Eurographics Symposium on Rendering 2010
Jason Lawrence and Marc Stamminger
(Guest Editors)

Volume 29 (2010), Number 4

SafeGI: Type Checking to Improve Correctness in Rendering System Implementation

Jiawei Ou Fabio Pellacini

Dartmouth College

Abstract

Historically, rendering system development has been mainly focused on improving the numerical accuracy of the rendering algorithms and their runtime efficiency. In this paper, we propose a method to improve the correctness not of the algorithms themselves, but of their implementation. Specifically, we show that by combining static type checking and generic programming, rendering system and shader development can take advantage of compile-time checking to perform dimensional analysis*, i.e. to enforce the correctness of physical dimensions and units in light transport, and* geometric space analysis*, i.e. to ensure that geometric computations respect the spaces in which points, vectors and normals were defined. We demonstrate our methods by implementing a CPU path tracer and a GPU renderer which previews direct illumination. While we build on prior work to develop our implementations, the main contribution of our work is to show that dimensional analysis and geometric space checking can be successfully integrated into the development of rendering systems and shaders.*

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems

1. Introduction

Implementation of Rendering Systems. Research in rendering systems has been mostly focused on advancing the efficiency of the implemented methods and taking advantage of the latest hardware platforms. To this day though, there is a lack of focus on improving the correctness not of the algorithms themselves, but of their implementation.

Researchers in programming languages have developed several methods to improve the correctness of algorithms' implementation, a review of which is beyond the scope of this paper. These methods vary greatly in the runtime costs associated with enforcing correctness as well as the programmer's effort required to take advantage of these methods. In the specific case of implementing rendering systems or writing shaders, runtime overhead has to remain minimal, since efficiency is still of maximum importance. Furthermore, since large numbers of shaders are required for graphics projects, e.g. feature films or games, any method that increases programmer time substantially would also be impractical.

Static Type Checking. Of the various methods that aid programmers in implementing correct code, static type checking has proven to be remarkably successful to the point where all languages used for implementing rendering systems (e.g. C, C++) and shaders (e.g. RSL [HL90], GLSL [Ros05], HLSL [Bly06], Cg [MGAK03]) are built on such a feature. Static typing improves not only on the correctness of the implementation, but the speed itself, when compared to dynamic language implementations. In the context of rendering algorithms, a common example of the use of type checking to improve correctness is in geometry calculations. By defining different types for points, vectors and normals, and their associated arithmetic operations, we can improve on the correctness of the implemented geometric computations. To the best of our knowledge though, besides this simple example, few other properties of algorithms are normally checked in either shading development or large rendering system development.

1.1. *SafeGI*

Overview. In this paper, we show that by combining static type checking and generic programming, rendering system and shader development can take advantage of compile-time checking to perform *dimensional analysis*, i.e. to enforce the correctness of physical dimensions and units in light transport, and *geometric space analysis*, i.e. to ensure that geo-

metric computations respect the spaces in which points, vectors and normals were defined. We build on prior work in programming languages that shows how to modify the type checker to perform these tests. The contribution of this paper is to show the practicality of this approach for rendering system and shader development as well as a set of best practices we found helpful during development. Our experiment shows that enforcing these additional checks has little to no runtime overhead, since type checking happens at compile time, and little user overhead, since type checking is already familiar to rendering system programmers and shader writers.

***SafeGI*'s CPU Renderer**. We implemented a simple unoptimized rendering system from scratch to demonstrate how a full system would be developed in a *SafeGI* framework, where we use template metaprogramming to perform dimensional and space checking. We chose to focus first on a very small codebase since it allows us to implement the same algorithms multiple times while varying how to model dimensions, units, spaces and APIs. In particular, we used C++ to implement a simple recursive path tracer where materials, lights and rendering APIs were modeled similarly to [PH04]. We developed and compared two different implementations, one with and without dimensional and geometric space analyses.

***SafeGI*'s GPU Renderer**. Within the same codebase, we also implemented a simple GPU renderer to preview direct illumination. For the GPU component, we show how to modify the real-time rendering APIs and shading languages, namely OpenGL [SBW*09] and GLSL [Ros05], such that type safety is maintained for code executed on the GPU and in the communications between CPU and GPU.

***SafeGI* in PBRT**. To test the feasibility of *SafeGI* in large rendering systems, we added *SafeGI* extended type checking to the *PBRT* open source renderer [PH04]. We found that doing so required very little programmer overhead, even if the original system was not designed with additional type checking in mind. Namely it took us 70 man-hours to port a large subset of the full *PBRT* renderer. Furthermore, the runtime overhead was only $3-4\%$, which we believe is acceptable given the additional checking and since the original system was not designed to be type safe in the first place. We chose to port *SafeGI* into *PBRT* to prove that it is feasible to incorporate *SafeGI* into existing production rendering implementations.

Validation. While we found it difficult to quantitatively evaluate the benefits of integrating dimensional analysis and geometry space checking into the implementation of rendering algorithms, we have certainly found that it helped us discover a few bugs even in our very simple setup as well as some minor inconsistencies in *PBRT*. This is quite remarkable given the small size of our simple system and the known robustness of *PBRT*.

Contributions. In the context of developing rendering systems and shaders, this paper makes the following contributions:

- we demonstrate, within the graphics community, how to use static type checking and generic programming to provide compile-time dimensional and geometric space analyses in the implementation of C++ rendering systems; our implementations exhibit little to no runtime overhead and little programmer cost
- we show how to modify real-time rendering APIs and shading languages to support these tests
- we discuss a set of best practices and programming patterns we found useful in modeling fully type-checked global illumination algorithms
- we provide full source code for our renderers to provide readers with concrete examples of how rendering systems can be developed with *SafeGI*

2. Background

Dimensional Analysis. Dimensional analysis [Ken94] is the process by which we can verify that physical quantities are properly combined in arithmetic operations. E.g., one cannot sum a velocity and a mass without violating some rules of physics. Since most rendering algorithms, e.g. global illumination simulations, are fundamentally simulating a physical process, it would certainly be beneficial to check whether any problems arise when checking for dimensional correctness. Furthermore, one could also seek to verify that units are properly used; when summing lengths for example, one needs to consistently take into account proper scaling of dimensions to avoid to add meters to feet, for instance.

Let us consider the simple example of computing the velocity of a point v defined as the distance d over the time t. A correct implementation would be to assign $v := d/t$, while an incorrect one would be $v := d \cdot t$. We can verify that the latter assignment is incorrect since the physical dimensions of the left and right sides of the assignment do not match. Namely, we expect $v_{[LT^{-1}]}$ to have dimension length L over time T, but $d_{[L]} \cdot t_{[T]}$ has dimension $(d \cdot t)_{[LT]}$.

Dimensional Constraints. Formally, the dimension D of a physical quantity q is represented as the product of powers of a fixed set of base dimensions B_i as $D = B_1^{n_1} B_2^{n_2} \cdots B_k^{n_k}$. For example, the dimension of our velocity variable is $L^1 T^{-1}$ where L and T are base dimensions. Since base dimensions are fixed, dimensions are uniquely identified by the set of powers $D \equiv \{n_1, ..., n_k\}$. While the set of base dimensions is known from physics, we will show that it is best to let the programmer define the base dimensions when implementing a rendering system. For this reason, we introduce formalism and implementation for a set of generic base dimensions.

Dimensions form an Abelian group and are closed under multiplication [LBB84]. Let a and b be two quantities of dimensions $D_a = \{n_1^a, ..., n_k^a\}$ and $D_b = \{n_1^b, ..., n_k^b\}$.

Addition $a+b$, subtraction $a-b$ and assignment $a := b$ can only be performed if $D_a = D_b$ and the result has the same dimension as the operands. Multiplication $a \cdot b$ and division a/b are defined for all D_a and D_b and have dimensions $D_{a \cdot b} = \{n_1^a + n_1^b, ..., n_k^a + n_k^b\}$ and $D_{a/b} = \{n_1^a - n_1^b, ..., n_k^a - n_k^b\}$. Functions of the form $f : D_1 ... D_j \rightarrow D_r$ can be applied to arguments when the dimension of each argument matches the expected one in the function definition.

Runtime Dimensional Analysis. Common implementations of dimensional analysis for many programming languages use run-time checking to enforce correctness [Hil88, CG88]. As an example, we could define a physical quantity as a tuple containing its value and the powers defining its dimension. At runtime, we can then validate the rules defined above. Since these tests need to be performed for all arithmetic operations, these runtime methods decrease performance too much to be useful in rendering algorithms.

Type-Checked Dimensional Analysis. [Ken97] has shown that by extending the type system of a functional language, we can provide compile-time dimensional analysis by adding dimensions as annotations to existing types and by extending the type checker to perform dimensional analysis. Specifically, we can represent physical quantities with a generic type, e.g. `float<D>`, where the generic type argument `D` is a type representing a dimension. A modified type checker would then validate that all arithmetic operations, assignments and function calls obeying the rules stated above are valid. For example, to compute velocity, we would declare the variables as `float<T> t;`, `float<L> d;` and `float<V> v = d/t;`, where `T`, `L` and `V` are types that represent time, length and velocity. To validate the assignment, the type checker would compute the type of the expression `d/t` and verify that this is the same type as `v`. The main advantage of this model is that it has little to no runtime overhead, since for statically typed languages type checking is done at compile time, and is easy for the programmer to learn.

Since dimensions form an Abelian group, standard type checking cannot be applied to check the equivalence of types representing dimensions [Ken97]. For example, if `A` and `B` are types representing dimensions, the type checker has to know how to compute a type to represent their product. The recently released F# [Har08], a functional language targeting the .NET runtime environment, implements a modified type system that provides dimensional analysis based on [Ken97]. Since most rendering systems are implemented in imperative languages, namely C++, the adoption of that language is unlikely. At the same time, it has been shown that C++ can be bent to implement dimensional analysis by using template metaprogramming [Umr94]. More recently the boost libraries [Kar05] have added support for dimensional analysis. *SafeGI* C++ components are based on these principles.

Space Analysis. In all production rendering systems we are aware of, geometric quantities like points, vectors and normals, are represented in cartesian coordinates with respect to some coordinate system, or space. Different spaces are used in different parts of the renderer. E.g. most raytracers perform lighting computations in world space, while they store shape geometry in model space. When mixing quantities defined with respect to different spaces, problems might arise when geometric computations are performed. E.g., one cannot subtract two points defined in different coordinate spaces. Let us consider the simple example of computing the vector $\mathbf{v} := \mathbf{p}_1 - \mathbf{p}_2$. This implementation is incorrect if the points $\mathbf{p}_1$ and $\mathbf{p}_2$ are not defined in the same space.

Space Constraints. Formally, coordinates of points, vectors and normals are represented with respect to a space S. Different components of rendering algorithms perform computations in different spaces. Geometric operations between points, vectors and normals are well defined only if the operands are represented in the same space. Formally, $c_{[S]} := a_{[S]} \oplus b_{[S]}$ for all S, where $\oplus$ is any geometric operation, such as addition, subtraction and dot and cross products. Space transformations, which we call maps, can be applied to geometric quantities to change the reference space. Formally, a map $m_{[S_1 \rightarrow S_2]}$ is a function that transforms geometric quantities from S_1 to S_2. Maps can be combined through function composition only when input and output spaces match; i.e. we can define $m_{[S_1 \rightarrow S_2]} = m_{[S \rightarrow S_2]} \circ m_{[S_1 \rightarrow S]}$.

Runtime Space Analysis. [DeR92] proposes a coordinate-free geometry model for geometric computations where points, vectors and normals store the space with respect to where they were defined. At runtime, changes of coordinate system operations are performed if required to ensure correctness of the geometric operations. This unfortunately leads to considerable runtime space and time overhead when applied to production rendering systems. Furthermore, programmers are not familiar with this formalism, leading to potential problems in its use in shading languages. To the best of our knowledge, no production rendering systems or shading languages use this approach. Within shading languages, RSL allows shader writers to define the space with respect to where a geometric quantity is defined when a variable is initialized. After the initial definition though, no checks are performed to avoid runtime costs, essentially leaving to the programmer the task of ensuring correctness. Real-time shading languages ignore the issue. In *SafeGI*, we use static type checking to perform this analysis for the first time.

3. *SafeGI* Rendering

***SafeGI* Renderers**. In this paper, we implemented three renderers to test the feasibility of deploying dimensional and space analysis in rendering systems. We used static type checking and generic programming to perform all analysis at compile time, minimally affecting execution speed. In supplemental material, we provide the full source code of all

renderers in two versions: one with type checked analyses and one without.

***SafeGI*'s CPU Renderer**. The first two renderers are simple codebases written from scratch to demonstrate how a new system should be developed with *SafeGI*'s annotations in mind. Specifically, we implemented a C++ path tracer where we model shapes, lights, materials and light transport similarly to [PH04]. We support direct and path tracing integrators, multiple importance sampling, opaque BRDFs (lambert, phong, mirror), point and area lights and motion blur of non-deformable objects.

***SafeGI*'s GPU Renderer**. Based on the same scene representation, we also implemented a GPU renderer to preview direct illumination and motion blur, using OpenGL and GLSL. The main benefits of using these simple, readable implementations is that readers can fully appreciate how to develop a rendering system with *SafeGI* annotations by comparing side-by-side the checked and unchecked implementations.

***SafeGI* in PBRT**. To test the method's scalability with system complexity, we inserted *SafeGI* type annotations into a subset of the *PBRT* [PH04] open source renderer. Since *PBRT* is a modular renderer based on plugins, we chose to port the core renderer together with path tracing and photon mapping light transport algorithms, a few textured materials (lambert, fresnel plastics and mirrors), and point, area and environmental map lights. The renderer supports motion blur and depth of field.

3.1. Dimensional and Space Analyses Implementation

Overview. In this section, we discuss the implementation of dimensional and space checking within the *SafeGI* renderers. In summary, we use type checking and template metaprogramming on the C++ components of the renderers. For the GPU components, we integrate an additional type checker to perform the analyses on shading language code and use type checking and generic programming to modify the rendering APIs to apply the analyses to the communication between CPU and GPU.

3.1.1. CPU Rendering Component

Dimensional Analysis. In the CPU component of *SafeGI*, we implemented dimensional analysis loosely following [Umr94]. We support user-defined base dimensions and units, while imposing no significant runtime overhead. We considered using an off-the-shelf implementation, such as boost [Kar05], but found the required notation to be quite verbose and the compile time significantly slower. We give here a sketch of these implementations, referring the reader to [Umr94, Kar05] for a complete discussion.

Physical quantities are represented as template types containing only one variable to store the numeric value of the quantity. E.g., `struct mfloat<D>{float v;}` where `D` is a type name. Dimensions are represented as a generic type whose type parameters are the dimension's integer powers. E.g. `struct D<N1,...,Nk> { }` where `N1,...,NK` are integer template arguments. E.g., the length dimension could be represented as `D<1,0,...,0>`, the time dimension as `D<0,1,...,0>` and the dimension representing velocity as `D<1,-1,...,0>`. Typedefs can be used to provide shorthand notations for these types. Operator overloading is used to define arithmetic operations with dimensional semantic. To compute the types representing dimensions of arithmetic expressions, template metaprogramming is used at compile time to perform arithmetic operations on the integer powers. For example, a multiplication operation should be defined as a function that takes in two float point numbers of type `mfloat<D<N1,N2,..Nk>>` and `mfloat<D<M1,M2,..,Mk>>`. The return value of the function is a float point number of type `mfloat<D<N1+M1,N2+M2,..,Nk+Mk>>`. In this case, when a float in time dimension and a float in velocity dimension is passed into the function, the C++ compiler can determine at compile time that the return value of this method is a float point value in length dimension of type `struct mfloat<D<1,0,..,0>>`.

Physical Units. Physical quantities are measured with respect to reference units. For example, a quantity of dimension length can be measured in meters or inches. When performing computations, quantities should only be combined when measured with respect to the same unit. A simple manner to enforce this is to store all values of quantities of the same dimension with respect to a single reference unit. Different units are constant scalar factors that convert a physical constant to the reference unit. E.g., `10.0*meters` and `10.0*inches` define quantities of dimension length, where `meters` and `inches` are constants of type `mfloat<length>`. While it is also possible to store runtime values with respect to different reference units [Kar05], runtime unit conversions produce slowdowns that discourage this use.

Space Analysis. In the CPU component of *SafeGI*, we implemented geometric space analysis by employing generic programming and static type checking. We enforce the correctness with respect to multiple geometric spaces, while imposing no significant runtime overhead.

Geometric quantities are represented as template types containing variables that store cartesian coordinates. E.g., `struct point<S>{mfloat<length> x,y,z;}` where `S` is a type name indicating a space. Spaces are simply defined as empty types. E.g., `struct world_s{}`. Geometric operations are defined by generic functions that take and return quantities defined in the same space. E.g., `point<S> operator-(point<S> a, point<S> b)`. Maps are represented as generic types that wrap linear algebra packages (in our case, our own simple repre-

sentation). E.g., a map $m_{[S_1 \to S_2]}$ would be implemented `struct map<S1,S2>{...}`. Map combinations and transformations are generic functions defined only for valid types.

3.1.2. GPU Rendering Component

Design Goals. In adding *SafeGI* type checking to our GPU renderer, our main design goal was to maintain the overall design of the OpenGL API we are using in our implementation. This has the benefit of programmer familiarity and further demonstrates the practicality of *SafeGI* annotations. Furthermore, our goal was not to add annotations to the entire API and shading language, but to provide enough coverage to demonstrate various aspects of shader and API programming.

Shading Language Extensions. Most commonly used real-time shading languages, including GLSL [Ros05], Cg [MGAK03] and HLSL [Bly06], do not support generic programming. For this reason, we cannot use the above implementations directly. In our prototype system, rather than implementing a full template engine in the shading language, we extended the GLSL shading language to support the additional type annotation used by *SafeGI*. Specifically, all variable types (i.e. uniform constant, vertex attributes and function and local variables) can be declared with *SafeGI* dimension and space type annotations using the same syntax as C++ to maintain familiarity. We provide a set of built-in arithmetic, geometric and texture operations to manipulate these quantities. We suggest the reader to consult the supplemental material for examples of shader code.

Type Checking. Standard GLSL shader programs are compiled at runtime by passing shader code to the driver. We follow the same design in our implementation, but add an additional type-checking stage before driver compilation. In particular, we implemented a simple type checker to validate dimensional and space analyses. If the code passes the type checker, all *SafeGI* annotations are elided by the type checker and the resulting code is then passed to the driver with standard API calls. Note that while we perform the check at runtime, this is only done once per shader compilation, so has negligible effect on rendering time. We did this to maintain the design of the OpenGL APIs.

Shader Parameters Binding. In *SafeGI*, we perform parameter binding similar to OpenGL, but store the handle as a generic type whose type parameter indicates the type of the shader variable. I.e., `struct id<T>{uint l;}`, where `T` is the variable type, e.g. `float<length>`. We query the handle by invoking a templated version of the OpenGL function. I.e., `id<T> l = safeglUniformLocation<T>("x")`. In our case, the query might fail if the variable name `"x"` does not exist (as in OpenGL) or if the type declared in the shader code does not match `T` (we store shader variable types in a symbol table during type checking). To assign values used in the typed handle, we provide a set of templated functions only defined when the value and handle have the same type. E.g., `safeglUniform<T>(T x, id<T> l)`. The same scheme works for binding any parameter type, including constants, vertex attributes and textures.

GPU Resources. The last component needed is the ability to keep type information for GPU resources, such as textures and FBOs. We also want to represent the object handle as a generic type, which is assigned at creation time. However, because OpenGL is a state machine, there is no strong connection between resource handle binding and resource management. In order to maintain type constraints in resource binding and management, we bundle several OpenGL function calls into one typed function. For example, we defined a function named `safeglCreateTexture2D<T>(id<T>& tid, ...)` which creates a texture with type `T` and returns a typed texture handle. In this function, a OpenGL texture handle is generated, then a texture of type `T` is created after binding that handle. In the end, a typed texture handle will be passed out of the function. Subsequent use of the texture object in binding, load and read operations would be handled in the same way and type checked with CPU values using the handle type. FBOs can be handled similarly. We refer to the attached source code for example uses.

3.2. Implementing *SafeGI* Renderers

Overview. This section presents a short summary of implementation details regarding dimensional and space analysis in our renderers. We also include a simple example to show how naturally physical dimensions and spaces translate to *SafeGI* annotations.

Geometric Space Analysis. All geometric quantities in *SafeGI* renderers are defined with respect to a set of spaces that are user-defined at compile time. Surfaces, lights and cameras store a transform object to perform transformations from world space (the renderer default) to a local space specific to the object class, where most computations are performed. For example, surfaces contain a world-to-shape transform; geometry data is stored with respect to that and ray-intersection computations are performed there. For the GPU renderer, we defined an additional project space and a homogeneous point type to represent projected quantities.

We originally implemented *SafeGI* renderers using the common practice of providing different types for points, vectors and normals. We found though that typecasts were often necessary throughout the codebase, since the vector datatype is used not only to represent vectors, but also pure directions. By introducing a separate type for directions, typecasts were eliminated, producing a codebase that was not only more readable, but where code speed improved slightly (by avoiding extra normalizations). This shows one of the benefits of using proper types for rendering objects.

Dimensional Analysis. In *SafeGI*, physical quantities are represented with respect to abstract base dimensions, rather than specific units. Specifically, we define base dimensions for length, time, energy and unit. This helps the programmer focus on the semantics of a dimension rather than the details of specific units. All physical quantities in our implementation are defined with generic types of the form `basetype<D>`, as discussed before, with the exception of dimensionless quantities that are store in standard types.

To further strengthen type checking, we explicitly create base dimensions for physical quantities that are formally dimensionless. Specifically, we add base units for solid and planar angles. We do so to ensure that the type checker can distinguish between different types of dimensionless quantities. For example, we would not want to accidentally use planar angles instead of solid angles in light transport computations. Furthermore, the addition of base dimensions lets us check for unit correctness so that degrees and radians are properly checked for. A minor drawback is that the programmer might have to explicitly typecast some computations. For example, this might happen when assigning the ratio of area and squared distance to a solid angle. We found this drawback to be minor since these cases are rare.

This simple example illustrates an important aspect of the *SafeGI* type system. By allowing the programmer to define any base dimension, we provide an extensible mechanism to add type checking for any quantity that respects the Abelian Group semantic, even if it might not have a physical counterpart (see F# for further discussion). For example, the programmer could add a base dimension to represent pixels in the rendering system. We take advantage of this flexibility by defining an additional dimension *proj* that represents the projection of incoming radiance, to ensure that such projection is never missing from light transport computations. We found that the major benefit of dimensional analysis is in the code that manipulates spectrum datatypes, since they can represent very different physical quantities (i.e. radiance, power, BRDFs, etc.).

Computing Direct Illumination. It is helpful to consider a simple example of light transport computation to make concrete observations about *SafeGI* implementations. Let us consider the case of computing direct illumination for a point x lit by one light source:

$$L(\mathbf{x},\psi) = \int_\Omega \hat{L}(\mathbf{x},\omega)\rho(\mathbf{x},\omega,\psi)V(\mathbf{x},\omega)(\mathbf{n}\cdot\omega)d\omega$$

where ω, ψ are the incoming and outgoing angles, V is the visibility function, $\hat{L}$ is the light self-emitted radiance, ρ is the BRDF, V is the visibility function and $(\mathbf{n}\cdot\omega)$ is the cosine of the incoming direction. We can compute this using Monte Carlo integration by

$$L(\mathbf{x},\psi) \approx \frac{1}{N}\sum_s \frac{\hat{L}(\mathbf{x},\omega^s)\rho(\mathbf{x},\omega^s,\psi)(\mathbf{n}\cdot\omega^s)V(\mathbf{x},\omega^s)}{pdf(\omega^s)}$$

Let us now add annotations for dimension and spaces:

$$L_{[R]}(\mathbf{x}_{[W]},\psi_{[W]}) \approx \frac{1}{N}\sum_s \frac{\hat{L}_{[R]}(\mathbf{x}_{[W]},\omega^s_{W})\rho_{[1/SP]}(\mathbf{x}_{[W]},\omega^s_{[M]},\psi_{[M]})(\mathbf{n}_{[W]}\cdot\omega^s_{[W]})_{[P]}V_{[-]}(\mathbf{x}_{[W]},\omega^s_{[W]})}{pdf_{[1/S]}(\omega^s_{[W]})}$$

where W and M are respectively world space and the space defined by the surface local frame, while R, S and P are the dimensions for radiance, solid angle, and projection. The visibility function is dimensionless. In the above notation, we omit dimensions for points and directions since they are the same for all geometric quantities.

The dimension and space annotations shown in the above example translate directly into type annotations in *SafeGI*. For example, the BRDF object has a method that takes two variables of type `direction<local_s>` and returns a value of type `spectrum<brdf_d>`, where `local_s` is the type for the local space and `brdf_d` is the type for the BRDF dimension. Similarly, light objects have a method to perform importance sampling that takes a point as `point<world_s>` and two random numbers and returns the emitted radiance as `spectrum<radiance_d>`, the sampling pdf as `float<invsolidangle_d>`, the sampling direction as `direction<world_s>` and the point to light distance as `float<length_d>`. This simple example shows that physical dimensions and spaces translate naturally to *SafeGI* annotations. We refer the reader to the supplied codebases for further examples.

3.3. Best Practices

This section summarizes a set of practices we found useful when implementing *SafeGI* renderers.

Base Dimensions. The use of additional base dimensions for angle, solid angle and projection provided us with tighter type checking, which we found useful in quickly detecting coding mistakes. Furthermore, it also improved the readability of code by more explicitly defining types; an example of this is reflected in the various pdfs used in the code. This suggests that adding even more of these annotations might prove useful.

Unit Conversion. To support different units for each dimension, but ensure that minimal overhead is incurred, renderers should perform all computations using only one reference unit per dimension. Quantities can still be defined with respect to additional units, but we suggest performing all unit conversions when such quantities are first defined, namely doing scene loading.

Interfacing with Standard Libraries. When interfacing with standard libraries that do not have *SafeGI* annotations, typecasting to and from standard C++ types is necessary. In our development we found it useful to provide *SafeGI* wrappers for the most used library functions. For example, we define functions for trigonometric operations such that angle units are properly checked. Another obvious example is

our OpenGL implementation where we wrap GL libraries in typed templated methods. This did not affect performance significantly, while letting the programmer work in a fully annotated system.

Type Inference. To further reduce programmer effort in adding annotations, we suggest using type inference when available. In particular, we implemented *SafeGI* using C++ 0x [Str05] to take advantage of type inference during initialization. E.g., we can simply say `auto v = expr;` to declare the variable `v` as the same type as the expression `epxr`. In our experience, we found that this drastically reduces the number of additional annotations a programmer has to specify.

4. Results

***SafeGI*'s Renderers**. We produce two versions of each *SafeGI* renderer: one with additional type checking for dimensional and space analyses, the other with these additional types removed. For *SafeGI* CPU and GPU renderers, we obtain the unchecked versions by substituting checked types with unchecked ones. This ensures that the only difference between the codebases are the types used to store all quantities and to perform computation.

In *SafeGI*'s *PBRT* port, the *PBRT* code had to be modified slightly when introducing *SafeGI* type annotations. This is due to the fact that the original renderer was not written with dimensional and space correctness in mind and in some rare instances it violates dimensional and space analyses, e.g. bump mapping and computing surface point/normal derivatives. We specifically did not optimize the *SafeGI* port, since our goal is to measure the cost associated with the additional type checks.

Performance. Table 1 shows the performance comparison between the typed and untyped implementation on some rendering scenes. In our settings, all source code was compiled using Intel® C++ Compiler Professional Edition 11 and run on a 32bit Windows 7 PC with an Intel® dual core 2.66GHz CPU, 4.00GB memory and a GeForce 8800 graphics card. All the images generated by our *SafeGI* CPU and GPU renderers are at a resolution of 512×512, while images generated by the *SafeGI* PBRT renderer are at a resolution of 1024×512 with 16 light samples.

As it can be seen, the CPU and GPU renderers show no decrease in performance when using dimensional and space analysis as implemented by *SafeGI*. In the case of *PBRT*, *SafeGI*'s port has a slowdown of roughly $3-4\%$. We believe this is mainly due to inefficiencies introduced in the port in places where the original renderer broke dimensional and space analysis. Finally, these results show that the benefit of additional type checking comes at almost no cost to performance.

Programmer Productivity. In our experience using *SafeGI* annotations, the human time required to add annotations themselves was not significant compared to total development time. While it is for us to measure this quantitatively, we did observe that one of the paper authors, with no prior knowledge of the *PBRT* codebase, took roughly 70 man-hours to port the *PBRT* subset described in the previous section, corresponding roughly to 13,400 lines of code. We believe this to be quite fast considering the the programmer had no expertise with *PBRT* and that we changed all basic types used in the systems.

					avg. time (sec.)	
	img.	feat.	samp.	prim.	typed	untyped
CPU	A	motion	256	35K	352.4	352.1
	B	dof	256	875K	1178.0	1177.3
	C	path	256	35K	1330.3	1323.9
GPU	D	direct	512	35K	8.1	7.9
	E	motion	512	35K	8.2	8.0
	F	direct	512	1715K	336.0	329.4
PBRT	G	direct	256	3404K	5375.1	5347.3
	H	direct	64	1144K	8088.0	7879.2
	I	path	512	65K	9907.6	9634.8
	J	photon	16	65K	5292.5	5180.9
	K	photon	32	1160K	1571.2	1545.0
	L	direct	256	15K	1088.3	1080.7

Table 1: *SafeGI rendering results*

In using *SafeGI* in the CPU and GPU renderers, we found that it was easier to find subtle bugs while developing code. E.g., the compiler quickly found errors in missing cosine projection and improper normalization of energy in some of our light sampling. Unfortunately it is impossible for us to quantitatively measure how many bugs were avoided only due to *SafeGI*. We considered running a user study to collect usage statistics, but it became clear to us that the amount of programming time required was too significant to include in this paper. In fact, we are not aware of any formal study on programmer productivity in the context of rendering systems or shader development. Still, it is our belief that since *SafeGI* can be seen as extending standard type checking, programmers should reap similar benefits.

5. Discussion and Limitations

Runtime Overhead. Our results show that for codebases developed with *SafeGI* in mind, runtime overhead is negligible compared to execution speed. This is also confirmed by extended testing performed for the boost [Kar05] implementation of dimensional analysis.

In *SafeGI* CPU components, we rely on the compiler optimization engine to remove the additional type checking. From our results, we conclude that current compiler optimizers perform well. There is no noticeable slow down in our prototype CPU renderer. In *SafeGI* GPU components, no

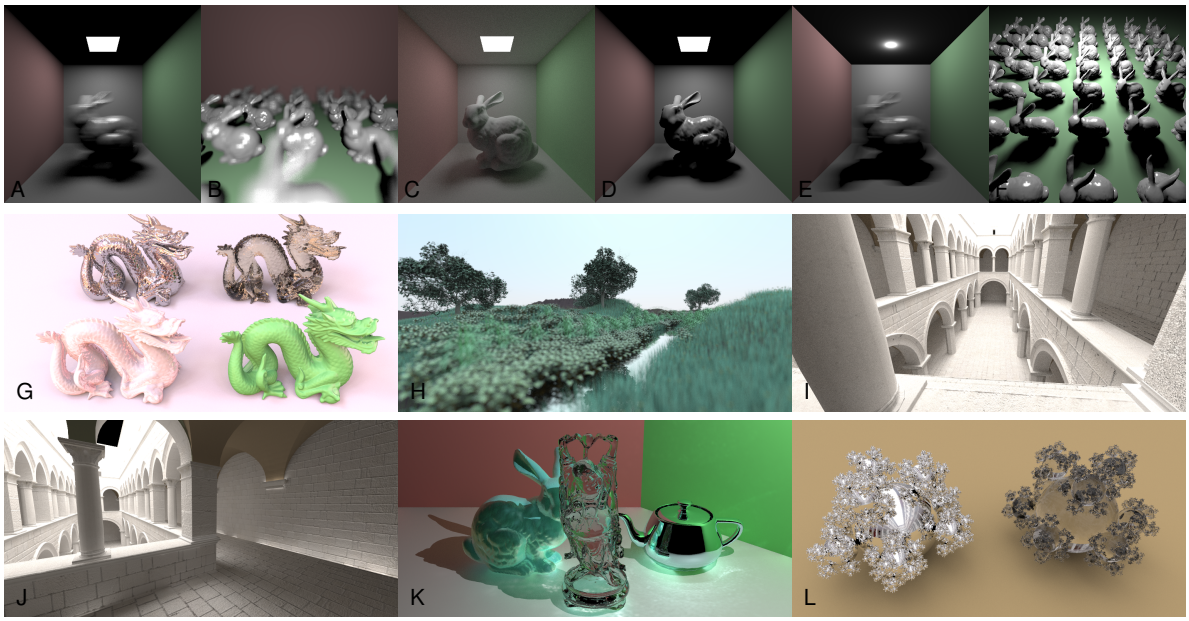

Figure 1: *Rendering Image of SafeGI and SafePBRT*

overhead is added since the additional type annotations are elided by the type checker before compilation. Furthermore, memory usage is not changed by introducing additional type checks.

Programmer Workflow. We believe that programmers' familiarity with type checking and generic programming makes introducing *SafeGI* type annotations easy for programmers to understand and integrate into their programming workflow. Furthermore, all IDE tools available for traditional programming, e.g. code completion and inspection tools, work well with *SafeGI*.

Code Understanding. We believe that *SafeGI* annotations are not only helpful in preventing bugs, but they make the code more understandable. By making the programmer responsible for explicitly declaring dimensions and spaces, a deeper understanding of the codebase is built. Furthermore, when passing code to others, more annotations are available to improve understanding. A case where this proved to be quite clear is when dealing with various pdfs used in our Monte Carlo renderer.

Limitations. We believe that the main limitation of this paper is the lack of a quantitative measure of how much *SafeGI* speeds up programmer workflow. Essentially, there is a tradeoff between the cost of adding additional types and the benefit of preventing bugs. In our experience, we found the former to be small enough to suggest that *SafeGI* annotations should be included in rendering and shading system development. At the same time, we hope that future work can more formally measure programmer workflow.

Data Driven Application. For a few scenarios we can think of where runtime content defines its own types for variables and data, static type checking is either impractical or unnecessary. If the C++ host does not need to manipulate the data of these types, the host can simply treat the content (data and shaders) as black boxes - we can still use our type checking in shader code, but not when it communicates with the host. If the host does need to manipulate the new types, or if the shaders need to access typed host data, the host must have equivalent static types, this raises another major technical difficulty which different content may have a same type signature (e.g. type name) with two different meanings. An equivalent runtime type cannot be defined unless there is a unified type signature model for all the contents.

6. Conclusion and Future Work

This papers presents a method to integrate dimensional and space analyses into the development of rendering systems and shaders. We use static type checking and generic programming to perform these analyses at compile time for CPU code, GPU shaders and GPU rendering APIs. In the future, we plan to investigate further methods to improve rendering system implementation correctness such as design by contract or formal semantic.

Acknowledgment

We would like to thank Lori Lorigo for her help in preparing this paper. The plant scene is courtesy of [DHL*98]. This

work was supported by NSF (CNS-070820, CCF-0746117), Intel and the Sloan Foundation.

References

[Bly06] BLYTHE D.: The Direct3D 10 system. *ACM Trans. on Graphics 25*, 3 (2006), 724–734. 1, 5

[CG88] CMELIK R. F., GEHANI N. H.: Dimensional analysis with C++. *IEEE Software 5*, 3 (1988), 21–27. 3

[DeR92] DEROSE T. D.: Three-dimensional computer graphics. a coordinate-free approach. Manuscript, University of Washington, 1992. 3

[DHL*98] DEUSSEN O., HANRAHAN P. M., LINTERMANN B., MECH R., PHARR M., PRUSINKIEWICZ P.: Realistic modeling and rendering of plant ecosystems. In *Proceedings of SIGGRAPH 98* (July 1998), Computer Graphics Proceedings, Annual Conference Series, pp. 275–286. 8

[Har08] HARROP J.: *F# for Scientists*. Wiley-Interscience, 2008. 3

[Hil88] HILFINGER P. N.: An Ada package for dimensional analysis. *ACM Trans. Program. Lang. Syst. 10*, 2 (1988), 189–203. 3

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. In *SIGGRAPH '90* (Aug. 1990), pp. 289–298. 1

[Kar05] KARLSSON B.: *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005. 3, 4, 7

[Ken94] KENNEDY A.: Dimension types. In *ESOP '94: Proceedings of the 5th European Symposium on Programming* (1994), pp. 348–362. 2

[Ken97] KENNEDY A. J.: Relational parametricity and units of measure. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1997), pp. 442–455. 3

[LBB84] LANKFORD D., BUTLER G., BRADY B.: Abelian group unification algorithms for elementary terms. *Contemporary Mathematics 29* (1984), 193–199. 2

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. on Graphics 22*, 3 (2003), 896–907. 1, 5

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004. 2, 4

[Ros05] ROST R. J.: *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005. 1, 2, 5

[SBW*09] SHREINER D., BOARD O. A. R., WOO M., NEIDER J., DAVIS T.: *OpenGL(R) Programming Guide, Version 3.0 and 3.1*. Addison-Wesley Professional, 2009. 2

[Str05] STROUSTRUP B.: The design of C++ 0x. *C/C++ Users Journal 23*, 5 (2005). 7

[Umr94] UMRIGAR Z. D.: Fully static dimensional analysis with C++. *ACM SIGPLAN Notices 29*, 9 (1994), 135–139. 3, 4