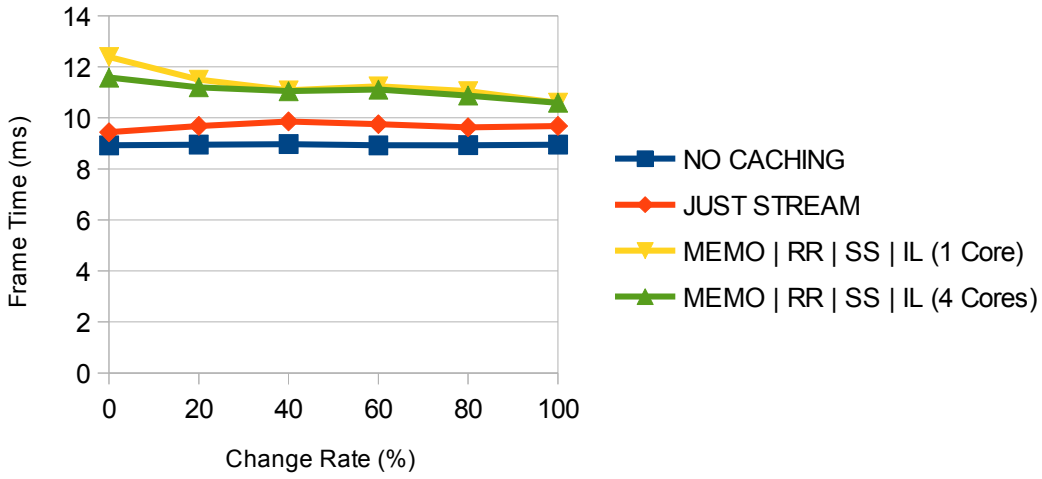


Figure 4.6: Startup times (in s) in the *SPHERES* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

In the *MENGER SPONGE* scene, building the cache causes a higher increase in startup times (Table 4.43 and Figure 4.7). While loading the scene without cache takes under 1 second at all dynamicity rates, doing so with an unoptimized cache takes more than 2 seconds (185%-197% increase) and with full optimization its more than 4 seconds (370%-500% increase). This high impact of cache building on the load time can be explained by the little data that needs to be uploaded to GPU memory: the same, small cube geometry is shared by all *geometry nodes* and only uses one diffuse texture and one normal map). At the same time the scene has a much higher structural complexity than the *SPHERES* scene, which is what influences cache building the most: the number of *render jobs* and *dependent resources* it needs to allocate.

MENGER SPONGE SCENE (Startup Times in s)

Change Rate (%)	NO CACHING	JUST STREAM	MEMO | RR | SS | IL (1 Core)	MEMO | RR | SS | IL (4 Cores)
0	0,75	2,23	2,59	4,50
20	0,78	2,26	2,71	4,58
40	0,82	2,40	2,99	4,80
60	0,84	2,40	2,89	4,70
80	0,87	2,51	3,01	4,66
100	0,92	2,65	3,08	4,33

MEMO ... Memoization RR ... Redundancy Removal
SS ... State Sorting IL ... Interleaved Uploads

Table 4.43: Startup times (in s) in the *MENGER SPONGE* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

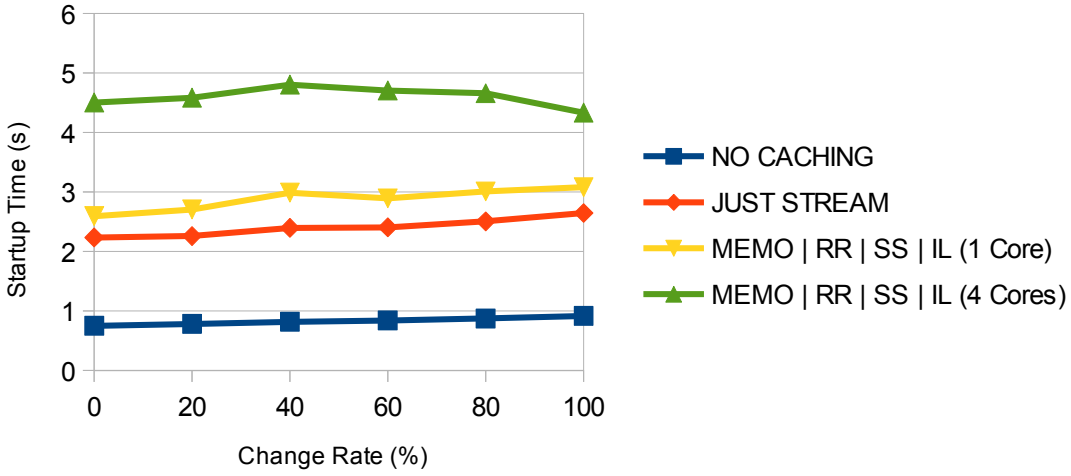


Figure 4.7: Startup times (in s) in the *MENGER SPONGE* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

Memory Consumption

The same factors that cause the difference between *SPHERES* and *MENGER SPONGE* scene in startup times also have a similar effect on heap memory consumption. As Table 4.44 and Figure 4.8 show, in the *SPHERES* scene with its flat structure and large geometry data, caching does not raise memory consumption by much (at most 2.36% increase). Contrastingly, in the *MENGER SPONGE* scene with its deep hierarchy, caching becomes a significant memory factor with between 120% and 170% memory consumption increase (see Table 4.45 and Figure 4.9).

SPHERES SCENE (Heap Memory in MiB)

Change Rate (%)	NO CACHING	JUST STREAM	MEMO | RR | SS | IL (1 Core)	MEMO | RR | SS | IL (4 Cores)
0	189	190	192	191
20	190	192	193	193
40	191	193	194	194
60	192	194	196	196
80	194	197	198	198
100	195	197	199	199

MEMO ... Memoization
SS ... State Sorting
RR ... Redundancy Removal
IL ... Interleaved Uploads

Table 4.44: Heap memory consumption (in MiB) in the *SPHERES* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

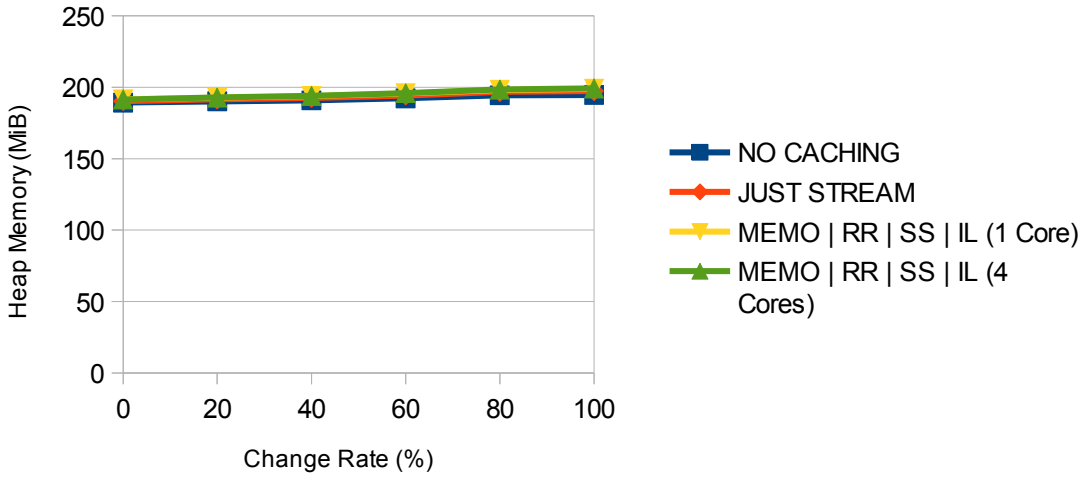


Figure 4.8: Heap memory consumption (in MiB) in the *SPHERES* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

MENGER SPONGE SCENE (Heap Memory in MiB)

Change Rate (%)	NO CACHING	JUST STREAM	MEMO | RR | SS | IL (1 Core)	MEMO | RR | SS | IL (4 Cores)
0	5,5	12,0	15,1	15,1
20	8,1	15,4	19,5	20,0
40	10,7	18,7	24,4	24,0
60	13,3	23,1	29,7	29,8
80	15,2	25,4	33,2	33,7
100	18,4	28,9	37,5	38,1

MEMO ... Memoization
SS ... State Sorting
RR ... Redundancy Removal
IL ... Interleaved Uploads

Table 4.45: Heap memory consumption (in MiB) in the *MENGER SPONGE* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

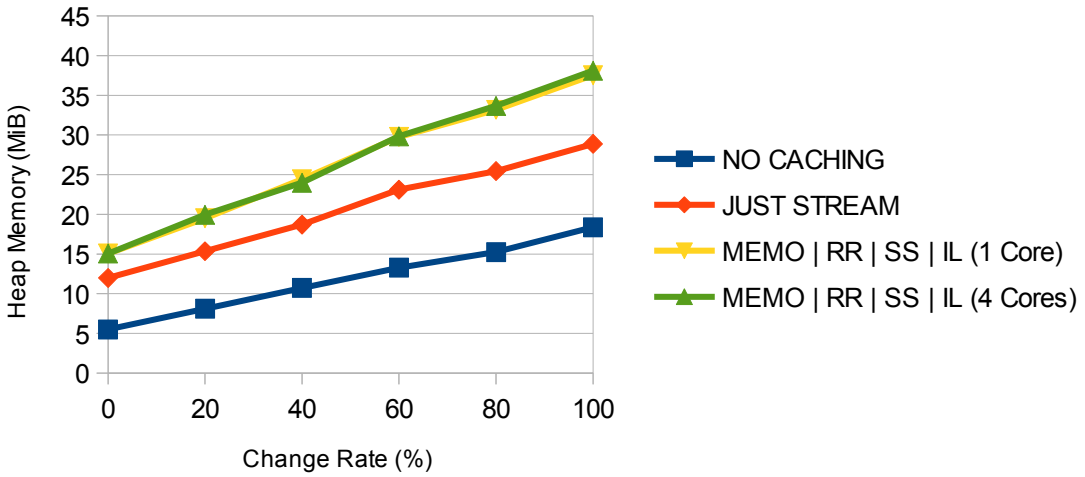


Figure 4.9: Heap memory consumption (in MiB) in the *MENGER SPONGE* scene at different rates of dynamicity with various combinations of caching optimizations enabled.

For graphics memory the picture is quite different. The cache only needs to store a higher number of *constant buffers* than the immediate mode renderer. All other graphics resources are used the same way by both versions. In the case of the *SPHERES* scene this means a difference of 420 KiB of graphics memory (see Table 4.46) and for the *MENGER SPONGE* scene the difference is 1 MiB (see Table 4.47).

SPHERES SCENE (Graphics Memory in MiB)

	NO CACHING	ANY CACHING
Graphics Memory (MiB)	620,23	620,64

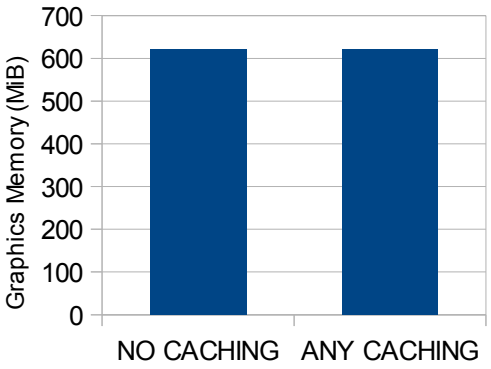


Table 4.46: Graphics memory (in MiB) in the *SPHERES* scene with and without caching.

MENGER SPONGE SCENE (Graphics Memory in MiB)

	NO CACHING	ANY CACHING
Graphics Memory (MiB)	42,67	43,64

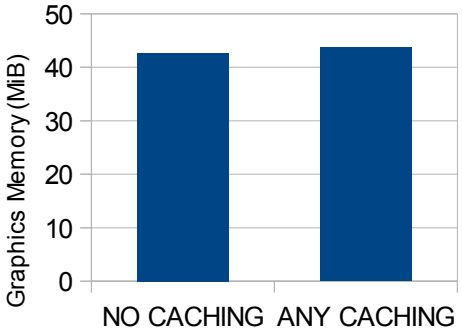


Table 4.47: Graphics memory (in MiB) in the *MENGER SPONGE* scene with and without caching.

CHAPTER 5

Conclusion

The aim of this work was to develop a way of improving rendering performance for scene graph based graphics applications. The chosen strategy to achieve this goal was to implement so-called *render caches* which contain an optimized representation of some part of a scene graph. This allows to render the cached subgraph without traversing it. Internally, *render caches* use *instruction streams* to capture the semantic of a render traversal in a form that allows for much faster execution of the same drawing commands. As traversal cost makes many scene graph applications CPU-bound, this exclusion of subgraphs from the render traversal can bring major overall performance gains.

Instruction streams were presented as a sequence of instructions, each encapsulating a call into the low-level graphics *API* and the arguments needed for this call. This linear list of commands can represent a program rendering a scene graph much like byte code can represent a generic program previously in the form of an abstract syntax tree.

The investigation of the circumstances under which a *render cache* becomes out-of-date yielded that even small changes, even outside of the cached subgraph can necessitate the *render cache* to be fully rebuilt. As rebuilding the cache from scratch is a costly operation this poses a major limitation of the possible use cases of the system. In an attempt to alleviate this problem the so-called *dependency system* was proposed. The *dependency system* provides means for annotating parts of a scene graph with dynamic predicates—so-called *dependencies*—which allow tracking changes of the annotated entities. In the terms of the *dependency system* these annotated entities are called *value sources*. Common examples of *value sources* are *transformation* and *camera nodes*. Using this metadata describing changes in the scene graph, the *dependency system* allows to lazily update so-called *dependent resources*, which can be *vertex*, *index* or *constant buffers* or other graphics resources.

Some important and useful properties of the *dependency system* have been examined:

- A notion of *dependent resource* equality has been established and proven to satisfy criteria which allow to optimally reuse *dependent resource* instances within a given context.

- A construction mechanism for *dependent resources* has been presented which implements the aforementioned instance reuse with an asymptotic complexity of $\mathcal{O}(1)$.

- It has been shown that *dependent resources* can automatically be composed in a *product type*-like manner without loosing any of their useful and defining properties: a tuple of *dependent resources* is again a *dependent resource*. This allows to naturally reason about composite resources like *constant buffers* consisting of multiple fields.

- The necessary evaluation order of *dependency* and *dependent resource* updates has been investigated and a simple scheme was suggested which implements the requirements found.

In the chapter *Dependency-Aware Scene Graph Caching* it has been shown how the *dependency system* naturally integrates with the concepts of *render caches* and *instruction streams*. By using *dependent resources* as instruction arguments, *render caches* can stay up-to-date at small cost in reaction to the most frequent class of scene graph changes which comprises common activities such as object or camera movement.

Instruction streams and their execution can be optimized in various ways. The section *Optimizations* investigates a number of such optimizations and explains their implementation in the prototype:

- *Removal of Redundant Instructions* reduces the number of instructions in the stream which results in fewer calls to the graphics *API*.

- *Super Instructions* combine a sequence of consecutive instructions into a single one, resulting in less instruction execution overhead.

- *State Sorting* prepares the *instruction stream* in a way as to reduce the number of graphics pipeline state changes.

- *Overdraw Sorting* allows to re-sort the *instruction stream* for front-to-back rendering in order to exploit the *Z culling* hardware optimization.

- *Concurrent Resource Value Computation* enables *render caches* to update graphics resources on multiple CPU cores simultaneously.

- *Memoization* works by caching intermediate results of transformation matrix computations which improves performance in scenes with deep transformation hierarchies.

- *Disposing Cached Subgraphs* shows that *render caches* can completely take over all rendering responsibilities from static cached subgraph, allowing to reduce memory consumption.

- Finally, *Cached Culling Hierarchy* shows that the caching system supports *view frustrum culling* at least as well as the existing *immediate mode* renderer, with potential to surpass it in certain dynamic scenarios.

At the end of the main part, the two default types of *render caches*—based on the concepts described so far—were presented, namely the *SolidPassCache* and the *TransparencyPassCache*.

The last part of the thesis examined the actual performance characteristics of the *scene graph caching* system using a number of synthetic test scenes in different configurations. These tests showed that the caching system is most useful in scenes with a high structural complexity (high geometry count and/or deep scene graph hierarchies) and low primitive count per geometry. In this kind of scene the *scene graph caching system*, with all optimizations enabled, reduced average frame times by a factor of 5 to 8 with all objects in the scene changing there transformation each frame. This performance gain could be achieved at the cost of startup times increased by 3 to 4 seconds for scenes with 3000 to 8000 *geometry nodes*. The additional main memory consumption was measured 4 MiB for the scene with 3000 geometries and a flat transformation hierarchy and 20 MiB for the scene with 8000 geometries and a deep transformation hierarchy.

Future Work

The *scene graph caching system* that is the result of this work should be considered a first experiment in the direction of incrementally updated scene graph caches. A short time goal for further research is the automatic utilization of *Geometry Instancing* [16, p. 52] by *render caches*. This can be integrated into the proposed system cleanly.

However, a more interesting step would be to investigate the possibility of a system that can deal with any kind of scene graph changes incrementally, including changes to the scene graph structure. Such a system would be similar in function to the one by Hopcroft et al. [31]. A possible way of implementing the required capabilities, while enabling many optimizations at the same time, could be based on an approach similar to the one suggested by Hudson [27]. The resulting system would unify the concepts of *Value Sources* and *Dependent Resources* and allow for clean, incremental graph evaluation.

Before any of this is pursued any further, however, a thorough performance comparison with other modern scene graph toolkits such as *OpenSceneGraph* [8], *OpenSG* [44], and *NVIDIA SceniX* [17] should be conducted, in order to find out whether the *scene graph caching system* can bring absolute performance gains as opposed to only ones relative to AARDVAARK's *immediate mode* renderer.

Bibliography

[1] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Pittsburgh, PA, USA, 2005. AAI3166271.

[2] Umut A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 1–6, New York, NY, USA, 2009. ACM.

[3] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.

[4] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Program. Lang. Syst.*, 32(1):3:1–3:53, November 2009.

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

[6] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 32–42, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[7] Paul E. Black. "inverted index". In Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 14 August 2008. http://www.nist.gov/dads/HTML/invertedIndex.html [Online; accessed 10-July-2012].

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

[9] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, July 1970.

[10] Microsoft Corporation. Accurately Profiling Direct3D API Calls (Direct3D 9), 2012. http://msdn.microsoft.com/en-us/library/bb172234(v=vs.85) [Online; accessed 04-June-2012].

[11] Microsoft Corporation. Primitive Topologies, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/bb205124(v=vs.85).aspx [Online; accessed 24-April-2012].

[12] Microsoft Corporation. Programming Guide for Direct3D 11: Graphics Pipeline (Windows), 2012. http://msdn.microsoft.com/en-us/library/ff476882(v=vs.85) [Online; accessed 04-June-2012].

[13] Microsoft Corporation. Reference for HLSL - Semantics, 2012. http://msdn.microsoft.com/en-us/library/bb509647(v=vs.85) [Online; accessed 02-July-2012].

[14] Microsoft Corporation. Windows Vista and Later Display Driver Model Architecture, 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/ff570589(v=vs.85).aspx [Online; accessed 04-June-2012].

[15] Microsoft Corporation. Windows Vista and Later Display Driver Model Operation Flow, 2012. http://msdn.microsoft.com/en-us/library/windows/hardware/ff570591(v=vs.85).aspx [Online; accessed 04-June-2012].

[16] NVIDIA Corporation. GPU Programming Guide, GeForce 8 and 9 Series, 2008. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf [Online; accessed 31-November-2012].

[17] NVIDIA Corporation. NVIDIA SceniX 7.1 Documentation, 2011. http://developer.nvidia.com/scenix-details [Online; accessed 12-November-2011].

[18] 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.

[19] M. Anton Ertl and David Gregg. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism*, 5:2003, 2003.

[20] Peter Forrest. The Identity of Indiscernibles, 2010. plato.stanford.edu/entries/identity-indiscernible/ [Online; accessed 20-June-2012] in Edward Zalta. Stanford Encyclopedia of Philosophy, plato.stanford.edu.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[22] Fabian Giesen. A trip through the Graphics Pipeline 2011: Index, July 2012. http://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/ [Online; accessed 31-November-2012].

[23] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 25–37, New York, NY, USA, 2009. ACM.

[24] Robert Harper. Practical Foundations for Programming Languages, 2012. Version 1.32 of 05.15.2012. www.cs.cmu.edu/~rwh/plbook/book.pdf [Online; accessed 21-June-2012].

[25] Henry Sowizral, Michael F. Deering, Kevin Rushforth, and Doug Twilleager. Method and apparatus for rapid processing of scene-based programs, 05 2003. US Patent 6570564 B1.

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

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

[28] Chris Lattern and Vikram Adve. LLVM Language Reference Manual, 2012. llvm.org/docs/LangRef.html [Online; accessed 08-June-2012].

[29] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM.

[30] Bertrand Meyer. *Object-Oriented Software Construction (2nd Edition)*. Prentice Hall, 2000.

[31] Michael Hopcroft and Brian Cabral. Method and apparatus for maintaining multiple representations of a same scene in computer generated graphics, 11 2000. US Patent 6154215.

[32] Diego Nehab, Joshua Barczak, and Pedro V. Sander. Triangle order optimization for graphics hardware computation culling. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 207–211, New York, NY, USA, 2006. ACM.

[33] OpenSceneGraph Documentation. osgUtil::Optimizer Class Reference.

[34] Paul S. Strauss. System and method for optimizing a scene graph for optimizing rendering performance, 04 1999. US Patent 5896139.

[35] Pierre S. Boudier. System, method, and computer program product for optimization of a scene graph, 02 2006. US Patent 6995765 B2.

[36] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 322–332, New York, NY, USA, 1995. ACM.

[37] 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.

[38] Dirk Reiners. FeaturesPerformance - OpenSG. http://www.opensg.org/wiki/FeaturesPerformance [Online; accessed 17-May-2012].

[39] Robert Osfield. Vrlab lectures: Performance features of the openscenegraph, 2002. http://www.openscenegraph.org/documentation/VRLabLecture/Motivations/Performance.html [Online; accessed 11-July-2012].

[40] 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.

[41] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.*, 26(3), July 2007.

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

[43] Petr Vanček and Ivana Kolingerová. Technical section: Comparison of triangle strips algorithms. *Comput. Graph.*, 31(1):100–118, January 2007.

[44] 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.

[45] Eric W. Weisstein. "Menger Sponge.". From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/MengerSponge.html [Online; accessed 15-November-2012].

Acknowledgements

I want to thank *Robert F. Tobler*, *Harald Steinlechner*, *Stefan Maierhofer*, and *Georg Haaser* from the *VRVis Research Center* for giving me the opportunity to write this thesis and their support in doing so. Thank you for being a great bunch of people that one can always bounce ideas off of.

I want to thank my parents, *Wilma* and *Erwin Wörister*, for my always being able to rely on them. Thank you for enabling my going to university, for supporting me financially as well as morally during the whole of my education and for buying that *Super Nintendo* back in the Nineties.

Finally, I want to thank my soon-to-be wife, *Gianna Zocco*, for simply changing my life for the better in every way for nearly ten years now. I am looking forward to the next ten years of talking to you.