EG UK Theory and Practice of Computer Graphics (2008)
Ik Soo Lim, Wen Tang (Editors)

Space-free shader programming: Automatic space inference and optimization for real-time shaders

Calle Lejdfors and Lennart Ohlsson

Department of Computer Science, Lund University, Sweden

Abstract

The graphics processing units (GPUs) used in todays personal computers can be programmed to compute the visual appearance of three-dimensional objects in real time. Such programs are called shaders and are written in high-level domain-specific languages that can produce very efficient programs. However, to exploit this efficiency, the programmer must explicitly express space transformations necessary to implement a computation. Unfortunately, this makes programming GPUs more error prone and reduces portability of the shader programs.
In this paper we show that these explicit transformations can be removed without sacrificing performance. Instead we can automatically infer the set of transformations necessary to implement the shader in the same way as an experienced programmer would. This enables shaders to be written in a cleaner, more portable, manner and to be more readily reused. Furthermore, errors resulting from incorrect transformation usage or space assumptions are eliminated. In the paper we present an inferencing algorithm as well as a prototype space-free shading language implemented as an embedded language in Python.

Keywords: Shading language, graphics processing unit (GPU), optimization, inference, embedded language

1. Introduction

The *graphics processing unit* (GPU) available on modern graphics cards makes it possible to execute user-defined *shader programs* in real-time. Using these *vertex* and *pixel shaders* it is possible to control the position and orientation, as well as the per-pixel color of rendered objects. Programmable shading hardware enables many highly realistic graphical effects to be implemented.

Shaders typically makes heavy use of vector calculations to compute the appearance of a mesh. When implement them using current GPU languages [MGAK03, Gra03, Ros04], it is necessary to perform several explicit coordinate or *space* transformations to a common space in which the appearance of the mesh may be computed. By using that, in general, there are more projected on-screen pixels than there are vertices in a mesh, it is possible to get very efficient shaders by performing the majority of these transformations per-vertex. By choosing an appropriate space, it is even possible to avoid some transformations completely.

The need to perform explicit transformations put an additional burden on the shader programmer. Moreover, the choice of space is dependent both on the application and the model that is to be rendered. This can result in a shader performing sub-optimally, or even incorrectly, when used in another application or for another model. This type of implicit assumptions together with the high performance-sensitivity of shaders, results in very tight coupling between shader and application. This makes reusing shaders in other applications, or even for other models, difficult.

In this paper we show that explicit transforms are not required. Instead, it is possible to automatically infer the same choice of space transforms that an experienced programmer would make. The basis for our *space-free* shader programming approach is a type system that is richer than those provided by other GPU languages and an inferencing algorithm based on this type system. We have developed a prototype compiler that implements space-free shader programming. In this language shaders are written similar to off-line shaders, as though every vector and point is already in an appropriate space. By removing explicit transforms in this way we increase portability and reuse, since the code no longer

```
varying vec3 L;
varying vec3 N;

void vertex() {
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
    N = normalize(gl_NormalMatrix*gl_Normal);
    L = (gl_LightSource[0].position −
                                gl_ModelViewMatrix*gl_Vertex).xyz;
}

void pixel() {
    vec3 Nn = normalize(N);
    vec3 Ln = normalize(L);
    float diffuse = max(dot(Nn, Ln), 0.0);
    gl_FragColor = diffuse*vec4(1,1,1,1);
}
```

Listing 1: *A Lambertian (diffuse) shading algorithm implemented in GLSL using view space.*

```
varying(unitvector, L)
varying(direction, N)

def vertex():
    gl_Position = gl_Vertex
    N = gl_Normal
    L = gl_LightSource[0].position − gl_Vertex

def pixel():
    diffuse = max(dot(N, L), 0.0)
    gl_FragColor = diffuse*rgba(1,1,1,1)
```

Listing 2: *A Lambertian shading algorithm in our prototype language. Here, no explicit vector transforms are required. Also, vector re-normalization in the pixel shader is automatically handled by our compiler.*

contains application-specific assumptions. Also, consistent usage of vector quantities is guaranteed.

As an example, consider the Lambertian (diffuse) lighting model where the light reflected from a point is proportional to cosine of the angle between the surface normal and the direction of the light source, i.e. the dotproduct of these two unit vectors. Listing 1 shows this shader written in GLSL and Listing 2 shows the same shader implemented in our space free language. In both versions the vertex function is executed once for every vertex of the mesh, and the pixel function once for every projected on-screen pixel. Data that should be interpolated across the primitive and used as input to the pixel program is declared as varying. The vertex program writes values to the varying parameters (surface normal N and vector to the light source L in this example) and the clip space position of the vertex (written to the dedicated variable gl_Position). The pixel program reads the interpolated result and uses it to compute the final pixel or fragment color which is written to the variable gl_FragColor.

The differences are that in the GLSL version, lighting is computed in view space and the necessary space transformations are performed explicitly by matrix multiplication. In the space-free version the choice of space and the necessary matrix multiplications are instead inferred and inserted by the compiler. Also, in the GLSL version, explicit re-normalization of unit vectors is required. In the space-free version this is automatically handled by the compiler.

In the rest of the paper we describe our prototype shading language and the implementation of our compiler.

2. Automatic space inference

Translating a space-free shader to the GPU consists of finding a set of spaces and transformations such that the generated program is both correct and gives the highest possible run-time performance. Here, correctness means that the result of the shading algorithm must be identical to the result which would have been obtained using a known, correct choice of spaces. This involves ensuring that each operation in the shader is evaluated in an appropriate space. For example, consider computing the dot-product dot(N,L) used in the Lambertian shader in Listing 1. By default, this quantity should be computed in view space, since we are interested in computing how a mesh appears to the viewer.

However, suppose we know that model space and view space are related via a unitary transform U, i.e. a transform that is both angle and length-preserving. Then, using the identity for unitary transforms

$$\mathrm{dot}(\mathsf{N}_v, \mathsf{L}_v) = \mathrm{dot}(\mathsf{U}*\mathsf{N}_m, \mathsf{U}*\mathsf{L}_m) = \mathrm{dot}(\mathsf{N}_m, \mathsf{L}_m)$$

we find that we may compute the scalar product in model space without introducing errors. Here the m and v subscripts denote the vector expressed in the view and model space frames, respectively. So, if N and L are expressed in model space, it is possible to avoid two transforms in this case.

In order to deduce which spaces are possible for a particular computation we must first, know what spaces and transforms are available, and how these transforms behave. Second, we must determine how each operation behaves under these space transformations. And, third, we must know the type of every expression in the shader to know how it should be transformed (e.g. unit vector should be transformed to unit vectors, normals to normals, and so on). Using this information, we may then deduce every possible space in which the particular shader can be implemented. By conservatively inferring the possible spaces for each operation the resulting shader can be guaranteed to be correct from a space usage perspective. Then, analyzing the cost of the different choices of transforms, we can choose the variant with the highest run-time performance.

2.1. Spaces and transforms

The spaces and transforms available are in most cases determined by the application. The application is responsible for placing and orienting the viewer in the world (thus determining the view-to-world and world-to-view transforms) and also, for placing each model in world (hence giving the model-to-world transform for each model). Some applications do not represent world space explicitly, instead preferring to represent the model-to-view transforms directly.

Furthermore, some meshes contain information about local spaces, such as *tangent space*, used in more complicated shading algorithms (see Section 4). In this case, these spaces are constructed explicitly in the shader and are available to the space inference algorithm just as any other space.

Operations can be divided into three classes, depending on how they behave under space transforms:

- space independent – does not change with changing space. Includes all scalar operations (such as sine, cosine, exp, max, min etc.) and color operations.
- space dependent – operations which require arguments to be in the same space (vector addition, subtraction, ...) but does not involve angle or length measurements.
- metric dependent – operations which require arguments to be in the same space and have the same metric (dot-product, normalization, length, etc.).

This classification lets us determine, given that an operations should be computed in a particular space, the set of possible spaces for that operation.

Space independent For a space independent operation that should be computed in a space S, the only possible space is S. For example, computing the final pixel color by multiplying a color by a floating point value

$$\mathsf{gl\_FragColor} = \mathsf{diffuse} * \mathsf{materialColor}$$

should be performed in view space. This follows by the reasoning above, that the pixel color should be computed as it appears to the viewer.

Space dependent For a *space dependent* operation that should be computed in a space S the set of possible spaces is equal to those spaces from which there exists a transform to S. For example, suppose we should compute the light vector L in view space by

$$\mathsf{L} = \mathsf{lightPos} - \mathsf{vertexPos}$$

Then we may compute the light position and vertex position in any space for which there exist a transform to view space. The resulting vector can then be transformed to view space using this transform. So, if there exist a transform from model to view space and we have the light and vertex position in model space. Then, we can compute the light vector in view space by

$$\mathsf{L} = \mathsf{modelToView} * (\mathsf{lightPos} - \mathsf{vertexPos})$$

where modelToView is the transformation matrix from view to model space. This saves us one transform.

Metric dependent In the previous section we saw an example of *metric dependence* when we argued that we could compute the diffuse term in model space rather than view space, when these spaces were related by a unitary transform. Generally, if an operation is metric dependent and should be computed in some space S then it may be computed in any space from which there exists a unitary transform to S.

It is possible to generalize metric dependence further by considering transforms which are angle but not length-preserving. For such a transform T we have

$$\mathsf{dot}(\mathsf{T} * \mathsf{v}, \mathsf{T} * \mathsf{u}) = \lambda^2 * \mathsf{dot}(\mathsf{v}, \mathsf{u})$$

for arbitrary vectors v and u and where $\lambda = \det \mathsf{T}$. However, due to lack of motivating real-world examples, we restrict ourselves to the unitary case, i.e. when $\lambda = 1$.

2.2. Typing and vector transforms

How a vector quantity is transformed is dependent on what type of vector it is and what kind of information it encodes. The type of the transformed vector must be the same as the original vector's. For example, transforming a unit vector v using a transform T must result in a new unit vector. Hence, this transform must be done by

$$\mathsf{v}' = \mathsf{T} * \mathsf{v} / \mathsf{length}(\mathsf{T} * \mathsf{v})$$

Here we can use transform properties to simplify this expression. For example, if we know that T is unitary, then $\mathsf{length}(\mathsf{T} * \mathsf{v}) = 1$ and the division can be skipped. However, if the vector v is a surface normal, then it should be transformed using the inverse transpose of T instead. This is necessary to ensure that the result is still a normal, i.e. orthogonal to the surface.

These examples illustrate the need for a fine-grained type system to be able to deduce how a vector quantity should be transformed. We use a type system which is an extensible form of the *semantic types* of McGuire et al. [MSPK06]. This type system provides types for vectors, points, unit vectors, directions, and normals, in addition to the vector types provided by GLSL. Similarly, the matrix types of GLSL are extended to include 4×4 unitary and 3×3 unitary matrices.

In addition to being used to deduce how vectors should be transformed, type information can be used to correctly interpolate values between the vertex and pixel shader. Two methods are interpolation methods are used: unit vectors are normalized first in the vertex shader, interpolated, and then re-normalized in the pixel shader. Directions, on the other hand, must not be normalized in the vertex shader, only in the pixel shader. Our system uses type information to chose the correct method depending on the type of varying data. In current shader languages, this must be handled manually by the shader programmer and it is a frequent source of errors.

2.3. Shader cost optimization

Selecting the best space choice can be done by estimating the run-time cost of every shader variant, and choosing the variant having the lowest cost. However, since every shader contains code executing per-vertex and per-pixel, the *computational frequency* [PMTH01] at which each operation of the shader executes must be taken into consideration when choosing the best variant. These frequencies are:

- Per-model or instance computations such as transform matrices and light positions which do not vary across the mesh. Computed on the CPU and then downloaded to the GPU.
- Per-vertex calculations for computing primitive interpolants and attributes such as texture coordinates, shadow map coordinates, and, depending in lighting model, light directions and tangent space transforms. Performed in the vertex processing unit of the GPU.
- Per-pixel calculations including texture accesses and possible pixel discards performed in the pixel processing unit of the GPU. This step is performed before alpha, stencil, and depth testing.

It is not possible to *a priori* determine how many times each part of a shader will be executed. However, for the majority of applications, the general rule is that the pixel shader is executed many more times than the vertex shader. This follows since there generally are many more projected on-screen pixels than vertices of a mesh. Similarly, the vertex shader is executed more often than the model computations, as model computations are performed only once for each mesh which typically consist of several thousand vertices.

This asymmetry can be described by summing the run-time cost of each individual step and representing them by a tuple (t_p, t_v, t_m) for the pixel, vertex, and model run-time cost, respectively. These tuples can then be compared using lexicographic ordering to find a least element. By using another ordering function it is possible to adapt the choice algorithm to optimize for non-standard applications, such as wire-frame renderers (where the ratio of pixels to vertices is lower).

2.4. Further optimizations

When constructing shader variants there are two important optimizations that need to be considered to correctly determine the cost of a shader. The first is that unitary transforms can be inverted without cost. Using that the GPU does not distinguish between row and column vectors we may perform the following rewrite:

$$\mathsf{T}^{-1} * \mathsf{v} = \mathsf{T}^t * \mathsf{v} = (\mathsf{v} * \mathsf{T})^t = \mathsf{v} * \mathsf{T}$$

using the fact that the inverse of a unitary transform is its transpose. This optimization is particularly important when transforming surface normals since such vectors should be transformed using the inverse transpose of the matrix used to transform the surface. Hence, for a unitary transform U we get $(\mathsf{U}^{-1})^t = (\mathsf{U}^t)^t = \mathsf{U}$.

Second, when constructing local frames in the shader, the vectors used to defined the coordinate system for the space have constant coordinates in that space. For example, if a shader constructs a local space S from the basis vectors u, v, and w. Then we may rewrite, for example:

```
dot(u, l) = l.x
```

i.e. as the first component of vector l, if the dot product is computed in the space S. This is possible since in S the coordinates for u, v, and w are $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, respectively. Also, if these vectors are used as interpolants, then we can further reduce computational requirements by not interpolating them, since they are constant in S.

3. Implementation

Our prototype, space-free shading language is implemented as an embedded language in Python, an object-oriented, interpreted, dynamically typed, high-level language. Language embedding allows us to reuse parts of the Python compiler framework to avoid writing a complete language front-end. This has enabled us to quickly experiment with different features and design choices in our implementation. For an example Lambertian shader, see Listing 2.

Just as in GLSL, our language provide a number of implicit input and output parameters (vertex position, pixel color, texture coordinates) named after their GLSL counterparts. These parameters are prefixed by gl_ and are provided either in model or view space. A non-exhaustive list of implicit parameters along with their type and space is given in Table 1.

3.1. The space inference process

Our space inference compiler is implemented as a constraint solving process. Starting at the output variables gl_Position and gl_FragColor for which the desired output spaces are known, it proceeds breadth-first back through the data-dependence graph of the program. For every operation, a space is chosen from the set of possible space (using the classification in Section 2.1) and this choice is propagated as a constraint back through the data-dependence graph. Once spaces have been chosen for every operation, the shader variant is passed to a backend code generator.

The backend inserts the necessary transforms to implement the shader in the chosen spaces and then optimizes the shader (in particular using the domain specific optimization in Section 2.4). After this a frequency analysis is performed, any computations which do not vary per-vertex or per-pixel are moved to the CPU to avoid unnecessary re-computations. Also, as part of this analysis, the shader cost at the model, vertex, and pixel frequency are estimated. Finally the GPU vertex and pixel shader programs are generated. The code

Parameter	Type	Space	Notes
gl_Vertex	Point	ModelSpace	
gl_Normal	Normal	ModelSpace	
gl_Position	Point	ClipSpace	Must write per-vertex
gl_FragColor	RGBA	ViewSpace	Must write per-pixel
gl_LightSource[*i*]			
.position	Point	ViewSpace	

Table 1: *A non-exhaustive list of implicit variables provided by our prototype language. The output variables* gl_Position *and* gl_FragColor *must, as indicated, be computed and written in the vertex and pixel shader, respectively.*

generator currently targets Python for the CPU code and GLSL for the GPU code.

3.2. Shader optimization

The space inference process is continued until the set of possible spaces has been exhaustively searched. Using the estimated shader cost computed by the backend the shader variant with the lowest run-time cost is found. Currently, the cost ranking is performed using lexicographic ordering as described in Section 2.3. The cost ranking function is passed as a parameter to the compiler allowing it to be replaced by a non-standard ranking method if required.

As it is currently implemented, the running time of our compiler is proportional to the number of possible space choices in a shader (worst case: exponential in the number of vector operations). Reducing this search time is not straightforward. Traditional methods for reducing the run-time for minimization algorithm relies on pruning or cutting down the potential search space. However, due to the potentially large performance impact of the optimizations in Section 2.4, the run-time cost of a shader is difficult to reliably estimate before the space for each operation has been chosen. These optimizations can lead to entire expression being eliminated or replaced by a simpler one. For example, if T is unitary and v is the first coordinate axis of T then

$$\text{dot}((T^{-1})^t * v, u) = \text{dot}(T * v, u) = \text{dot}((1,0,0), u) = u.x$$

In this case a dot product computations and a transform has been reduced to a single vector member access (which can be performed without run-time cost on the GPU). Compilation times for our examples ranges from sub-second to at most a few seconds and we have not found this to be a problem. Nevertheless, reducing compilation times is an interesting area for further research.

Another item of note is that space inference algorithm will never place a transform inside a loop. If a transform were to be placed inside a loop, the run-time cost of the generated shader can not be estimated since the number of loop iterations can in general not be determined. In some restricted cases, it is possible to statically determine the number of times the body of a loop will be executed. However, in non-trivial examples the number of loop iterations can not be determined at compile time. For example, the presence of flow-control commands such as break, return, or continue generally makes such analysis impossible. We have found no realistic examples where this restriction is a limitation, however.

```
uniform(sampler2D, bumpmap)
attribute(vector, ModelSpace, tangent)
varying(vector, L)
varying(vector, H)
constant(point, ViewSpace, eyePos, (0,0,0))

def vertex():
    gl_Position = gl_Vertex
    gl_TexCoord[0] = gl_MultiTexCoord0
    N = gl_Normal
    T = tangent
    B = cross(N, T)
    space(TangentSpace, unitary3(T,N,B))
    L = gl_LightSource[0].position − gl_Vertex
    V = eyePos − gl_Vertex
    H = normalize(L+V)

def pixel():
    offset = TangentSpace(tex2D(bumpmap,gl_TexCoord[0]).xyz)
    N = normalize(offset + N)
    diffuse = max(dot(N, L), 0.0)
    specular = pow(max(dot(N, H), 0.0), 8)
    gl_FragColor = diffuse∗rgba(1,0,0,1)+specular∗rgba(1,1,1,1)
```

Listing 3: *A space-free bump mapping shader. The surface normal is perturbed using a texture to give the impression of fine-scale structure such as bumps or ridges on a surface. Here, the vertex shader is used to construct a tangent frame which can be used for shading computations. Note that the space of the* offset *vector and eye position must be explicitly specified for the inference algorithm to work.*

4. Examples

The Lambertian example given so far is a very simple example of an *isotropic* lighting model, i.e. a surface that scatters light uniformly in every direction. Such lighting models can typically be efficiently implemented in model, world or view space. However, in most real-world surfaces there is an asymmetry, relative to the local surface orientation, in how light is reflected. In such *anisotropic* models, the local orientation is usually expressed by the tangent space of every point on the surface. Common examples of such models are *bump* [Bli78] and *parallax mapping* [KTI*01], that uses tangent space to compute normal and, in the case of parallax mapping, texture space offsets to give appearance of surface wrinkles or bulges. Listing 3 gives an implementation of bump mapping in our language. Note the declaration of the new space TangentSpace, constructed from the basis vec-

Shader	Performance (FPS) Space-free	Hand-written	Rel. perf.
Lambertian	1550	1580	98%
Blinn-Phong	1280	1310	97%
Bump mapping	1230	1240	99%
Refraction	917	950	96%
Parallax mapping	568	581	98%
View space bump m.	-	1100	-

Table 2: *Performance measurements of a simple synthetic scene running different shaders. The view space bump mapping is a hand written example demonstrating the performance impact of using an inappropriate space. Performance figures giving absolute performance rendered frames per second (FPS) of a synthetic scene. All measurements were run on an NVIDIA GeForce 8600 GT graphics card using a C++ rendering engine.*

tors T, N, and B. Our compiler can use this space to infer that, in this case, the best space choice is tangent space.

For performance results of some common isotropic and anisotropic shading algorithms see Table 2. These algorithms demonstrate different levels of complexity, ranging from just a few scalar products (the Lambertian and Blinn-Phong) to complex, iterative algorithms that perform ray-tracing in texture space (Parallax mapping). All measurements compare a space-free implementation written in our prototype language to an equivalent, manually optimized, version implemented in GLSL. All hand-written examples contain implicit assumptions (e.g. that the view to model space transform is unitary) and explicit vector normalization in the pixel shader. All such assumptions are eliminated in the space-free variants. In all examples, the transforms and spaces inferred by our compiler precisely matches those used in the hand-written examples.

We see that the hand-optimized consistently out-perform the space-free shader variants. However, the performance gains of hand-optimizing a shader is only in the order of a few percent. This indicates that the space inference approach is useful in practice. The exact source of the performance disparity is difficult to pin-point since the GLSL compiler does not allow the generated assembler to be inspected. However, the most likely source is that hand-writing lets the programmer exploit instruction-level SIMD parallelism available on the GPU. Presumably, the structure of the code generated by our backend is less amenable to parallelization by the GLSL compiler.

5. Discussion

We have presented a novel domain-specific language for programming GPUs that enables shader programs to be written without explicit space transforms. This enables shading algorithms to be implemented in a more general and portable manner while allowing optimization to be performed on a per-application or even per-model basis. Consequently, space-free shaders can be reused in other applications without changes. The performance of the generated GPU code is very close to that of hand-optimized versions which indicates that the approach given in this paper is useful in practice.

Much previous work has been devoted to translating shaders for off-line use, typically written in the RenderMan shading language [HL90], to real-time graphics hardware: reducing memory-accesses [OKS03], automatic multi-pass shader factorization [POAU00], and semi-automatic vertex/pixel factorization of shaders [PMTH01]. Our paper represent a previously unexplored avenue that could be used to further improve on these results.

References

[Bli78] BLINN J. F.: Simulation of wrinkled surfaces. In *Computer Graphics (Proceedings of SIGGRAPH)* (1978), vol. 5, pp. 286–292.

[Gra03] GRAY K.: *DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH)* (1990), vol. 17, pp. 289–298.

[KTI*01] KANEKO T., TAKAHEI T., INAMI M., KAWAKAMI N., YANAGIDA Y., MAEDA T., TACHI S.: Detailed shape representation with parallax mapping. In *Proceedings of the ICAT2001* (2001), vol. 12, pp. 205–208.

[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. Graph. 22*, 3 (2003), 896–907.

[MSPK06] MCGUIRE M., STATHIS G., PFISTER H., KRISHNAMURTHI S.: Abstract shade trees. In *I3D '06* (New York, NY, USA, 2006), ACM, pp. 79–86.

[OKS03] OLANO M., KUEHNE B., SIMMONS M.: Automatic shader level of detail. In *Graphics hardware* (2003), Eurographics Association, pp. 7–14.

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (Proceedings of SIGGRAPH)* (2001), vol. 28, pp. 159–170.

[POAU00] PEERCY M. S., OLANO M., AIREY J., UNGAR P. J.: Interactive multi-pass programmable shading. In *Computer Graphics (Proceedings of SIGGRAPH)* (2000), vol. 27, pp. 425–432.

[Ros04] ROST R. J.: *OpenGL Shading Language*. Addison-Wesley Professional, February 2004.