Geometry Types for Graphics Programming

ANONYMOUS AUTHOR(S)

In domains that deal with physical space and geometry, programmers need to track the coordinate systems that underpin a computation. We identify a class of *geometry bugs* that arise from confusing which coordinate system a vector belongs to. These bugs are not ruled out by current languages for vector-oriented computing, are difficult to check for at run time, and can generate subtly incorrect output that can be hard to test for.

We introduce a type system and language that prevents geometry bugs by reflecting the coordinate system for each geometric object. A value's *geometry type* encodes its reference frame, the kind of geometric object (such as a point or a direction), and the computational representation (such as Cartesian or spherical coordinates). We show how these types can rule out geometrically incorrect operations, and we show how to use them to automatically generate correct-by-construction code to transform vectors between coordinate systems. We implement a language for graphics programming, Gator, that checks geometry types and compiles to OpenGL's shading language, GLSL. Using case studies, we demonstrate that Gator can raise the level of abstraction for shader programming and prevent common errors without inducing significant annotation overhead or performance cost.

1 INTRODUCTION

Applications across a broad swath of domains use linear algebra to represent geometry, coordinates, and simulations of the physical world. Scientific computing workloads, robotics control software, and real-time graphics renderers all use matrices and vectors pervasively to manipulate points according to linear-algebraic laws. The programming languages that express these computations, however, rarely capture the underlying *geometric* properties of these operations. In domains where performance is critical, most languages provide only thin abstractions over the low-level vector and matrix data types that the underlying hardware (i.e., GPU) implements. A typical language might have a basic `vec2` data type for vectors consisting of two floating-point numbers, for example, but not distinguish between 2D vectors in rectangular or polar coordinates—or between points in differently scaled rectangular coordinate systems.

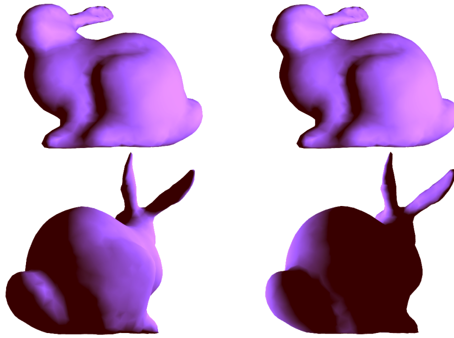
This paper focuses on real-time 3D rendering on GPUs, where correctness hazards in linear algebra code are particularly pervasive. The central problem is that graphics code frequently entangles application logic with abstract geometric reasoning. Programs must juggle vectors from a multitude of distinct coordinate systems while simultaneously optimizing for performance. This conflation of abstraction and implementation concerns makes it easy to confuse different coordinate system representations and to introduce subtle bugs. Figure 1 shows an example: a coordinate system handling bug yields incorrect visual output that would be difficult to catch with testing.

1.1 The Problem

Coordinate systems proliferate in graphics programming because 3D scenes consist of many individual objects. Figure 2 depicts a standard setup for rendering two objects in a single scene. Each object comes specified as a *mesh*, which consists of coordinate vectors for each vertex position. The mesh provides these vectors in a local, object-specific coordinate system called *model* space. The application positions multiple objects relative to one another in *world* space, and the simulated camera's position and angle define a *view* space.

Renderer code needs to combine vectors from different coordinate systems, such as in this distance calculation:

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/



(a) Correct implementation. (b) With geometry bug.

Fig. 1. Objects rendered with an implementation of the diffuse component of Phong lighting [Phong 1975], without (a) and with (b) a coordinate system transformation bug. The root cause is an incorrect spatial translation of the light source. The problem is only visible from one side of the model.

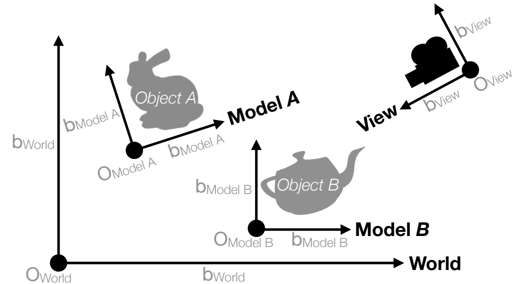


Fig. 2. Coordinate systems in graphics code. **Model A**, **Model B**, **World**, and **View** are coordinate systems. A coordinate system is defined by its basis vectors b and origin O. The **View** represents the perspective of a simulated camera.

```
float dist = length(teapotVertex - bunnyVertex);
```

This code may be incorrect, however, depending on the representation of the `teapotVertex` and `bunnyVertex` vectors. If the values come from the mesh data, they are each represented in their respective model spaces—and subtracting them yields a geometrically meaningless result. A correct computation needs to convert the operands into a common coordinate system using *affine transformation* matrices:

```
float dist = length(teapotToWorld * teapotVertex - bunnyToWorld * bunnyVertex);
```

Here, the `teapotToWorld` and `bunnyToWorld` matrices define the transformations from each model space into world space.

Geometry bugs are hard to catch. Mainstream rendering languages like OpenGL's GLSL [Segal and Akeley 2017] cannot statically rule out coordinate system mismatches. In GLSL, the variables `teapotVertex` and `bunnyVertex` would both have the type `vec3`, i.e., a tuple of three floating-point numbers. These bugs are also hard to detect dynamically. They do not crash programs—they only manifest in visual blemishes. While the buggy output in Figure 1b clearly differs from the correct output in Figure 1a, it can be unclear what has gone wrong—or, when examining the buggy output alone, that anything has gone wrong at all. Accordingly, writing assertions or unit tests to catch this kind of bug can be challenging: specifying the behavior of a graphics program requires formalizing how the resulting scene should be perceived. Viewers can perceive many possible outputs as visually indistinguishable, so even an informal specification of what makes a renderer "correct," for documentation or testing, can be difficult to write.

Geometry bugs in the wild. Even among established graphics libraries, geometry bugs can remain latent until a seemingly correct API change reveals the bug. For example, in LÖVR, a framework for rapidly building VR experiences, the developers discovered a bug where a variable that was in one space was being used as if it was in another.[1] This bug lay dormant while, to quote one of the

[1] https://github.com/bjornbytes/lovr/issues/55

maintainers, “there was a change in the rendering method that amplified the problems caused by this.” The maintainer then noted that they needed to “go backfill this fix to all the docs/examples that have the broken version.” Because their effects are hard to detect, geometry bugs can persist and cause subtle inaccuracies that grow as code evolves.

We found similar issues that arise when APIs fail to specify information about vector spaces. In the Processing graphical IDE, for example, confusion surrounding a camera API led to a 20-comment thread before a developer concluded that “better documentation could alleviate this to some extent: it needs to be clear that modelspace is relative to the camera at the time of construction.”[2] And in the visualization library GLVisualize.jl, users disagree about the space that the library uses for a light position.[3] The root cause in both cases is that the programming language affords no opportunity to convey vector space information.

This paper advocates for making geometric spaces manifest in programs themselves via a type system. Language support for geometric spaces can remove ambiguity and provide self-documenting interfaces between parts of a program. Static type checking can automatically enforce preconditions on geometric operations that would otherwise be left unchecked.

1.2 Geometry Types

We introduce a type system that can eliminate this class of bugs, and we describe a mechanism for automatic transformation that can rule out some of them by construction. *Geometry types* describe the coordinate system representing each value and the transformations that manipulate them. A geometry type encodes three components: the *reference frame*, such as model, world, or view space; the *geometric object*, such as a point or a direction; and the *coordinate scheme*, such as Cartesian or spherical coordinates. Together, these components define which geometric operations are legal and how to implement them.

The core contribution of this paper is that all three components of geometry types are necessary. The three aspects interact in subtle ways, and real-world graphics rendering code varies in each component. Simpler systems that only use a single label [Ou and Pellacini 2010] cannot express the full flexibility of realistic rendering code and cannot cleanly support automatic transformations. We show how encoding geometry types in a real system can help avoid and eliminate realistic geometry bugs. We will explore further how these components are defined and interact to provide operation information in Section 3.

We design a language, Gator, that builds on geometry types to rule out coordinate system bugs and to automatically generate correct transformation code. In Gator, programmers can write `teapotVertex in world` to obtain a representation of the `teapotVertex` vector in the `world` reference frame. The end result is a higher-level programming model that lets programmers focus on the geometric semantics of their programs without sacrificing efficiency.

We implement Gator as an overlay on GLSL [The Khronos Group Inc. [n. d.]], a popular language for implementing shaders in real-time graphics pipelines. Most GLSL programs are also valid in Gator, so programmers can easily port existing code and incrementally add typing annotations to improve its safety. We formalize a geometry type system and show that erasing these types preserves soundness. In our evaluation, we port rendering programs from GLSL to qualitatively explore Gator’s expressiveness and its ability to rule out geometry bugs. We also quantitatively compare the applications to standard GLSL implementations and find that Gator’s automatic generation of transformation code does not yield meaningfully slower rendering time than hand-tuned (and unsafe) GLSL code.

[2] https://github.com/processing/processing/issues/187
[3] https://github.com/JuliaGL/GLVisualize.jl/pull/188

This paper's contributions are:

- We identify a class of geometry bugs that exist in geometry-heavy, linear-algebra-centric code such as physical simulations and graphics renderers.
- We design a type system to describe latent coordinate systems present in linear algebra computations and prevents geometry bugs.
- We introduce a language construct that builds on the type system to automatically generate transformation code that is type correct by construction.
- We implement the type system and automatic transformation feature in Gator, an overlay on the GLSL language that powers all OpenGL-based 3D rendering.
- We experiment with case studies in the form of real graphics rendering code to show how Gator can express common patterns and prevent bugs with minimal performance overhead.

We begin with some background via a running example before describing Gator in detail.

2 RUNNING EXAMPLE: DIFFUSE SHADING

This section introduces the concept of geometry bugs via an example: we implement *diffuse lighting*, a component of the classic Phong lighting model [Phong 1975].[4] We assume some basic linear algebra concepts but no background in graphics or rendering.

2.1 Gentle Introduction to Shader Programming

Shader programs are code, typically written in C-like languages such as GLSL or HLSL, that runs on the GPU to render a graphics *scene*. The GPU executes a pipeline of shader programs, where each shader is specialized to transform a certain property of a graphical object. The shader pipeline consists of several stages. The most notable of these stages are the vertex shader, which outputs the position of each vertex as a pixel and the fragment shader, which outputs the color of each *fragment* corresponding to an on-screen pixel.

In graphics, the *scene* is a collection of objects. The shape of an object is determined by mesh data consisting of *position vectors* for each vertex, denoting the spatial structure of the object, and *normal vectors*, denoting the surface orientation at each vertex.

The kind of transformation each graphics shader applies to a graphical object depends on the pipeline stage. We focus on the vertex and fragment shader, the most common user-programmable stages of the graphics pipeline.

2.2 Diffuse Lighting

Diffuse lighting is a basic lighting model that simulates the local illumination on the surface of an object. Given a point on an object, the intensity of its diffuse component is proportional to the angle between the position of the light ray and the local surface normal. The diffuse model first computes the direction of the light by subtracting the mesh (surface) position, *fragPos*, from the light position:

$$\mathit{lightDir} = \text{normalize}(\mathit{lightPos} - \mathit{fragPos})$$

We normalize the vector, which preserves the angle but sets the magnitude to 1. We calculate the resulting diffuse intensity at this fragment as the angle between the incoming light ray and the fragment normal using the vector dot product (which is algebraically the sum of the product of vector components):

$$\mathit{diffuse} = \max(\mathit{lightDir} \cdot \mathit{fragNorm}, 0.)$$

[4] Appendix A gives a complete GLSL implementation of the Phong model.

197 The max function used here prevents light from passing through the object by rejecting reflection
198 angles greater than perpendicular.
199

200 2.3 Where Things Go Wrong: GLSL Implementation

201 To implement the diffuse lighting model, we must write a GLSL shader program that operates
202 on a per-fragment basis. This section shows how this seemingly simple program translates to
203 surprisingly complex code. We identify pitfalls in this implementation process that our type system
204 will address.
205 GLSL has vector and matrix types, with names like vec3 and mat4, along with built-in vector
206 functions that make an initial implementation of the diffuse component seem straightforward:

```
207 float naiveDiffuse(vec3 lightPos, vec3 fragPos, vec3 fragNorm) {
208   vec3 lightDir = normalize(lightPos - fragPos);
209   return max(dot(lightDir, normalize(fragNorm)), 0.);
210 }
```

211 Although lightPos and fragPos have the same type, they are not geometrically compatible: real
212 renderers need to represent them with different reference frames and coordinate schemes. While
213 this incorrect code directly reflects the mathematical description above, the output is nonetheless
214 incorrect: it produces the buggy output in Figure 1b.
215

216 *Coordinate Systems.* The underlying problem is that software needs to represent different vectors
217 in different coordinate systems. Information needed to render the shape of a single graphical object,
218 the positions and normal vectors, lies in the object's *model space*, as can be seen in Figure 2. A
219 model space represents the coordinates local to a single object in the scene. The origin of this space
220 is centered in the model, with basis vectors matching the model orientation and scale. Both may
221 change dynamically as time passes in the scene; however, each is fixed during a single iteration of
222 the shader. *World space* gives the absolute coordinates for the entire scene, so the basis vectors and
223 origin of world space are typically fixed.
224 Mesh data is scene independent, so we represent mesh parameters such as fragPos and fragNorm
225 initially in model space, independent of the object's current relative position within the scene. In
226 contrast, we represent the position of a light source relative to the entire scene—so lightPos is
227 in world space. As a result, the subtraction expression lightPos - fragPos attempts to compare
228 vectors represented in different spaces, yielding a geometrically meaningless result. This bug
229 produces the incorrect output seen in Figure 1b.
230

231 *Transformation Matrices.* To fix this program, the shader needs to *transform* the two vectors to a
232 common coordinate system before subtracting them. Mathematically, coordinate systems define
233 an affine space, and thus geometric transformations on coordinate systems can be linear or affine.
234 Affine transformations can change the origin and basis vectors, which can represent translation,
235 while linear transformations affect only the basis vectors, which can represent rotation and scale.
236 These geometric transformations are represented in code as *transformation matrices*. To apply a
237 transformation to a vector, shader code uses matrix-vector multiplication. For example, the shader
238 application may provide a matrix uModel that defines the transformation from model to world space
239 using matrix multiplication:

```
vec3 lightDir = normalize(lightPos - uModel * fragPos));
```

240

241 *Homogeneous Coordinates.* Unfortunately, this matrix multiplication implementation introduces
242 another bug. Transforming fragPos from model to world space requires both a linear scaling and
243 rotation transformation and a translation to account for change of origins. This linear transfor-
244 mations with translation is represented by an *affine transformation matrix*. This is a problem: an
245

246 affine transformation matrix for 3D vectors must be represented as a 4×4 matrix. To multiply this
247 matrix by fragPos (which is a 3-dimensional vector), we need a sensible representation of fragPos
248 as a 4-dimensional vector. It is thus not immediately clear by what vector we need to multiply:

```
vec3 lightDir = normalize(lightPos - vec3(uModel *?));
```

250 Because a 3×3 Cartesian transformation matrix on 3-dimensional vectors can only express linear
251 transformations, graphics software typically uses a second kind of coordinate system called *ho-*
252 *mogeneous coordinates*. An n-dimensional vector in homogeneous coordinates uses $n + 1$ values:
253 the underlying Cartesian coordinates and a *scaling factor*, w. A 4×4 transformation matrix in
254 homogeneous coordinates can express *affine* transformations on the underlying 3-dimensional
255 space, including translation.

256 To convert from Cartesian to homogeneous coordinates, a vector $[x, y, z]$ becomes $[x, y, z, 1.]$;
257 in the opposite direction, the homogeneous vector $[x, y, z, w]$ becomes $[x/w, y/w, z/w]$. To fix
258 our example to use the 4-dimensional affine transformation uModel, we can extend fragPos into a
259 homogeneous vec4 value:

```
vec3 lightDir = normalize(lightPos - vec3(uModel * vec4(fragPos, 1.)));
```

261 The GLSL functions vec4 and vec3 extend a 3-dimensional vector with the given component and
262 truncate a 4-dimensional vector, respectively. We now have a lightDir in a consistent coordinate
263 system, namely in the world space.

264 The final calculation of the diffuse intensity uses this expression:

```
max(dot(lightDir, normalize(fragNorm)), 0.)
```

267 Here, fragNorm resides in model space and should be transformed into world space. One tricky
268 detail, however, is that fragNorm denotes a *direction*, as opposed to a *position* as in fragPos. These
269 require different geometric representations, because a direction should not be affected by translation.
270 Fortunately, there is a trick to avoid this issue while still permitting the use of our nice homogeneous
271 coordinate representation. By extending fragNorm with $w = 0$, affine translation is not applied.

```
return max(dot(lightDir, normalize(vec3(uModel * vec4(fragNorm, 0.))));
```

273 This subtle difference is a common source of errors, particularly for novice programmers. Finally,
274 we have a correct GLSL implementation of diffuse. This version results in the correct output in
275 Figure 1a.

276
277
278 ## 3 GEOMETRY TYPES

279 The problems in the previous section arise from the gap between the abstract math and the
280 concrete implementation in code. We classify this kind of bug, when code performs geometrically
281 meaningless operations, as a *geometry error*. Gator provides a framework for declaring a type
282 system that can define and catch geometry errors in programs.

283 The core concept in Gator is the introduction of *geometry types*. These types refine simple
284 GLSL-like vector data types, such as vec3 and mat4, with information about the geometric object
285 they represent. A geometry type consists of three components:

286
287 - The *reference frame* defines the position and orientation of the coordinate system. A reference
288 frame is determined by its basis vectors and origin. Examples of reference frames are model,
289 world, and projective space.
290 - The *coordinate scheme* describes a coordinate system by providing operation and object
291 definitions, such as homogeneous and Cartesian coordinates. Coordinate schemes expresses
292 how to represent an abstract value computationally, which identifies what the underlying
293 GLSL-like type is.

294

295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343

- The *geometric object* describes which geometric construct the data represents, such as a point, vector, or transformation.

In Gator, the syntax for a geometry type is `scheme<frame>.object` This notation invokes both module members and parametric polymorphism. Coordinate schemes are parameterized by a reference frame, while geometric objects are member types of a parameterized scheme. For example, `cart3<world>.point` is the type of a point lying in world space represented in a 3D Cartesian coordinate scheme.

The three geometry type components suffice to rule out the errors described in Section 2. The rest of the section details each component.

3.1 Reference Frames

We can enhance the mathematical diffuse light computation above using geometry types:

```
float diffuseNaive(
    cart3<world>.point lightPos,
    cart3<model>.point fragPos,
    cart3<model>.direction fragNorm) {
  cart3<world>.direction lightDir = normalize(lightPos - fragPos);
  return max(dot(lightDir, normalize(fragNorm)), 0.0);
}
```

With these stronger types, the expression `lightPos - fragPos` in this function is an error, since `lightPos` and `fragPos` are in different frames. It is geometrically legal to subtract two positions to produce a vector; the only issue with this code is the difference of reference frames. We will further discuss how Gator determines subtraction is legal in Section 3.2.

Definition. Reference frames in Gator are labels with an integer dimension. The dimension of a frame specifies the number of linearly independent basis vectors which make up the frame. Gator does not require explicit basis vectors for constructing frames; keeping basis vectors implicit helps minimize programmer requirements and helps avoid cluttering definitions with information we don't really need. We will discuss what keeps these basis vectors are implicit through transformations between reference frames in Section 4.

The Gator syntax to declare the three-dimensional model and world frames is:

```
frame model has dimension 3;
frame world has dimension 3;
```

3.2 Coordinate Schemes

To transform `fragPos` and `fragNormal` to the `world` reference frame, we need to provide an affine transformation matrix `uModel`.

```
float diffuse(
    cart3<world>.point lightPos,
    cart3<model>.point fragPos,
    cart3<model>.direction fragNorm,
    hom3<model>.transformation<world> uModel) {
  cart3<world>.direction lightDir =
    normalize(lightPos - (uModel * fragPos));
  return max(dot(lightDir, normalize(uModel * fragNorm)), 0.0);
```

For this example, we define matrix–vector multiplication $m * v$ to update types akin to function application: it ensures that m is a transformation in the same frame as the vector and parameterized on the destination frame f, then produces an output direction in the frame f. With this definition,

multiplying `uModel` by an object in the `model` reference frame will result in an object in the `world` frame.

Unfortunately, multiplying `uModel * fragPos` produces a Gator type error since `uModel` and `fragPos` are in different coordinate schemes. We will resolve this issue in the next subsection by converting between schemes.

Definition. Coordinate schemes provide definitions of geometric objects and operations. Concretely, they consist of operation type declarations and concrete definitions for member objects and operations. Geometric operations defined in coordinate schemes are expected to provide geometrically correct code, and are generally intended (though not required) to operate between objects within the coordinate scheme. Recall that, instead of "baking in" a particular notion of geometry, Gator lets coordinate schemes provide types that define correctness for a given set of geometric operations.

```
with frame(3) r:
coordinate cart3 : geometry {
  object vector is float[3];
  ...
}
```

For example, we can define 3D vector addition in Cartesian coordinates, which consists of adding the components of two vectors together.

```
vector +(vector v1, vector v2) {
  return [v1[0] + v2[0], v1[1] + v2[1], v1[2] + v2[2]];
}
```

All coordinate schemes are required to be parameterized with reference frames, so `cart3<model>` and `cart3<world>` are different instantiations of the same scheme. Gator's `with` syntax provides parametric polymorphism in the usual sense; in this example, the 3-dimensional Cartesian coordinate scheme is polymorphic over all 3-dimensional reference frames.

3.3 Geometric Objects

To apply the `uModel` affine transformation to our position and normal, we first need to convert each to homogeneous coordinates. Recall from Section 2.3, however, that this coordinate system transformation *differs for points and directions.* To capture this distinction, we introduce the overloaded function `homify`:[5]

```
hom<model>.point homify(cart3<model>.point p) {
  return [p[0], p[1], p[2], 1.];
}
hom<model>.direction homify(cart3<model>.direction p) {
  return [p[0], p[1], p[2], 0.];
}
```

Unlike Cartesian coordinates, homogeneous coordinates have different representations for points and directions: the latter must have zero for its last coordinate, w.

To send `fragPos` and `fragNorm` to homogeneous coordinates, it suffices to call `homify` and let the Gator compiler select the correct overloaded variant:

```
homify(fragPos);  // Extends fragPos with w=1.
homify(fragNorm); // Extends fragNorm with w=0.
```

We repeat this process to define the function `reduce`, which maps homogeneous to Cartesian coordinates. Finally, we apply these functions to our model:

[5]For simplicity, this example `homify` is written only for objects in the `model` frame. Gator supports function parameterization on reference frames, so we would normally write `homify` to work on any frame.

```
float diffuse(
    cart3<world>.point lightPos,
    cart3<model>.point fragPos,
    cart3<model>.direction fragNorm,
    hom3<model>}.transformation<world> uModel) {
  cart3<world>.direction lightDir = normalize(lightPos - reduce(uModel * homify(fragPos)));
  return max(dot(lightDir, normalize(reduce(uModel * homify(fragNorm))), 0.0));
}
```

Now, by using all three components of the geometry type, our code will compile and produce the correct Phong diffuse color shown in Figure 1a.

Definition. The object component of a geometry type describes the type's underlying datatype and provides information on permitted operations. Object type definitions can be parameterized on reference frames, such as writing affine transformations *to* a specific frame. For example, we can define some objects in homogeneous coordinates:

```
coordinate hom3 : geometry {
  object point is float[4];
  object direction is float[4];
  with frame(3) r:
  object transformation is float[4][4];
  ...
}
```

Object and type declarations in Gator extend existing types; for example, here `point` is defined as a subtype of `float[4]`. When an operation is applied to one or more objects, Gator requires that they have matching coordinate schemes and that the function being applied has a definition in this matching scheme. For example, by omitting a definition for addition between `points` and their supertypes, we ensure that Gator will reject `fragPos + fragPos`.

4 AUTOMATIC TRANSFORMATIONS

Gator's type system statically rules out bad coordinate system transformation code. In this section, we show how it can also help automatically generate transformation code that is correct by construction. The idea is to raise the level of abstraction for coordinate system transformations so programmers do not write concrete matrix–vector multiplication computations—instead, they declaratively express source and destination spaces and let the compiler find the right transformations. A declarative approach can obviate complex transformation code that obscures the underlying computation and can quickly become out of date, such as this shift from `model` to `world` space:

```
cartesian<world>.direction worldNorm =
  normalize(lightPos - reduce(uModel * homify(fragNorm)));
```

We extend Gator with an `in` expression that generates equivalent code automatically:

```
cartesian<world>.direction worldNorm = normalize(lightPos - fragNorm in world);
```

The new expression converts a vector into a given representation by generating the appropriate function calls and matrix–vector multiplication. Specifically, the expression `e in scheme<frame>` takes a typed vector expression e from its current geometry type `T.object` to the type `scheme<frame>.object` by finding a series of transformations that can be applied to e. With this notation, either the `scheme` or `frame` can be omitted without ambiguity, so writing `x in world` where `x` is in scheme `cart3` is the same as writing `x in cart3<world>`. Gator `in` expressions can only be used to change the coordinate scheme or parameterizing reference frame; that is, the geometric object of the target type must be the same as the original value type.

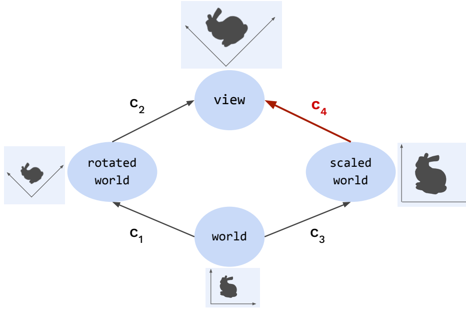


Fig. 3. A *transformation graph* with provided transformations. The highlighted edge represents a newly added transformation function, which must be unique and agree with the existing paths on the graph.

$$
\begin{aligned}
c &\in \text{constants} \\
x &\in \text{variables} \\
f &\in \text{function names} \\
p &\in \text{primitives} \\
t &\in \text{types} \\
\tau &::= \text{unit} \mid \top_p \mid \bot_p \mid t \\
e &::= v \mid c \mid f(e_1, e_2) \mid x \text{ as! } \tau \mid x \text{ in } \tau \\
C &::= \tau\, x = e \mid e \\
P &= C; P \mid \epsilon
\end{aligned}
$$

Fig. 4. Core Gator syntax.

Implementation. The Gator compiler implements in expressions by searching for transformations to complete the chain from one type to another. It uses a *transformation graph* where the vertices are types and the edges are transformation matrices or functions. Figure 3 gives a visual representation of a transformation graph.

4.1 Canonical Functions

The transformations that gator reasons about for automatic application are special: they must uniquely define a map from their domain to their range. Gator requires these functions to be labeled with the word canon. Gator defines three requirements on these transformations: (1) there can be only one canonical function between each pair of types in a given scope, (2) all canonical functions between reference frames must map between frames of the same dimension, and (3) a canonical function can only have one non-canonical argument.

To expand on condition (3); canonical functions may take in *canonical arguments*, which are variables labelled with the canon keyword. The most familiar example of this use is defining matrix—vector multiplication to be canonical; the matrix itself must be included and must be a canonical matrix:

```
with frame(3) target:
canon point *(canon transformation<target> t, point x) {
  ...
}
...
// Now declare the matrix as canonical for use with multiplication
canon hom<model>.transformation<world> uModel;
homPos in world; // --> uModel * homPos
```

It is legal to manually fill canonical arguments to functions with non-canonical variables; however, in expressions will never do so.

The intuition of canonical functions comes from affine transformations between frames and coordinate schemes. Since each frame has underlying basis vectors, transformations between frames of the same dimension which preserve these frames are necessarily unique; further, applying these bijective transformations does not cause data to "lose information." Similarly, coordinate schemes

491 simply provide different ways to view the same information; there are often unique transformations
492 between schemes that can be applied as needed to unify data representation.
493 This construction of canonical functions and automatic transformations is similar to constructions
494 provided by C# and C++'s type coercion. The slightly different approach needed for `in` expressions
495 will be discussed briefly in Section 8.
496
497 ### 4.2 Correctness of Generated Transformations

498 With `in` expressions, Gator programmers sacrifice control for convenience: the compiler picks
499 which transformation functions and matrices to use to get from one coordinate system to another.
500 If all the individual transformations marked with `canon` are correct, then the composed "chain"
501 generated for an `in` expression must also be correct. Functional verification of transformations,
502 however, is not feasible in Gator's purely static setting: it would require not only the value of
503 every transformation matrix, which typically varies dynamically over time, but also an intrinsic
504 description of each coordinate system, such as the basis vectors for every reference frame, which is
505 never available in real graphics code. We view heavyweight dynamic debugging aids for checking
506 transformation correctness as important future work.
507 We can, however, state a simple consistency condition that is necessary but not sufficient for
508 a system of canonical transformations to be correct. The transformation system should be *path*
509 *independent:* for any two types τ_1 and τ_2, the behavior of any chain of transformations from τ_1 to
510 τ_2 should be equivalent. In other words, every edge in the transformation graph corresponds to
511 a function—so every path corresponds to a function composition, and every such path between
512 the same two vertices should yield the same composed function. (This definition is equivalent to
513 commutativity for diagrams [Murota 1987].) Otherwise, the semantics of an expression e `in` τ
514 would depend on the graph search algorithm that Gator uses to find routes in the transformation
515 graph, which is clearly undesirable.
516 Because it is a purely static system, Gator does not enforce path independence. However, path
517 independence motivates Gator's requirement that canonical transformations preserve dimensional-
518 ity (see Section 4.1). Without this condition, we have found it is easy to accidentally violate path
519 independence with non-invertible functions and result in an ambiguous transformation graph for
520 `in` expressions.
521
522 ## 5 FORMAL SEMANTICS

523 Gator provides a framework for defining geometry types as an "overlay" on top of computation-
524 oriented programs in a base language without geometry types. In this section, we formalize a core of
525 Gator to show that its constructs are sound with respect to such an underlying language. The goal is
526 a theorem stating that well-typed Gator programs, when translated, result in well-typed programs
527 in the target language. We focus on the generic, extensible Gator language rather than formalizing
528 the rules for any specific geometric system—affine transformations on Cartesian coordinates, for
529 example. Proving soundness with respect to a linear algebra domain would be interesting future
530 work but is out of scope for this paper.
531 We define two languages: a high-level core semantics for Gator that includes its user-defined
532 types, and a low-level abstract target language, Hatchling. Hatchling represents a sound imperative
533 language with some set of primitive types and operators on those types. For example, an instance
534 with fixed-size vector and matrix types can reflect a simple core of GLSL.
535
536 ### 5.1 Syntax

537 Figure 4 lists the syntax of the formal core of Gator that we formalize in this section. The types in
538 this core language consist of `unit` and a lattice over each primitive type p. The choice of primitives
539

$$\frac{\tau_1 \leq \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \qquad \frac{\mathrm{X}(c) = p}{\Gamma \vdash c : \bot_p} \qquad \frac{\Gamma(v) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = \tau}{\Gamma \vdash \tau\ x = e : \text{unit}}$$

$$\frac{\Gamma \vdash C : \tau_1 \qquad \Gamma \vdash P : \tau_2}{\Gamma \vdash C; P : \text{unit}} \qquad \frac{}{\Gamma \vdash \epsilon : \text{unit}} \qquad \frac{\Gamma \vdash e : \top_p \qquad \tau \leq \top_p}{\Gamma \vdash e \text{ as! } \tau : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \mathrm{P}(\tau_1, \tau_2) = f}{\Gamma \vdash e \text{ in } \tau_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, \vdash e_2 : \tau_2 \qquad \Phi(f, \tau_1, \tau_2) = \tau_3}{\Gamma \vdash f(e_1, e_2) : \tau_3}$$

Fig. 5. Typing Judgment

is kept abstract in this formalism to highlight that the Gator extend over arbitrary underlying datatypes. For example, in a GLSL core language, we might have a primitive `float` or `vec3` – something like `vector` would be a custom type t and not a primitive.

A program in Gator is a series of commands; we simplify these to variable declaration, assignment, and expressions. Gator expressions are constructed around function applications, with `as` and `in` expressions to help manage types.. We assume functions always take two arguments for simplicity; extending this assumption for other argument counts is straightforward.

5.2 Typing Rules

We define a typing judgment for Gator programs, $\Gamma \vdash P : \tau$, that, for any program P and typing context Γ, produces a type τ. The complete semantics for this judgment can be seen in Figure 5. Note that Γ is kept constant throughout; declaring a variable requires looking up into the constant Γ to determine if the declared type matches the expected type. Keeping Γ constant will later help with translation; the type of any expression can be determined exactly from the constant global contexts Γ, X, and Φ along with the judgment P.

Gator requires a lattice for each primitive type; custom types on each lattice introduces new subtyping relations. We define a type ordering among types $\leq$ where $t_1 \leq t_2$ means that t_1 is a subtype of t_2. $\leq$ is expected to be reflexive and transitive. In a well formed program, $\leq$ must contain a rule for every user defined type, every type (except unit) must be a subtype of a primitive top type, and every bottom type $\bot_p$ must be a subtype of each subtype of the associated $\top_p$. In other words, $\leq$ must conform to a lattice structure for each primitive p. The complete summary of subtyping rules can be found in the attached supplementary materials.

The typing information for functions is stored in a function typing context, Φ, which maps the tuple of function name and input types to the output type. The semantics of Φ are built to support overloaded functions.

Gator, as defined in these semantics, is parameterized over primitive types stored in primitive type context X, which maps a literal to its primitive type.

The map from in expressions to paths is managed by the judgment P. More precisely, P maps a given start and end type τ_1 and τ_2 to a function name that, when applied to an expression of type τ_1, produces an expression of type τ_2. We simplify the judgment of P here to only allow one step for notation clarity; in the real Gator implementation, the transformation may be a chain of functions. The details of this judgment P are omitted for simplicity, but amount to a simple lookup through the available functions for a function of the correct type.

$$
\begin{aligned}
[\![c]\!]_\Gamma &\triangleq c & [\![x]\!]_\Gamma &\triangleq x \\
[\![\tau\ x := e]\!]_\Gamma &\triangleq [\![\tau]\!]\ x := [\![e]\!]_\Gamma & [\![e \text{ as! } \tau]\!]_\Gamma &\triangleq [\![e]\!]_\Gamma \\
[\![e \text{ in } \tau_2]\!]_\Gamma &\triangleq [\![f(e)]\!]_\Gamma & &\text{where } \Gamma \vdash e : \tau_1 \text{ and } f = \mathrm{P}(e, \tau_1, \tau_2) \\
[\![f(e_1, e_2)]\!]_\Gamma &\triangleq f'(e_1, e_2) & &\text{where } \Gamma \vdash e : \tau_1, \Gamma \vdash e : \tau_2, \text{ and } f' = \Psi(f, e_1, e_2, \tau_1, \tau_2) \\
[\![\epsilon]\!]_\Gamma &\triangleq \epsilon & [\![C; P]\!]_\Gamma &\triangleq [\![C]\!]_\Gamma ; [\![P]\!]_\Gamma \\
[\![t]\!] &\triangleq \top_p & &\text{where } t \leq \top_p \\
[\![\top_p]\!] &\triangleq \top_p & [\![\bot_p]\!] &\triangleq \top_p \\
[\![\text{unit}]\!] &\triangleq \text{unit}
\end{aligned}
$$

Fig. 6. Translational semantics for expressions and types

5.3 Translation Soundness

To prove the translation soundness of Gator, we need to first define Hatchling and our translation from Gator to Hatchling. We will show that a well-typed Gator program must translate to a well-typed Hatchling program.

Primitives in Gator can be translated to a type in the target language. For notation convenience we name primitives such that $\top_p$ in Gator translates to $\top_p$ in Hatchling.

We define the syntax of Hatchling to be identical to Gator syntax except for τ, which is instead written as $\tau ::= \text{unit} \mid \top_p$, and without as! or in expressions. In other words, Hatchling is simply Gator with custom type labels and associated operations erased. In the formalism of Hatchling, abstraction over operation implementation is done using operation context Ξ that maps an operator name to its output type.

When Hatchling is parameterized to be a simple core of GLSL, some top types we might see are the `float` and `vec3` types. Translation from Gator would consist of erasing custom geometry types, such as `cart3<model>.point`, to their associated top type; in this case `vec3`.

To translate Gator's externally-defined functions (which may be overloaded on types not part of Hatchling), we invoke the context Ψ. Ψ maps the tuple of a function name, input expressions, and input types to an expression in the target language. For example, we might map Gator's definition of subtraction between points to be a GLSL subtraction between two `vec3`s. The resulting function names must each be unique and preserve the translation of Gator's primitive types. A well formed function translation context Ψ would necessarily map functions to expressions of the correct return type, as constrained by Φ under translation.

We reuse the judgment P as a mechanism to resolve in expressions, applying the function result of evaluating the judgment to e. This must produce a result of the correct translated type for a well-formed judgment P.

The typing rule for operation expressions is:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, \vdash e_2 : \tau_2 \qquad \Xi(o) = \tau_3}{\Gamma \vdash o(e_1, e_2) : \tau_3}$$

We emphasize this rule as being similar to Gator's rules for operations, but with a "translated" context using only Hatchling (i.e. primitive) types. We also note that Ξ does not take in types as arguments, thus Hatchling does not support overloaded functions.

We define translational semantics from type-annotated Gator to Hatchling in Figure 6. The typing contexts Γ and Φ are translated by replacing every τ in their range with $[\![\tau]\!]$.

638 Using structural induction over expressions in Gator, we are now able to show that

639 THEOREM 1 (TRANSLATIONAL SOUNDNESS). *For all* Γ, e, *and* τ, *if* $\Gamma \vdash e : \tau$ *then* $[\![\Gamma]\!] \vdash [\![e]\!]_\Gamma : [\![\tau]\!]$.

640 That is to say, if Gator code type checks, then Hatchling code type checks. Since Hatchling is
641 constrained to be sound, Gator must be sound. A sketch of this proof is included in the provided
642 supplementary materials.
643

644 ## 6 IMPLEMENTATION

645 We implemented Gator in a compiler that statically checks user-defined geometric type systems as
646 described in Section 3 and automatically generates transformation code as described in Section 4.
647 The compiler consists of 2,800 lines of OCaml. It can emit either GLSL or TypeScript source code,
648 to target either GPU shaders or CPU-side setup code, respectively.
649 The rest of this section describes how the full Gator language implementation extends the core
650 language features to enable real-world graphics programming. We demonstrate these features in
651 detail in a series of case studies in Section 7.
652

653 ### 6.1 Practical Features

654 *Types.* While Gator is designed around geometry types, writing realistic code requires a more
655 complete language design. Aside from the primitive types `bool`, `int`, `float`, and `string`, Gator
656 supports fixed-length array types, such as `float[3]`, and type aliases.
657 New types may be declared as a *subtype* of an existing type. For instance, we can add support for
658 the GLSL-style `vec3`:
659

```
type vec3 is float[3];
```

660 Through creating a custom type alias, we can, for example, provide support for a subtype of
661 `float[3]`, the GLSL `vec3`. While the built-in `float[3]` type does not support vector addition, we
662 will be able to write $x + y$ for `vec3`s x, y as in GLSL.
663 To allow literal values to interact intuitively with custom types, literals in Gator have special
664 types. For example, the number 42 is of type `%int`. Gator introduces a typing rule where each literal
665 type $\%p$ is a subtype of every subtype of p. In other words, the literal type $\%p$ is the bottom type
666 for the type hierarchy with top type p. We summarize these ideas in this example:
667

```
type vec3 is float[3];
vec3 s1 = [4.2, 4.2, 4.2];  // Legal
float[3] x = s1;            // Legal
vec3 s2 = x;                // ERROR: float[3] is not a vec3
```

671 This behavior of literal values allows us to capture the Gator-style intuition that a given vector can
672 either be a geometric point or just a raw GLSL `vec3`, but this information is not known until the
673 data is assigned to a variable.
674

675 *Type Inference.* Gator supports local type inference using the `auto` keyword:

```
cart3<model>.point fragPos = ...;
// worldPos will have type cart3<world>.point
auto worldPos = fragPos in world;
```

679
680 *External Functions.* Functions and variables defined externally in the Gator target can be written
681 using the `declare` keyword.

```
declare vec3 normalize(vec3 v);
```

683 All arithmetic operations in Gator are functions which can be declared and overloaded. Gator has
684 no built-in functions. Requiring this declaration allows us to include GLSL-style infix addition of
685 vectors without violating coordinate systems restrictions:
686

```
declare vec3 +(vec3 v1, vec3 v2);
```

Addition is then valid for values of type `vec3`:

```
vec3 x = [0., 1., 2.];
vec3 result = x + x; // Legal
```

But emits an error when applied to two points, as desired, since they are not subtypes of vec3 and so there is no valid function overload:

```
cartesian<model>.point fragPos = [0., 1., 2.];
auto result = fragPos + fragPos; // ERROR: No addition defined for points
```

Import System. To support using custom Gator libraries in a readable way, we built a simple import system in Gator. Files can be imported with the keyword `using` followed by the name of the file:

```
using "../glsl_defs.lgl";
```

Unsafe Casting. As an escape hatch from strict vector typing, Gator provides an unsound cast expression written with `as!`:

```
vec3 position = fragPos as! vec3;
```

Casts must preserve the primitive representation; we could not, for instance, cast a variable with type `float[2]` to `float[3]`. Unsafe casts syntactically resemble `in` expressions but are unsound and carry no run-time cost. These casts both allow for unsafe transformations for defining a function that is externally "known" to be safe, and for allowing the user to forgo Gator's type system and work directly with GLSL-like semantics, as seen in the example above.

6.2 Standard Library

Per Section 5, Gator does not include any built-in functions or operations. Our implementation does provide array indexing as a built-in function to help simplify definitions, but otherwise matches requires that operations such as + be explicitly declared.

We implement a standard library provides access to common GLSL operations. This library consists of GLSL function declarations, scheme declarations for Cartesian and Homogeneous coordinates, and basic transformation functions such as `homify` and `reduce`. Relevant GLSL functions are declared to work on GLSL types, such as the addition operation operation in section 6:

```
declare vec3 +(vec3 x, vec3 y);
```

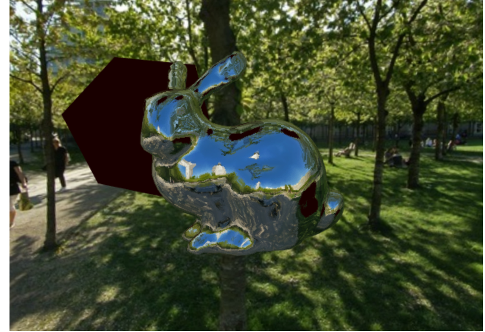
We build schemes in much the same way as introduced in Section 3.2, as with the sketch of the `cart3` scheme:

```
with frame(3) r:
coordinate cart3 : geometry {
  object vector is float[3];
  vector +(vector v1, vector v2) {
    return [v1[0] + v2[0], v1[1] + v2[1], v1[2] + v2[2]];
  }
}
```

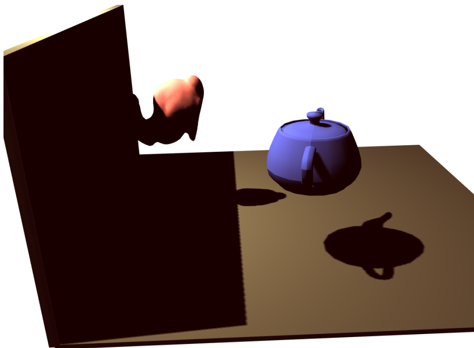
Finally, we include `homify` and `reduce` transform between homogeneous and cartesian coordinates as discussed in Section 3.3:

```
hom<model>.point homify(cart3<model>.point p) {
  return [p[0], p[1], p[2], 1.];
}
cart3<model>.point reduce(hom<model>.point p) {
return [p[0], p[1], p[2]];
}
```

(a) Texture.



(b) Reflection.



(c) Shadow map.

(d) Microfacet.

Fig. 7. Example outputs from four renderers used in our case studies.

We use this same library when implementing each shader for the case study.

7 GATOR IN PRACTICE

This section explores how Gator can help programmers avoid geometry bugs using a series of case studies. We use the Gator compiler to implement OpenGL-based renderers that demonstrate a variety of common visual effects, and we compare against implementations in plain GLSL. We report qualitatively on how Gator's type system influences the expression of the rendering code (Section 7.1 and quantitatively on the performance impact of Gator's `in` expressions (Section 7.2).

7.1 Case Studies

To qualitatively study Gator's safety and expressiveness, we used it to implement 8 renderers based on the OpenGL API in its browser-based incarnation, WebGL [Jackson and Gilbert 2015]. To the

best of our knowledge, there is no standard benchmark suite for evaluating the expressiveness and performance of graphics shader programs. Instead, we assemble implementations of a range of common rendering effects:

- *Phong:* The lighting model introduced in Section 2.
- *Reflection:* Use two-pass rendering to render an object that reflects its surroundings.
- *Shadow map:* Simulate shadows for moving objects by computing a projection.
- *Microfacet:* Texture model for simulating roughness on a surface.
- *Texture:* Use OpenGL's texture mapping facility to draw an image on the surface of an object.
- *Spotlight:* Phong lighting restricted to a spotlight circle.
- *Fog:* Lighting model with integration to simulate distortion from fog.
- *Bump map:* Texture model for simulating bumps on surfaces.

Each renderer consists of both CPU-side "host" code and several GPU-side shader programs. Figure 7 depicts the output of a selection of these renderers.

The rest of this section reports on salient findings from the case studies and compares them to standard implementations in GLSL and TypeScript. For the sake of space, we highlight the most distinct cases where Gator helped clarify geometric properties and prevent geometry bugs that would not be caught by plain GLSL. The complete code of both the Gator and reference GLSL implementations can be found online.[6]

Reflection. Our reflection case study, shown in Figure 7b, renders an object that reflects the dynamic scene around it, creating a "mirrored" appearance. The surrounding scene includes a static background texture, known as a *skybox*, and several non-reflective floating objects to demonstrate how the reflected scene changes dynamically.

Rendering a reflection effect requires several passes through the graphics pipeline. The idea is to first render the scene that the mirror-like object will reflect, and then render the scene again with that resulting image "painted" onto the surface of the object. There are three main phases: (1) Render the non-reflective objects from the perspective of the reflective object. This requires six passes, one for each direction in 3-space. (2) Render the reflection using the generated cube as a texture reference. (3) Finally, render all other objects from the perspective of the camera.

Reflection: Inverse Transformation. For the second step, we refer to a cubemap—a special GLSL texture with six sides—to refer to the six directions of the scene. To calculate the angle of reflection, we need to reason about the interactions of the light rays in view space *as they map onto our model space.* Specifically, calculating the reflection amounts to the following operations, where `V` is the current vertex's position and `N` is the current normal vector, which must both be in the view frame:

```
uniform samplerCube<alphaColor> uSkybox;
...
void main() {
  ...
  cart3<view>.vector R = -reflect(V, N);
  auto gl_FragColor = textureCube(uSkybox, R in model);
}
```

The key feature to note here is the transformation `R in model`, which accomplishes our goal of returning the light calculation to the object's perspective (the model frame). This transformation requires that we map backwards through the world frame, a transformation which requires the inverse of the `model`→`world` matrix and the `world`→`view` matrix multiplied together. This interaction produces a unique feature in Gator's type system, where we need to both have a forward

[6]URL omitted for anonymous review

transformation and its inverse. The shader declares the matrices as follows, with the inversion being done preemptively on the CPU:

```
canon uniform hom<world>.transformation<view> uView;
canon uniform hom<model>.transformation<world> uModel;
canon uniform cart3<view>.transformation<model> uInverseViewTransform;
```

The inverse view transform uses a Cartesian (`cart3`) matrix because we intend only to use it for the vector `R`, which ignores the translation component of the affine transformation. The inverse transformation is what permits us to write `R in model`, while the forward transformations must be uniquely given to actually send our position and normal to the view frame (as noted before):

```
varying cart3<model>.point vPosition;
varying cart3<model>.normal vNormal;
void main()
  auto N = normalize(vNormal in view);
  auto V = -(vPosition in view);
  ...
}
```

Reflection: Normal Transformation. Additionally, we need to reason about the correct transformation of the normal *with translation* (that is, when moving the object in space), which means that we need the inverse transpose matrix, which provides a distinct path between the model and view frames. The use of the inverse transpose of the model-view matrix is perhaps unexpected; it arises specifically for a geometry normal from a convenient algebraic result.

In GLSL, it is easy to mistakenly transform the normal as if it were an ordinary direction:

```
varying vec3 vNormal;
void main()
  auto N = normalize(vec3(uView * uModel * vec4(vNormal, 0.)));
}
```

This code is wrong because `uModel * vec4(vNormal, 0.)` does not apply the translation component of the `uModel` transformation. To prevent this kind of bug, the Gator standard library defines the `normal` type, which is a subtype of `vector`. A new `normalTransformation` type can only operate on normals. Using these types, a simple `in` transformation suffices:

```
canon uniform cart3<model>.normalTransformation<view> uNormalMatrix;
varying cart3<model>.normal vNormal;
void main()
  auto N = normalize(vNormal in view);  // uNormalMatrix * vNormal
}
```

The compiler uses the normal version of the transformation, correctly applying the translation component.

Shadow Map: Light Space. Shadow mapping is a technique to simulate the shadows cast by 3D objects when illuminated by a point light source. Our case study, shown in Figure 7c, renders several objects that cast shadows on each other and a single “floor” surface. The non-shadow coloring is simulated through Phong lighting as previously discussed.

As with the reflection renderer, to calculate shadows in a scene, we require several passes through the graphics pipeline. The first pass renders the scene from the perspective of the *light* and calculates the whether a given pixel is obscured by another. The second pass uses this information to draw shadows; a given pixel is lit only if it is not obscured from the light.

The first pass does all geometric operations in the vertex shader to render the scene from the light’s perspective. This is easy to get wrong in GLSL by defaulting to the usual transformation chain:

```
void main() {
  // The usual transformation chain here is wrong!
  // We should instead be using uLightProjective and uLightView
  vec4 gl_Position = uProjective * uView * uModel * vec4(aPosition, 1.);
}
```

This incorrect transformation chain will lead to shadows in strange places and hard-to-debug effects.

In Gator, on the other hand, the work is done when typing the matrices themselves. From there, the transformation to light space is both documented and correct by construction:

```
attribute cart3<model>.point aPosition;
canon uniform hom<model>.transformation<world> uModel;
canon uniform hom<world>.transformation<light> uLightView;
canon uniform hom<light>.transformation<lightProjective> uLightProjection;

void main() {
  auto gl_Position = aPosition in hom<lightProjective>;
  // ...
}
```

We use the depth information in the final pass in the form of `uTexture`. To look up where the shadow should be placed, we must lookup the position of the current pixel in the light's projective space (which is where the position was represented in the previous rendering). In GLSL, we require the following hard-to-read code:

```
float texelSize = 1. / 1024.;
float texelDepth = texture2D(uTexture,
  vec2(uLightProjective * uLightView * uModel * vec4(vPosition, 1.))) + texelSize));
```

Using the correct transformations is difficult and hard to be sure if the correct transformation chain was used once again. In Gator, on the other hand, this is straightforward:

```
float texelSize = 1. / 1024.;
float texelDepth = texture2D(uTexture, vec2(vPosition in lightProjective) + texelSize));
```

Microfacet: Custom Canonical Functions. Anisotropic microfacet shading creates an illusion of roughness and bumpiness on a 3D modeled surface using information from the normal map of that surface. Modeling this correctly, however, requires an unusual technique: building a local reference frame from the perspective of the normal vector called the local normal frame.

Converting to the local normal frame of a given normal consists of a function call with the appropriate normal vector.

```
vec3 proj_normalframe(vec3 m, vec3 n) { ... }
vec3 geom_normal;
vec3 result = proj_normalframe(viewDir, geom_normal);
```

However, as with other conversions between spaces, writing this kind of code in GLSL can involve multiple nonobvious steps. If the normal and target direction are in different spaces, the GLSL code must look like this:

```
vec3 result = proj_normalframe(vec3(uView * uModel * vec4(modelDir, 1.)), geom_normal);
```

In Gator, we instead declare `proj_normalframe` with the appropriate types and a canonical tag, noting that the normal itself is a canonical part of the transformation:

```
frame normalframe has dimension 3;
canon cart3<normalframe>.direction proj_normalframe(
  cart3<view>.direction m, canon cart3<view>.normal n) { ... }
```

932 We then declare the normal `geom_normal` with the appropriate type, and the transformation type
933 becomes straightforward:

```
canon cart3<view>.normal geom_normal;
auto result = modelDir in normalframe;
```

Textures: Parameterized Types. A *texture* is an image that a renderer maps onto the surface of a 3D object, creating the illusion that the object has a "textured" surface. Our texture case study renders a face mesh with a single texture (shown in Figure 7a). While this example does not provide any geometry insight, we highlight the study to show the broad utility of the types introduced by Gator for a graphics context. GLSL represents a texture using a `sampler2D` value, which acts as a pointer to the requested image, which is typically an input to a shader:

```
uniform sampler2D uTexture;
```

Textures are mapped to the image using the object's current texture coordinate:

```
varying vec2 vTexCoord;
```

Whereas textures themselves are typically constant (using the `uniform` keyword), a texture coordinate like `vTexCoord` differs for each vertex in a mesh (as the `varying` keyword indicates). To sample a color from a texture at a specific location, a fragment shader must use the GLSL `texture2D` function:

```
vec4 gl_FragColor = texture2D(uTexture, vTexCoord);
```

The result type of `texture2D` in GLSL is `vec4`: while textures typically contain colors (consisting of red, green, blue, and alpha channels), renderers can also use them to store other data such as shadow maps or even points in a coordinate system.

In Gator and its GLSL standard library, `sampler2D` is a polymorphic type that indicates the values it contains:

```
with float[4] T:
declare type sampler2D;
with float[4] T:
declare T texture(sampler2D<T> tex, vec2 uv);
```

For this renderer, the texture contains `alphaColor` values, which represent color values that can be used as `gl_FragColor`. The fragment shader is nearly identical to GLSL but with more specific types:

```
uniform sampler2D<alphaColor> uTexture;
varying vec2 vTexCoord;
void main() {
  alphaColor gl_FragColor = texture2D(uTexture, vTexCoord);
}
```

With this code, we guarantee that the texture represented by `uTexture` will produce a color which can be directly used by `gl_FragColor`. We therefore both provide documentation and prevent errors with trying to use the resulting vector as, say, a point for later calculations.

7.2 Performance

While Gator is chiefly an "overhead-free" wrapper that expresses the same semantics as an underlying language, there is one exception where Gator code can differ from plain GLSL: its automatic transformation insertion using `in` expressions (Section 4).

The Gator implementation compiles `in` expressions to a chain of transformation operations that may be slower than the equivalent in a hand-written GLSL shader. In particular, hand-written GLSL code can store and reuse transformation results or composed matrices, while the Gator compiler does not currently attempt to do so. The Gator compiler also generates function wrappers

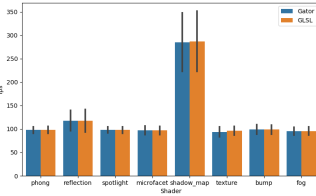


Fig. 8. The mean frames per second (fps) for each shader for both the baseline (GLSL) and Gator code. Error bars show the standard deviation.

Shader	Gator Mean	Gator S.E.	GLSL Mean	GLSL S.E.	p-value Wilcoxon	p-value TOST
phong	97.84	0.22	97.67	0.21	0.187	0.003*
texture	95.82	0.27	93.75	0.31	<0.001*	0.996
reflect	117.8	0.72	117.7	0.65	0.638	0.188
shadow	287.0	1.91	285.1	1.85	0.365	0.636
bump	98.60	0.29	99.07	0.29	0.063	0.098
microfacet	96.71	0.27	96.91	0.28	0.640	0.020*
fog	95.74	0.26	95.41	0.25	0.119	0.033*
spotlight	97.83	0.21	98.07	0.20	0.299	0.005*

Table 1. Mean and standard error of the frame rate for the Gator and GLSL (baseline) implementation of each benchmark. We also give the p-value for a Wilcoxon sign rank test and two one-sided t-test (TOST) equivalence test that checks whether the means are within 1 fps, where * denotes statistical significance ($p < 0.05$).

to enable its overloading. While both patterns should be amenable to cleanup by standard compiler optimizations, this section measures the performance impact by comparing Gator implementations of renderers from our case study to hand-optimized GLSL implementations.

7.2.1 Experimental Setup. We perform experiments on Windows 10 version 1903 with an Intel i7-8700K CPU, NVIDIA GeForce GTX 1070, 16 GB RAM, and Chrome 81.0.4044.138. We run 60 testing rounds, each of which executes the benchmarks in a randomly shuffled order. In each round of testing, we execute each program for 20 seconds while recording the time to render each frame. We report the mean and standard deviation of the frame rate across all rounds.

7.2.2 Performance Results. Figure 8 shows the average frames per second (fps) for the GLSL and Gator versions of each renderer, and Table 1 shows mean and standard deviation of each frame rate. The frame rates for the two versions are generally very similar—the means are all within one standard deviation. Several benchmarks have frame rates around 100 fps because they render the same number of objects and the bulk of the cost comes from scene setup. We used around 100 objects for all scenes except reflection and shadow to reduce natural variation and focus on measuring the cost of the shaders.

Table 1 shows the results of Wilcoxon signed-rank statistical tests that detect differences in the mean frame rates. At an $\alpha = 0.05$ significance level, we find a statistically significant difference only for texture. However, a difference of means test cannot *confirm* that a difference does *not* exist. For that, we also use we use the two one-sided t-test (TOST) procedure [Schuirmann 2005], which yields statistical significance ($p < \alpha$) when the difference in means is within a threshold. We use a threshold of 1 fps. The test rejects the null hypothesis—concluding, with high confidence, that the means are similar—for the phong, microfacet, fog, and spotlight shaders.

The anomaly is `texture`, where our test concludes that a small (2 fps) performance difference does exist, although the differences are still within one standard deviation. Our best guess as to the reason is due to a result of the boilerplate functions inserted by Gator, some of which be optimized away with more work.

8 RELATED WORK

SafeGI [Ou and Pellacini 2010] introduces a type system as a C/C++ library for geometric objects parameterized on reference frame labels not unlike Gator's geometry types. The types introduced by SafeGI do not include information about the coordinate scheme, and so also requrie abstracting the notion of transformations to a map type which must be applied through a layer of abstraction. Additionally, SafeGI does not attempt to introduce automatic transformations like Gator's `in` expressions nor attempt to study the result of applying these types to real code.

The dominant mainstream graphics shader languages are OpenGL's GLSL [The Khronos Group Inc. [n. d.]] and Direct3D's HLSL [Microsoft 2008]. Research on graphics-oriented languages for manipulating vectors dates at least to Hanrahan and Lawson's original *shading language* [Hanrahan and Lawson 1990]. Recent research on improving these shading languages has focused on modularity and interactions between pipeline stages: Spark [Foley and Hanrahan 2011] encourages modular composition of shaders; Spire [He et al. 2016] facilitates rapid experimentation with implementation choices; and Braid [Sampson et al. 2017] uses multi-stage programming to manage interactions between shaders. These languages do not address vector-space bugs. Gator's type system and transformation expressions are orthogonal and could apply to any of these underlying languages.

Scenic [Fremont et al. 2019] introduces semantics to reason about relative object positions and λCAD [Nandi et al. 2018] introduces a small functional language for writing affine transformations, although neither seem to have a type system for checking the coordinate systems they've defined. Practitioners have noticed that vector-space bugs are tricky to solve and have proposed using a naming convention to rule them out [Sylvan 2017]. A 2017 enumeration of programming problems in graphics [Sampson 2017] identifies the problem with latent vector spaces and suggests that a novel type system may be a solution. Gator can be seen as a realization of this proposal.

Gator's type system works as an overlay for a simpler, underlying type system that only enforces dimensional restrictions. This pattern resembles prior work on type qualifiers [Foster et al. 1999], dimension types [Kennedy 1994], and type systems for tracking physical units [Kennedy 1997]. Canonical transformations in Gator are similar in feel to Haskell's type class polymorphic functions, where Gator's `space` type can be defined as a type class and the `in` keyword behave similarly to Haskell lookup calls. Additionally, Gator's notion of automatic transformations is a specialized use type coercion, similar to structures introduces in the C# and C++ languages. What is particular about Gator's automatic type coercion is the notion of path independence discussed in Section 4, along with a definition of uniqueness and bijectivity of canonical transformations. Together, these requirements allow automation of coordinate system transformations that would not be allowed in other, similar systems.

9 CONCLUSION

Gator attacks a main impediment to graphics programming that makes it hard to learn and makes rendering software hard to maintain. Geometry bugs are extremely hard to catch dynamically, so Gator shows how to bake them into a type system and how a compiler can declaratively generate "correct by construction" geometric code. We see Gator as a foundation for future work that brings programming languages insights to graphics software, such as formalizing the semantics of geometric systems and providing abstractions over multi-stage GPU pipelines.

Geometry bugs are not just about graphics, however. Similar bugs arise in fields fields ranging from robotics to scientific computing. In Gator, users can write libraries to encode domain-specific forms of geometry: affine, hyperbolic, or elliptic geometry, for example. We hope to expand Gator's standard library as we apply it to an expanding set of domains.

1079

REFERENCES

1080 Tim Foley and Pat Hanrahan. 2011. Spark: Modular, Composable Shaders for Graphics Hardware. In *SIGGRAPH*.

1081 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *ACM Conference on*
1082 *Programming Language Design and Implementation (PLDI)*.

1083 Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2019. Functional Programming for Compiling and Decompiling Computer-Aided Design. In *ACM Conference on*
1084 *Programming Language Design and Implementation (PLDI)*.

1085 Pat Hanrahan and Jim Lawson. 1990. A Language for Shading and Lighting Calculations. In *SIGGRAPH*.

1086 Yong He, Tim Foley, and Kayvon Fatahalian. 2016. A System for Rapid Exploration of Shader Optimization Choices. In
1087 *SIGGRAPH*.

1088 Dean Jackson and Jeff Gilbert. 2015. WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/1.0/.

1089 Andrew J. Kennedy. 1994. Dimension Types. In *European Symposium on Programming (ESOP)*.

Andrew J. Kennedy. 1997. Relational Parametricity and Units of Measure. In *ACM SIGPLAN–SIGACT Symposium on Principles*
1090 *of Programming Languages (POPL)*.

1091 Microsoft. 2008. Direct3D. https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx.

1092 K. Murota. 1987. Homotopy Base of an Acyclic Graph: a Combinatorial Analysis of Commutative Diagrams by Means of
1093 Preordered Matroid. *Discrete Applied Mathematics* 17, 1–2 (May 1987), 135–155.

1094 Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-Aided Design. In *ACM SIGPLAN International Conference on*
1095 *Functional Programming (ICFP)*.

1096 Jiawei Ou and Fabio Pellacini. 2010. SafeGI: Type Checking to Improve Correctness in Rendering System Implementation.

1097 Bui Tuong Phong. 1975. Illumination for Computer Generated Pictures. *Commun. ACM* 18, 6 (June 1975), 311–317.

1098 Adrian Sampson. 2017. Let's Fix OpenGL. In *Summit on Advances in Programming Languages (SNAPL)*.

1099 Adrian Sampson, Kathryn S McKinley, and Todd Mytkowicz. 2017. Static Stages for Heterogeneous Programming. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

1100 Donald J. Schuirmann. 2005. A comparison of the Two One-Sided Tests Procedure and the Power Approach for assessing
1101 the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics* 15 (2005), 657–680.

1102 Mark Segal and Kurt Akeley. 2017. *The OpenGL 4.5 Graphics System: A Specification*. https://www.opengl.org/registry/doc/
1103 glspec45.core.pdf.

1104 Sebastian Sylvan. 2017. Naming Convention for Matrix Math. https://www.sebastiansylvan.com/post/matrix_naming_convention/.

1105 The Khronos Group Inc. [n. d.]. *The OpenGL ES Shading Language* (1.0 ed.). The Khronos Group Inc.

1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127