Lazy Incremental Computation for Efficient Scene Graph Rendering

(draft)

Michael Wörister, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler
VRVis Research Center, Vienna, Austria*

Abstract

In order to provide a highly performant rendering system while maintaining a scene graph structure with a high level of abstraction, we introduce improved rendering caches, that can be updated incrementally without any scene graph traversal. The basis of this novel system is the use of a *dependency graph*, that can be synthesized from the scene graph and links all sources of changes to the affected parts of rendering caches. By using and extending concepts from *incremental computation* we minimize the computational overhead for performing the necessary updates due to changes in any inputs. This makes it possible to provide a high-level semantic scene graph, while retaining the opportunity to apply a number of known optimizations to the rendering caches even for dynamic scenes. Our evaluation shows that the resulting rendering system is highly competitive and provides good rendering performance for scenes ranging from completely static geometry all the way to completely dynamic geometry.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems

Keywords: rendering, optimisation, scene-graph

1 Introduction

Designing a scene representation for a rendering system poses the challenge of choosing the appropriate trade-off between providing a high level of abstraction and achieving high rendering performance. Thus in the scene graph systems with the highest rendering performance, the abstraction level is limited, and the construction of scene graphs is heavily influenced by performance considerations [Rohlf and Helman 1994; Reiners et al. 2002; Burns and Osfield 2004]. This severely increases the complexity of scene graph design for complex, dynamic scenes. On the other hand scene graph systems that provide a high level of abstraction [Tobler 2011] are limited in their rendering performance, due to a semantic structure that cannot be easily optimized.

Some mitigation of this problem is provided by the use of so-called *render caches*. Typically these render caches consist of stores associated with nodes in the scene graph, that contain all rendering calls and associated arguments that are caused by the subgraph beneath the respective scene graph node. Thus the traversal of the subgraph can be omitted by just executing the rendering calls in the cache. Normally render caches are built for only those portions of the scene graph that do not dynamically change from frame to frame. If changes are detected, each of the affected caches must be completely rebuilt.

*email:{woerister,hs,sm,rft}@vrvis.at

In order to improve the performance of such dynamic scenes, and to provide a strong separation between a scene graph system that operates on a high level of abstraction and a performant render backend, we introduce links from all sources of changes to the scene graph (e.g. user events) to each of the affected parts of each rendering cache that needs to be updated. These links are stored in a so-called *dependency graph* that completely eliminates the traversal cost of the scene graph, as long as the structure of the scene graph does not change. Here we use concepts introduced by *incremental computation* in order to optimally perform these tasks: given some input data x (in our case the scene graph) and the computation result $f(x)$ (in our case the rendering caches created from the scene graph), find the necessary changes of $f(x)$ given some changes in x [Ramalingam and Reps 1993].

By allowing dynamic updates of parts of rendering caches, we avoid costly rebuild operations of render caches in dynamic scenes. Additionally, due to the structure of our rendering caches that store instructions to setup the rendering pipeline as well as commands for emitting draw calls, all the typical rendering optimizations such as state sorting, removal of redundant instructions, and super-instructions can be performed and combined arbitrarily on our render caches without affecting the structure of the scene graph, thereby completely separating performance optimizations from the high level design of the scene graph.

In summary we introduce the following contributions:

- An update propagation mechanism that operates on a dependency graph which is synthesized from the scene graph. We utilize a modified version of Hudson's update propagation mechanism (see next section) to keep values consistent in an on-demand manner. We introduce a lazy marking scheme that is adapted to the needs of typical rendering applications by avoiding the eager marking and evaluation of visibility-culled scene parts in order to increase updating performance.
- We introduce rendering caches which are sub-structured so that parts of the caches can be updated incrementally and executed without the need to traverse scene graphs.
- And finally our system introduces a clean separation of scene graph structure and rendering caches that allows to apply a number of known optimizations on the rendering caches without affecting the scene graph even for dynamic content.

2 State of the art

Many existing scene graph systems make use of render call caching in some form or other. The most common approach is the employment of *OpenGL Display Lists*. *Open Inventor* [Wernecke 1993], *OpenSceneGraph* [Burns and Osfield 2004], *Java3D* [Sowizral et al. 1997], and *NVIDIA SceniX* [NVIDIA Corporation 2013] all provide support for building *display lists* that capture the rendering instructions of scene graph nodes. This can provide tremendous performance gains, especially when used in conjunction with *OpenGL*'s immediate mode.

However, once such a display list is built, it cannot be modified. As a consequence, whenever the underlying scene graph changes, all affected display list caches must be destroyed and rebuilt from

scratch, canceling out any positive caching effects. Therefore scenes containing a high percentage of dynamic content derive no benefit from this caching approach.[1]

Microsoft's *Windows Presentation Foundation (WPF)* [Microsoft 2013]—a retained-mode UI system—uses a similar approach. Internally it keeps the so-called *visual tree*, which is a kind of scene graph where each node represents a visual element and edges signify parent-child relationships between those elements. Each node caches a *vector graphics instruction list*, which is used to draw the tree in a depth-first left-to-right traversal. The system's documentation gives no indication that the graphics instructions of a visual node can be updated incrementally: A change to the node causes the whole instruction cache of the node to be re-created. The approach taken by *WPF* is therefore very similar to the display list strategy taken by *Open Inventor* for example.

A more sophisticated, hierarchical caching system is proposed by Durbin et al. [Durbin et al. 1995] As opposed to relying on the *OpenGL* driver implementation of *display lists*, rendering calls are stored explicitly in main memory. The system builds caches in three phases. First, for each scene graph node an instruction array with GPU commands and their precomputed arguments (e.g. accumulated transformation matrices) is created. In the second phase, redundant instructions—setting graphics pipeline state that is already set—are purged from the array. In the third and final phase, the instruction arrays of child nodes are combined at parent nodes so that each node contains a cache representing the whole subgraph below. This way large parts of a scene graph can be rendered without traversing it and without performing unnecessary work.

In order to support dynamic content, the caching system at runtime collects statistical data on the changes done to the scene graph and using this data tries to estimate where the cost of creating caches will amortize over time. Yet again, when the scene graph changes, affected caches need to be destroyed and rebuilt.

None of the above systems allows for truly incremental cache updates such as we propose in this paper. This makes these systems very sensitive to any kind of change in the scene data. In order to find a solution without this limitation, we looked at the field of *incremental computation*, which has been investigated for a long time [Ramalingam and Reps 1993]. One of the first areas where incremental computation was thoroughly examined were language-based editors that need to continuously analyze the edited source code, without disrupting the programmers work flow by performing the analysis from scratch even after only small changes.

There has also been a recent surge of interest in incremental computation (e.g. [Hammer et al. 2009; Acar et al. 2010; Burckhardt et al. 2011; Chen et al. 2012]) following Acar's work on *self-adjusting computation* [Acar 2005], which strives to provide a general purpose method for making arbitrary programs incremental. Yet, due to the similarity between abstract syntax trees in programming languages and scene graphs in computer graphics, it seems more appropriate to us to use the earlier, specialized concepts for our purposes.

Part of the semantic of a programming language can be described by so-called *attribute grammars* [Knuth, Donald E. 1968], which—in addition to the language's syntax given as a context-free grammar—also define how certain attributes (e.g. the set of variable bindings) can be calculated from parent or child nodes in the abstract syntax tree of the program. This is very similar to how attributes, like material properties or spatial transformation, are propagated in a scene graph.

The "standard optimal algorithm" [Hudson 1991, p. 2] for incremental attribute evaluation in trees was developed by Reps et al. [Reps et al. 1983]. The algorithm builds a *dependency graph* that denotes which attribute instances in the tree depend on which other attributes. Upon modification to the tree or attribute values it is then able to propagate resulting changes throughout the data structure in optimal time, i.e. it performs only $\mathcal{O}(|\mathrm{AFFECTED}|)$ steps, where AFFECTED is the set of changed attribute instances.

In the context of scene graph rendering, a modified version of this algorithm could be used to incrementally update those parts of existing render caches which are affected by a change. However, since in a graphical setting only a portion of the scene is visible, not all render caches need to be up-to-date if they will not be used for rendering. As the algorithm by Reps et al. will always propagate changes throughout the whole graph, regardless of visibility, it is likely to perform unnecessary work.

An algorithm better suited to this specific constraint of visibility is presented by Hudson. [Hudson 1991] This algorithm performs attribute updates (mostly) lazily, starting evaluation at attribute instances for which a demand exists. Again, the relationship between attribute instances is represented by a dependency graph, stating which instances need which other instances to compute their value. The change propagation process is split into two phases:

Marking. The first phase marks nodes in the dependency graph as *out-of-date*, starting at known points of change and following the edges of the graph. This phase is performed in an *eager* manner. However, it is rather light-weight and the marking traversal can stop when encountering a node that is already *out-of-date*. At the end, all attribute instances in the system will correctly have their *out-of-date* flag set.

Evaluation. Starting at attribute instances for which a demand exists (because they are "visible") the algorithm recursively updates attribute values. If an attribute is up-to-date, no work has to be done. Otherwise, all input attributes which are out-of-date are updated (this is the recursive step) and then the attribute value itself can be computed. If the attribute value computation contains conditional branches, the algorithm will only traverse parts of the graph that are needed: First, the part representing the value(s) needed for checking the condition, second the branch indicated by the outcome of the condition. The other branch remains untouched. At the end of this phase, all attribute instances in demand are up-to-date, which is now indicated by their correctly set out-of-date flags.

This surprisingly simple algorithm provides good performance and—in a slightly modified version—is what we chose to use for our incremental caching system.

Incremental methods have already been applied in the computer graphics field, especially where expensive computations (such as lighting) meet with interactive edit-review cycles. Examples are the technique by Laine et al. [Laine et al. 2007]—which incrementally maintains a set of *virtual point lights* for radiosity computations—or the Lightspeed system [Ragan-Kelley et al. 2007]. However, these systems use incremental algorithms specialized for their specific computation.[2] Lightspeed separates static and dynamic sub-

[1] Note that a combination of display lists with uniform buffers can provide a simple mechanism for achieving dynamism. In theory, it would be possible to use display lists as a limited replacement to our *instruction arrays*. Display lists, however, do not provide any way of providing the updated data for any of the referenced uniform buffers, which makes recreating them from scratch the only option when no dependency data about their argument buffers is available.

[2] *Lightspeed* uses a *computation graph* to manage the complex depen-

computations. Static computations generate image space caches, which are used by dynamic parts running on graphics hardware. Sitthi-amorn et al. [Sitthi-amorn et al. 2008] exploit temporal coherence in scenes and optimize shaders automatically by utilizing a reprojection cache storing the results of previous frames.

Since modeling applications naturally have to deal with scenes organized in deep hierarchies, it is not surprising that the modeling software Maya [Autodesk 2013] utilizes a dependency graph for attribute evaluation. Maya uses eager marking and lazy evaluation similar to Hudson's algorithm [Hudson 1991], that propagates invalidation information to all parts of the scene irrespective of visibility. To the best of our knowledge, Maya does not employ render caches with in-place updates and non-local instruction array optimization.

Most incremental techniques in computer graphics [Laine et al. 2007; Ragan-Kelley et al. 2007; Sitthi-amorn et al. 2008] use algorithms very specialized to their task. In contrast to this, we try to use the more extensible basis of dependency graphs. While a specialized algorithm exploiting problem-specific properties will always be faster than solutions with less information available, the concept of data and computation dependencies is easily understood and often goes a long way towards practical incrementalization of a task at hand. This paper shows how a dependency graph can be derived from a given scene graph and how the dependency information can be used to build better render caches than is possible without it.

3 Lazy incremental computation for scene graphs

In a scene graph based rendering system repeated traversal of the same node hierarchy incurs an unnecessary overhead if large parts of that hierarchy do not change from frame to frame. In order to avoid this inefficiency, many scene graph based rendering systems use caches to store the sequence of rendering calls generated by a traversal (or some other equally compact representation). These caches can be associated with nodes in the scene graph and obviate the need to traverse the sub-nodes of the respective node (for an example see Figure 1.

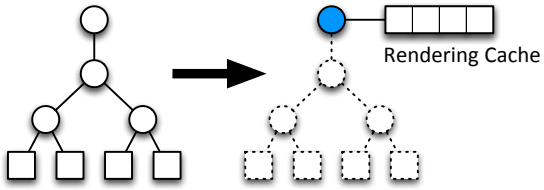


Figure 1: *A rendering cache captures all rendering instructions in a subgraph and takes over its rendering responsibilities.*

For static scenes it may be possible to use a single rendering cache for the complete scene, however even in this case it is normally beneficial to split the rendering cache into various partial caches, since due to visibility culling only part of the scene may be rendered at any one time (see Figure 2). Rendering caches are normally built in such a way that they can only be rendered completely or not at all, no culling of parts of a cache is usually possible.

dencies within its multi-pass rendering work flow, however, this graph is not itself used as a means to incrementality. The incremental aspects of the system stem from (1) the partial evaluation done in the preprocessing step and (2) the specialized *light caching* exploiting the linearity of lighting calculations.

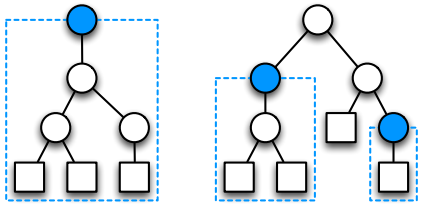


Figure 2: *Rendering caches (blue nodes) can be built for a complete scene graph or for parts of a scene graph.*

As long as the scene does not change, these rendering caches do not need to be modified, but as soon as any change in a dynamic scene takes place, a number of nodes in the scene graph are affected, and subsequently some of the rendering caches need to be updated. Figure 3 shows an example scene graph with two modified attributes a and b (with the sub-graphs affected by these attributes indicated in red), that require a subsequent update of the two rendering caches in the graph.

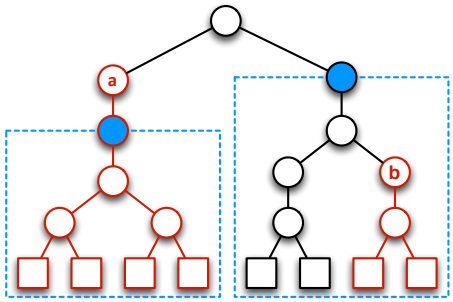


Figure 3: *Modifying some attributes (*a *and* b*) requires the update of rendering caches (blue nodes) that capture on them (affected scene graph parts are shown in red).*

Depending on the nature of the changes in the scene graph, we need to distinguish between two types of updates of rendering caches:

In-place updates. If a scene graph modification only changes the value of a rendering cache without changing its structure, no allocation of buffers and binding of resources has to occur, making this kind of update fairly inexpensive (see example in Figure 4).

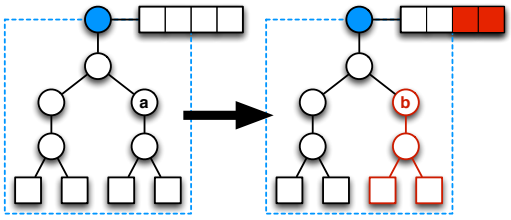


Figure 4: *An in-place update of a rendering cache due to a modification of an attribute (affected scene graph parts and rendering cache parts shown in red).*

Structural updates. If a scene graph modification gives rise to some changes in the structure of a rendering cache, new buffers have to be allocated and resources for the graphics hardware have to be bound (see example in Figure 5). In the current implementation of our system these updates are not handled automatically.

A standard scene graph traversal is used for building all affected render caches anew.

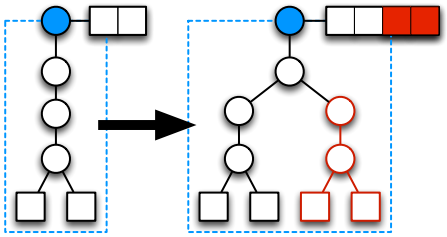


Figure 5: *A structural update of a rendering cache due to the insertion of a new sub-graph (affected scene graph parts and rendering cache parts shown in red).*

In-place updates can be implemented via simple call-backs that are stored in the rendering caches. In order to efficiently compute which parts of rendering caches need to be updated, it is necessary to introduce the concept of a *dependency system* that maintains meta data for each entity in the scene graph and describes on which other parts within or outside the scene graph each entity depends. The meta data in our dependency system introduces the following roles:

Dependencies. A dependency is a stateful predicate that 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 of the observed object 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. The ability to efficiently determine if an object described by such a dependency needs to be updated is key for implementing an efficient dependency system. Changed dependencies serve as triggers that cause the execution of the call-back functions that are used to update all rendering caches.

Value Sources. A value source is an item in the scene graph or global environment that is needed when computing the value of an object that is dependent on it. 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 of the scene graph that holds geometry. A value source holds a set of dependencies that 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. Value Sources serve as parameters of the update call-back functions.

Dependent Resources A dependent resource is an object whose value changes whenever one of its dependencies changes. It uses its value sources to compute its new value. In a rendering system the buffers and parameters that are uploaded to the graphics hardware are typically dependent resources. The update call-back functions compute new values for dependent resources.

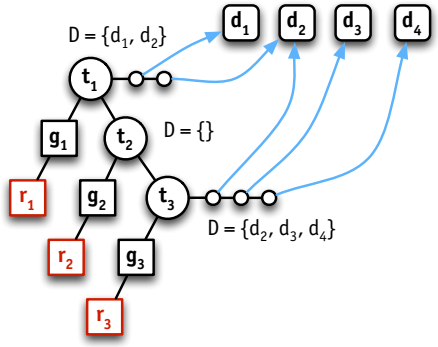


Figure 6: *A small scene graph with transformation nodes as value sources (t_i), a number of dependencies for these value sources (d_j), a number of geometries (g_k), and a number of dependent resources (r_l).*

As an example (see Figure 6), consider three geometries ($\mathbf{g}_1$, $\mathbf{g}_2$, and $\mathbf{g}_3$) in a scene graph, each of which is transformed by a transformation node ($\mathbf{t}_1$, $\mathbf{t}_2$, and $\mathbf{t}_3$). The transformations in turn are dependent on a number of external events ($\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{d}_3$, and $\mathbf{d}_4$) and are arranged in a stacked fashion. Each of the three geometries gives rise to a dependent resource, dependent on their model-transformation which is loaded onto the graphics hardware for rendering ($\mathbf{r}_1$, $\mathbf{r}_2$, and $\mathbf{r}_3$).

The dependencies depicted in Figure 6 constitute all changes that trigger modifications to the transformation nodes. These can be caused by user inputs or by simulation results external to the scene graph (e.g. a particle simulation computes new transformations for moving particles).

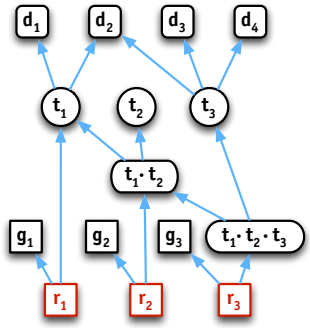


Figure 7: *The dependency graph constructed from the scene graph in Figure 6.*

In order to explicitly represent all the necessary changes that need to be computed, we define the so-called *dependency graph*: By introducing intermediate value sources (in the example the two nodes $\mathbf{t}_1 \cdot \mathbf{t}_2$ and $\mathbf{t}_1 \cdot \mathbf{t}_2 \cdot \mathbf{t}_3$) each with their own dependencies (in this case other value sources provide the dependencies for the newly created value sources), a complete chain of dependencies for each rendering resource is encoded in the dependency graph. Figure 7 shows the dependency graph that has been built from the simple example in Figure 6.

Note that the implied dependency in the scene graph—that leaf nodes depend on all attributes above them—can be exploited to avoid the complete and explicit computation of the dependency graph. However, a more explicit representation of the graph can be used to increase update performance: As an example, the explicit

creation of the multiplication nodes (in the example the two nodes $\mathbf{t}_1 \cdot \mathbf{t}_2$ and $\mathbf{t}_1 \cdot \mathbf{t}_2 \cdot \mathbf{t}_3$) with stored multiplication results can be avoided at the cost of doing the additional work of re-performing the matrix multiplication $\mathbf{t}_1 \cdot \mathbf{t}_2$ if only dependencies $\mathbf{d}_3$ or $\mathbf{d}_4$ change.

The rendering caches that have been introduced into the scene graph now consist of the rendering instructions for drawing the cached subgraph, and the dependent resources that serve as arguments to these rendering instructions. Rendering the contents of a rendering cache involves two steps:

1. Make the arguments consistent with the scene graph they mirror. This is done by triggering their update callback functions (if necessary).
2. Execute the rendering commands in the order they are stored in the cache.

As mentioned in the *State of the Art* section, we chose to use a modified version of Hudson's algorithm [Hudson 1991] to perform change propagation, that is step (1) of the above two steps. Our version of the algorithm performs the same two tasks of (a) finding the set of dependent resources that need to be updated and (b) subsequently actually performing the update callbacks of these resources.

However, in contrast to Hudson's algorithm, we want to take visibility information into account for out-of-date checking too: only for rendering caches that have survived visibility culling the out-of-date check should be performed. Therefore we replace Hudson's eager out-of-date marking with on-demand out-of-date polling. To implement this, every updateable node in the dependency graph stores the set of all transitively reachable dependency predicates and the reference version number for each dependency. An out-of-date check for a given dependent resource can then be performed by polling the version number of each of its dependencies and comparing it with the reference version number.

The polling of unculled nodes is additionally sped up by building an *inverted index* data structure [Knuth, Donald E. 1998, pp. 560-563] per render cache, mapping dependency objects to the list of their dependent resources. This *dependency index*, as we call it, allows to touch unchanged dependencies only once per render cache. Changed dependencies still have to be read once per dependent resource for keeping reference version numbers consistent.

Note that some nodes in the dependency graph are already needed to perform culling, namely bounding boxes and everything that is needed to compute them. For these resources the out-of-date check cannot be eliminated as the culling system will always place demand on them. However, the concrete implementation of dependency predicates can be used to filter out changes insignificant to bounding box updates in order to at least avoid unnecessarily recomputing bounding box and transformation values.

4 Integrating incremental caching into scene graphs

In our prototypical implementation of the incremental caching system we introduce *cache nodes* which mark the subgraph under them for caching. This node has the responsibility of creating the instruction array and the dependent resources that are the arguments of these instructions. It also has the responsibility of keeping these dependent resources up-to-date, as their value sources change. To this end, it also stores the dependency index mentioned in the previous section.

Building a rendering cache is very similar to performing a regular rendering traversal. When a cache node is reached by a rendering traversal, and no cache has been built yet, it will send a so-called `ExtractCachingDataTraversal` into the subgraph below. This special traversal behaves very much like the regular rendering traversal but instead of calling into the graphics hardware API, it records these calls as instructions, adding them to the instruction array of the cache. For every instruction argument, it allocates a dependent resource. The information needed for creating a dependent resource is: (1) the semantics of the resource (e.g. `model-transformation`), and (2) the value sources of the resource (e.g. the transformation nodes affecting the transformation at the leaf node). From this information the dependencies of the dependent resource (the union of the dependencies of its value sources) and the update callback can be inferred. The dependent resource is then added to the dependency index. When the traversal is finished, instruction array and dependency index are completely populated.

Now that the cache is built, it can take over the responsibility of rendering the graph below the cache node. From now on, when a rendering traversal reaches the cache node, it will execute the render cache instead of descending into the sub-graph. As described before, this consists of first updating the instruction arguments using the dependency index, and then executing the instruction array.

5 Rendering Cache Optimizations

With the introduction of rendering caches, the high-level data model of the scene graph is cleanly separated from the rendering specific data structures in these caches. Thus it is possible to perform a number of optimizations without affecting the high-level data model. We perform the following optimizations on the rendering caches for maximizing rendering performance:

Removal of Redundant Instructions. Often the correct GPU program, texture, or other graphics API state is already set to the correct value before a certain geometry is rendered. By analyzing the instruction stream it is therefore possible to remove instructions that set state to a value that it is known to already have.

This optimization is often already performed by the graphics driver. But the driver can only filter out GPU instructions after the call to the graphics API has been made while we can do so before any API interaction. The optimization is also done by existing scene graph toolkits, such as *OpenSceneGraph* and *IRIS Performer* [Rohlf and Helman 1994, Section 2.2.3]. However, these systems perform the filtering every frame before drawing, while with rendering caches it only has to be done once when the instruction array is built.

Superinstructions. A common optimization in byte-code interpreters are so-called *superoperators* [Proebsting 1995] or *superinstructions* [Ertl and Gregg 2003, p. 20]. These work by folding common sequences of primitive instructions into larger, semantically equivalent *superinstructions*. The same can be done for rendering instructions stored in main memory. The benefit is reduced instruction dispatch overhead. In the case of our implementation this means a smaller number of virtual function calls and type-casts.

State Sorting. Graphics hardware can work most efficiently when it is able to process large batches of data without switching internal state such as GPU programs, render target, or textures. It therefore is beneficial to sort the render jobs for opaque geometry in order to reduce these kinds of state changes before building the instruction stream from them. The sorting algorithm uses a cost model that assigns larger penalties for costly state changes (such as switching the render target) and smaller ones to cheaper ones (such

as switching vertex buffers).

This is a standard optimization performed by most scene graph toolkits. However, the use of render caches allows to perform the optimization without modifying the scene graph, and the optimization only has to be performed once when the cache is built (as opposed to every frame, as done by *OpenSceneGraph* and *IRIS Performer* [Rohlf and Helman 1994, Section 3.1.3] for example). Note, that this optimization cannot be performed at driver level since draw calls cannot be reordered in general. We apply state sorting only if admissible i.e. no transparent geometry is rendered.

Transformation Matrix Memoization. Normally in a scene graph, transformation matrices are accumulated along the path until a leaf node is reached. Like Durbin et al. [Durbin et al. 1995] we achieve a speedup by caching the once computed end result together with the instruction that uses it (the dependent resources in our system). However, when a transformation node changes its value, the cached matrices for all paths that contain that node are invalidated. To alleviate the burden of updating the cached matrices, we also optionally cache all intermediate results of the matrix accumulation process. With these intermediate results available, only values *after* a changed transformation node have to be updated and leafs sharing path prefixes can share the cached results for these prefixes.

Parallel Updates of Dependent Resources. Since the dependency system defines a clean computational model for updating dependent resources, it is possible to devise a strategy for safely performing these resource updates in parallel on multiple CPU cores. For scenes with a high percentage of changing resources (such as transformations in uniform buffers) a parallel update strategy achieves significant performance gains.

Generalized Draw Sorting. With dependencies rendering caches can refer to arbitrary attributes of the scene like model transformations and bounding boxes. This for example allows us to automatically sort draw calls relative to the current viewport. Without overlapping geometry this feature can be used to implement correct back to front rendering. Alternatively, performance in opaque scenes can be improved with front to back rendering exploiting early z optimization.

6 Implementation

Our implementation of the scenegraph as well as the rendering backend is written in C#. We use SlimDX[3] which exposes DirectX as a managed class library, usable from C# directly. As graphics API we use Direct3D 11.
As described in the previous section render caches contain arrays of rendering instructions. These virtually reflect D3D device methods, but at slightly higher abstraction level which allows to target other graphics APIs such as OpenGL. Examples for render commands are:

- `SetVertexShader(IShader shader)` uses a shader as input and binds it to the rendering pipeline when executed.
- `SetConstantBuffers(Dictionary<int, IConstantBuffer> constantBuffers, ShaderType shaderType)` assigns a set of constant buffers to the shader stage specified by `shaderType`.
- `SetDepthStencilState(IDepthStencilState state)` binds the specified state to the rendering pipeline.

[3] `http://slimdx.org/`, accessed on Feb. 15th, 2013

Whenever possible instruction arguments are directly allocated as GPU resources, like buffers (e.g. *constant buffers*, *vertex buffers*). Note that GPU resources indeed match our notion of dependent resources and therefore integrate with the dependency system directly.
In order to execute a render cache we could simply execute each instruction sequentially by calling its `Execute()` method. Our improved implementation however creates so-called *native instructions* of these render commands. Native instructions are specialized versions of the original render commands with their arguments already prepared for the specific graphics API. This further reduces execution overhead because all arguments can be converted to the appropriate type at optimization time instead of execution time.

7 Results

In order to assess the impact of the use of dependencies on rendering performance we focussed on benchmarks with a high number of draw calls. Typical optimizations like occlusion culling, view frustum culling and pre-packing and transforming geometry are all geared towards reducing the number of draw calls, however in the most highly demanding scenes these optimizations alone are sometimes not sufficient to achieve acceptable rendering speed. Thus performing the measurements in benchmarks with a large number of draw calls highlights the performance in worst-case scenarios. Based on the same considerations we also switched off visibility culling, ensuring that each of the geometries in our benchmark scene causes a draw call that needs to be performed.

For all our results we used an Intel(R) Core(TM) i7-3770 @ 3.40GHz (4 cores with Hyper-Threading), 32GB RAM, 64bit Windows 7 and an NVIDIA GeForce GTX 680 graphics card with 2048MB memory. OpenSceneGraph [Burns and Osfield 2004] binaries are compiled with highest compiler optimization `-O2`. Our system is implemented in C# and runs on top of `.NET 4.0`. In order to warm up the system we discard the first iteration of each test run.

7.1 Static Scenes

First, we investigated the performance characteristics of the incremental caching system with a scene that does not change over time, but causes a high load of the graphics system. As rendering caches mostly minimize scene graph traversal overhead, we tried to make this the varying factor in our test setup while other factors are fixed. To achieve this goal, we created a configurable synthetic test scene containing different numbers of spheres, each with a different number of triangles. To keep the GPU workload roughly constant, all configurations contain approximately 1.6 million triangles. These triangles are distributed over a varying number of geometry nodes which leads to different amounts of traversal overhead. Eight different surface configurations are distributed evenly over the spheres, each using a diffuse texture map, a normal map, and a shared environment map. There is no visibility culling involved in this test setup: everything is considered visible in every frame (see Figure 8).

The results of the test runs with different geometry counts (but constant triangle count), comparing our traversing renderer with our caching renderer are shown in Figure 9. Note that each geometry causes a single draw call. As can be seen, the time needed to render a frame (*frame time*) increases linearly with rising geometry count, with and without caching enabled. However, as can also be observed, render caches improve performance consistently by a factor of about 2.6 in this test setup. This suggests that traversal indeed is the performance bottleneck of the traversing renderer since the sequence of graphics API interactions is nearly the same in both

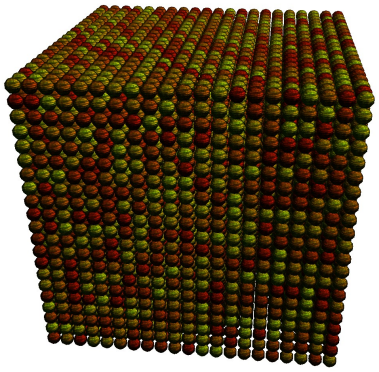


Figure 8: *Screenshot of our worst case test setup designed to cause a high load on the graphics pipeline: the camera slowly rotates around this grid of spheres which thus remains inside the view frustum at all times.*

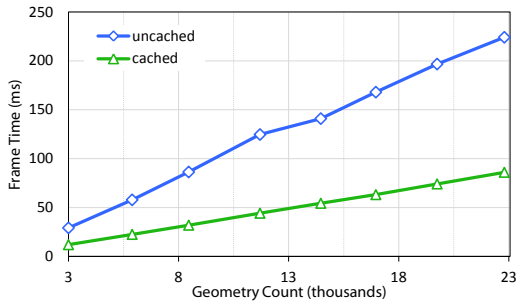
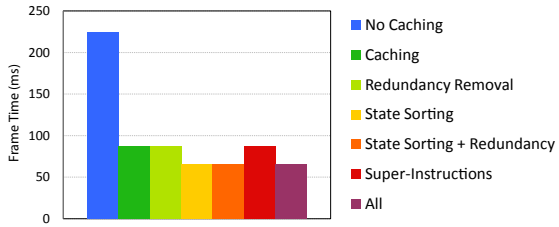


Figure 9: *Comparison of frame-times for our reference implementation with the proposed caching mechanism in our static scene. While the total scene complexity (number of primitives) is constant the number of geometries (equivalent to draw calls) varies to show the overhead due to the traversal of the scene graphs.*

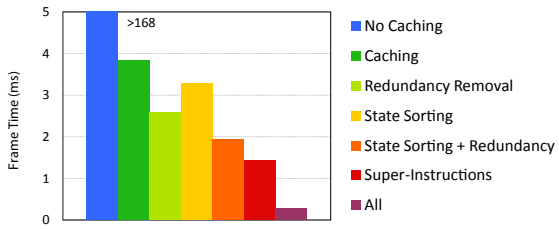
cases. The large difference is due to our use of a semantic scene graph as introduced by Tobler [Tobler 2011]. The high flexibility and clean semantic model of this approach is normally hampered by a significant traversal cost. The use of rendering caches completely eliminates this deficiency.

As mentioned before, rendering caches allow for a number of optimizations without modifying the underlying scene graph structure. Figure 10a shows the effects of enabling different combinations of optimizations in the test scene at 22736 geometries. Here it can be observed that only configurations with the *state sorting* optimization enabled yield additional performance gains while the frame time does not change for the other configurations. This can be explained under the assumption that unoptimized caching already makes the application entirely GPU-bounded. The optimizations that solely affect the CPU-side (*removing redundant instructions* and *super instructions*) cannot achieve any visible improvement (the CPU is already waiting on the GPU, the optimizations just make it wait a greater percentage of the time). The state sorting optimization allows the GPU to process the data more efficiently and thus has a visible effect on frame time.

Figure 10b shows that the super instruction and redundancy removal optimizations actually do have an effect on the CPU workload. These tests were run with the same settings but without actually calling into the graphics API, in order to completely eliminate any interaction with either the GPU or the graphics driver. Here



(a) *Frame times with rendering enabled.*



(b) *CPU only frame times with rendering disabled, i.e. no DirectX calls.*

Figure 10: *The benefit of different optimizations applied to a static rendering cache containing 22K objects with randomly assigned material properties. Note that some optimizations gain no additional speedup (10a) in our test scene although CPU overhead is reduced significantly (10b).*

the traversing renderer needs around 168 milliseconds to traverse the scene graph. In contrast, instruction array execution takes under 4 milliseconds. The other optimizations can reduce this further, however as mentioned before, the difference does not become visible when the GPU is enabled, but does free up CPU resources for additional application-specific processing.

To summarize, without caching the application is traversal and therefore CPU-bounded, while it becomes GPU-bounded when caching is enabled. As the caching system (apart from the state sorting optimization) only affects the CPU-side of the rendering process, this already poses an important goal of the caching system: Even for a flexible, but inefficient scene graph implementation the performance bottleneck can be moved to the GPU. This was shown for scenes with a high numbers of geometry nodes (in the thousands).

However, as the scene graph traversal of a single node is unlikely to be the performance bottleneck, caching and traversing renderer have to start with similar performance for very low geometry counts. To find out where the scene graph traversal becomes the limiting factor of the execution we have to look at frame times for lower geometry counts. Figure 11 shows frame times for configurations with geometry counts between 10 and 800. Here, it can be seen that caching starts to have a positive effect beginning at around 200 geometries. It must also be noted that the test scene has a very simple scene graph structure with no stacked transformation nodes and no redundant group nodes. For settings with more complex graph structure the turning point is likely to occur even earlier.

Table 1 shows the caching system's cost in additional startup time. Again, the numbers were measured for the largest scene with 22736 geometries. Loading the scene and creating a rendering cache will take 120% of the time of just loading the scene normally. Enabling additional optimizations will incur additional startup time.

The upper bound of the caching system's memory consumption is linear in the number of draw calls the cache captures. As the system does not duplicate static geometry data and only stores additional constant buffers, instructions and dependency information—all of

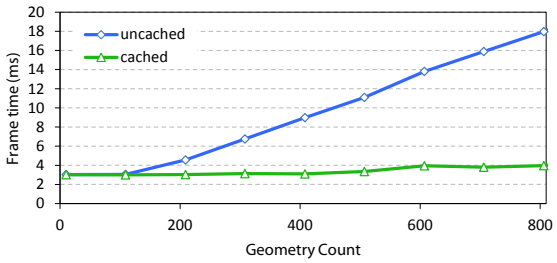


Figure 11: *Close-up of Fig. 9 with low number of draw calls. With more than approx. 200 draw calls, caching pays off due to reduction of CPU overheads.*

Configuration	Additional startup time
Caching	20%
Caching+Redundancy Removal	20%
Caching+State Sorting	25%
Caching + Redundancy Removal, State sorting, Superinstructions	49%

Table 1: *Time required to optimize rendering caches for 22K geometry nodes*

which are relatively lightweight—the memory overhead will mostly account only for a small fraction of the entire application's memory consumption. In the largest test case shown before, the cache took up 3 MB of main memory and 3 MB of graphics memory, with the whole application using 324 MB main and 669 MB graphics memory.

Our system is implemented in C# and uses a flexible scene graph system similar to Tobler's semantic scene graph [Tobler 2011], resulting in a comparatively high traversal cost. We compared performance of our test scenes in *OpenSceneGraph 3.0.1*, which does not employ our caching system. Figure 12 shows that our cached rendering algorithm (with state sorting enabled) can compete with *OpenSceneGraph*'s frame times (and is even a bit faster). In static test scenes our caching system was able to completely eliminate traversal overhead and CPU-boundedness.

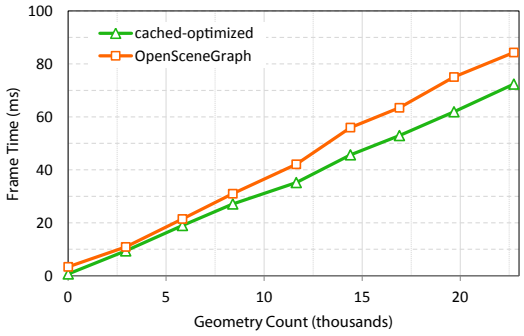


Figure 12: *Frame times for the static scenes with varying number of geometries in our system with render caches (state sorting and redundancy removal enabled) and in OpenSceneGraph (single-threaded).*

7.2 Dynamic Scenes

An important feature of our incremental caching system is its ability to cope with non-structural scene changes. In Figure 13 we measured frame times in our test scene with different percentages of objects changing their transformation matrix each frame. As can be seen, our incremental algorithm compares favorably with *OpenSceneGraph*: in the non-parallel/non-threaded version, our incremental update scheme outperforms *OpenSceneGraph* in all cases, since it eliminates all CPU overhead and makes the benchmark GPU bounded. *OpenSceneGraph*'s multi-threaded[4] execution is needed to compete with our non-parallel execution speed. Parallelization does not benefit our method in this case, since even in the non-parallel case the GPU is already supplied with rendering commands at the speed it can handle them.

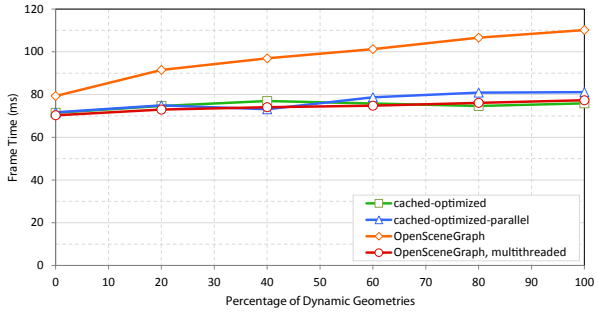


Figure 13: *Frame times in scenes with varying percentage of moving objects. Transformation matrices are updated each frame – these updates require additional CPU processing due to matrix multiplication but also involve the GPU for resource updates.*

For testing the effectiveness of our incremental update algorithm, we built a hierarchical scene with the structure of an octree, where two possibly dynamic transformation nodes are placed at each hierarchy level (see Figure 14). This structure reflects the deep hierarchies that are often present in editing applications that allow interactive changes at every level of a semantic hierarchy.

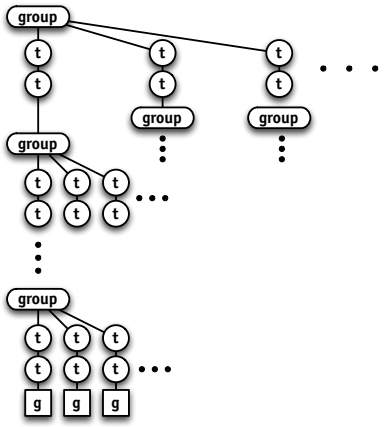


Figure 14: *The scene graph structure of our benchmark for incremental updates.* ***t*** *denotes transformation nodes,* ***g*** *denotes geometry nodes.*

Figure 15 shows the performance of our system with memoiza-

[4]The exact threading model used was *DrawThreadPerContext*, which provided the best performance on our test machine.

tion[5] and parallel updates when compared to OpenSceneGraph for a scene with 22736 geometries and a hierarchy depth of 8 transformations. We vary the number of changed transformation nodes right above the geometries from 0 to 100%. The use of the dependency graph eliminates the traversal cost along each scene graph path to a geometry, resulting in a significant speedup. Even the non-parallel version already slightly out-performs the multi-threaded *OpenSceneGraph* version, and both the version with memoization and the version with parallel updates increase the gap. The comparatively small difference between the static and the fully dynamic case, as well as the minimal speed-up attained by combining parallelization and memoization, indicates that our update system is again fast enough to become nearly entirely GPU bounded. Nevertheless this clearly demonstrates the benefits of incremental computation in reducing the work load in dynamic scenes to the minimal necessary amount.

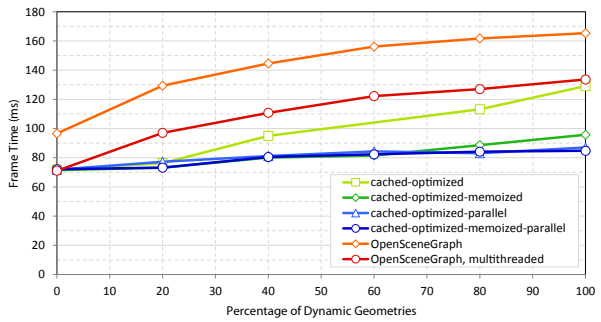


Figure 15: *Frame times in a scene with a depth complexity of 8 transformations. The scene is structured as an octree with additional dynamic transformation nodes at each level, as shown in Figure 14.*

8 Conclusions and Future Work

We have applied the concepts from incremental computation to real-time rendering of scenes based on semantically structured scene graphs in order to efficiently handle dynamic content in a uniform and clean manner. Based on the usual optimization of using rendering caches to speed up scene graph rendering, we constructed the implied dependency graph and introduced an improved version of Hudson's method for computing the necessary updates. By using call-back functions for performing in-place updates of rendering caches, and exploiting the parallelization information inherent in the dependency graph, we demonstrate that our system can eliminate the costs typically associated with a semantic scene graph with a high level of abstraction, and handle fully dynamic scenes with minimal overhead.

In the future we intend to research the necessary extension for dealing with structural updates and thus enable fast editing changes in massive, fully dynamic scene graphs.

References

ACAR, U. A., BLELLOCH, G., LEY-WILD, R., TANGWONGSAN, K., AND TURKOGLU, D. 2010. Traceable data types for self-adjusting computation. In *Proc. of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ACM, New York, NY, USA, PLDI '10, 483–496.

ACAR, U. A. 2005. *Self-adjusting computation*. PhD thesis, Pittsburgh, PA, USA. AAI3166271.

AUTODESK, 2013. Dependency Graph (DG) nodes. `http://docs.autodesk.com/MAYAUL/2014/ENU/Maya-API-Documentation/index.html`. Accessed: 2013-06-01.

BURCKHARDT, S., LEIJEN, D., SADOWSKI, C., YI, J., AND BALL, T. 2011. Two for the price of one: a model for parallel and incremental computation. In *Proc. of the 2011 ACM international conference on Object oriented progr. systems languages and applications*, ACM, New York, NY, USA, OOPSLA '11, 427–444.

BURNS, D., AND OSFIELD, R. 2004. Open scene graph a: Introduction, b: Examples and applications. In *Proc. of the IEEE Virtual Reality 2004*, IEEE Computer Society, Washington, DC, USA, VR '04, 265–.

CHEN, Y., DUNFIELD, J., AND ACAR, U. A. 2012. Type-directed automatic incrementalization. In *Proc. of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, PLDI '12, 299–310.

DURBIN, J., GOSSWEILER, R., AND PAUSCH, R. 1995. Amortizing 3d graphics optimization across multiple frames. In *Proc. of the 8th annual ACM symposium on User interface and software technology*, ACM, New York, NY, USA, UIST '95, 13–19.

ERTL, M. A., AND GREGG, D. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism 5*, 2003.

HAMMER, M. A., ACAR, U. A., AND CHEN, Y. 2009. Ceal: a c-based language for self-adjusting computation. In *Proc. of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ACM, New York, NY, USA, PLDI '09, 25–37.

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

KNUTH, DONALD E. 1968. Semantics of context-free languages. *Mathematical systems theory 2*, 127–145.

KNUTH, DONALD E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

LAINE, S., SARANSAARI, H., KONTKANEN, J., LEHTINEN, J., AND AILA, T. 2007. Incremental Instant Radiosity for Real-Time Indirect Illumination. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'07, 277–286.

MICROSOFT, 2013. WPF Graphics Rendering Overview. http://msdn.microsoft.com/en-us/library/ms748373.aspx [Online; accessed May 31, 2013].

NVIDIA CORPORATION, 2013. SceniX — NVIDIA Developer Zone. developer.nvidia.com/scenix [Online; accessed February 12, 2013].

PROEBSTING, T. A. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of progr. languages*, ACM, New York, NY, USA, POPL '95, 322–332.

[5]The additional memory cost of enabling memoization amounted to about 40 MB in the given test scene.

RAGAN-KELLEY, J., KILPATRICK, C., SMITH, B. W., EPPS, D., GREEN, P., HERY, C., AND DURAND, F. 2007. The Lightspeed Automatic Interactive Lighting Preview System. *ACM Trans. Graph. 26*, 3 (July).

RAMALINGAM, G., AND REPS, T. 1993. A categorized bibliography on incremental computation. In *Proc. of the 20th ACM SIGPLAN-SIGACT symposium on Principles of progr. languages*, ACM, New York, NY, USA, POPL '93, 502–510.

REINERS, D., VOSS, G., AND BEHR, J. 2002. Opensg: Basic concepts. In *1. OpenSG Symposium*.

REPS, T., TEITELBAUM, T., AND DEMERS, A. 1983. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July), 449–477.

ROHLF, J., AND HELMAN, J. 1994. 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*, ACM, New York, NY, USA, SIGGRAPH '94, 381–394.

SITTHI-AMORN, P., LAWRENCE, J., YANG, L., SANDER, P. V., NEHAB, D., AND XI, J. 2008. Automated reprojection-based pixel shader optimization. *ACM Trans. Graph. 27*, 5 (Dec.), 127:1–127:11.

SOWIZRAL, K., RUSHFORTH, K., AND SOWIZRAL, H. 1997. *The Java 3D API Specification*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

TOBLER, R. F. 2011. Separating semantics from rendering: a scene graph based architecture for graphics applications. *Visual Computer 27*, 6-8 (June), 687–695.

WERNECKE, J. 1993. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*, 1st ed. Addison-Wesley Longman Publ. Co., Inc., Boston, MA, USA.