Deep Compositing Using Lie Algebras

TOM DUFF
Pixar Animation Studios

Deep compositing is an important practical tool in creating digital imagery, but there has been little theoretical analysis of the underlying mathematical operators. Motivated by finding a simple formulation of the merging operation on *OpenEXR*-style deep images, we show that the Porter-Duff **over** function is the operator of a Lie group. In its corresponding Lie algebra, the splitting and mixing functions that *OpenEXR* deep merging requires have a particularly simple form. Working in the Lie algebra, we present a novel, simple proof of the uniqueness of the mixing function.

The Lie group structure has many more applications, including new, correct resampling algorithms for volumetric images with alpha channels, and a deep image compression technique that outperforms that of *OpenEXR*.

CCS Concepts: • **Computing methodologies** → **Antialiasing**; *Visibility*; *Image processing;*

Additional Key Words and Phrases: Compositing, deep compositing, alpha, atmospheric effects, Lie group, Lie algebra, exponential map

ACM Reference Format:

Tom Duff. 2017. Deep compositing using lie algebras. ACM Trans. Graph. 36, 3, Article 26 (March 2017), 12 pages.
DOI: http://dx.doi.org/10.1145/3023386

1. INTRODUCTION

Porter-Duff [1984] compositing is the standard way to combine images with masks (alpha channels). Its primary use is to combine elements that have been rendered or photographically captured separately into a composite image. To do that, the depth ordering of the elements must be known a priori. In particular, the method has no intrinsic way to deal with elements whose depth order may vary from pixel to pixel. Even more problematic are volumetric elements like clouds whose pixels may occupy extended depth regions. Combining two such volumes requires computing new pixel values for regions that overlap in depth, something that the Porter-Duff **over** operator cannot do directly.

Note: Throughout, this article will refer to a color with an associated opacity value as a *pixel value* rather than a *color* to emphasize the opacity channel.

Authors' address: T. Duff, Pixar Animation Studios, 1200 Park Avenue, Emeryville, CA 94608; email: td@pixar.com.

0730-0301/2017/03-ART26 $15.00
DOI: http://dx.doi.org/10.1145/3023386

Deep images extend Porter-Duff compositing by storing multiple values at each pixel with depth information to determine compositing order. The *OpenEXR* deep image representation stores, for each pixel, a list of nonoverlapping depth intervals, each containing a single pixel value [Kainz 2013]. An interval in a deep pixel can represent contributions either by hard surfaces if the interval has zero extent or by volumetric objects when the interval has nonzero extent. Displaying a single deep image requires compositing all the values in each deep pixel in depth order. When two deep images are merged, splitting and mixing of these intervals are necessary, as described in the following paragraphs.

Merging the pixels of two deep images requires that corresponding pixels of the two images be reconciled so that all overlapping intervals are identical in extent. This is done by using the endpoints of each interval of an overlapping pair to divide the other interval, as in Figures 2(a) and 2(b). We call this operation *splitting*. The subdivided intervals are assigned pixel values that, if recombined, would reproduce the undivided interval's value.

Having been reconciled, the values of corresponding subintervals can be *mixed* to produce a merged list that will have no overlaps, as in Figure 2(c).

The splitting and mixing functions must be carefully chosen to avoid anomalous behavior. Kainz [2013] and Hillman [2012] describe in detail what *OpenEXR* intends the splitting and mixing functions to be. Note that Egstad et al. [2015] describe an interesting extension using bitmasks instead of just α to represent opacity, an important development orthogonal to the matters we discuss in this article.

The *OpenEXR* deep image implementation builds on the deep shadow work of Lokovic and Veach [2000]. Deep shadows are a special case of deep images that attenuate incoming light but add no new color of their own. Merging deep shadows is simple. If an interval with transmittance τ is to be split into two subintervals of relative lengths λ and $1 - \lambda$, the subinterval transmittances are τ^{λ} and $\tau^{1-\lambda}$. Combining two (reconciled) intervals with transmittances τ and υ gives a transmittance of $\tau\upsilon$.

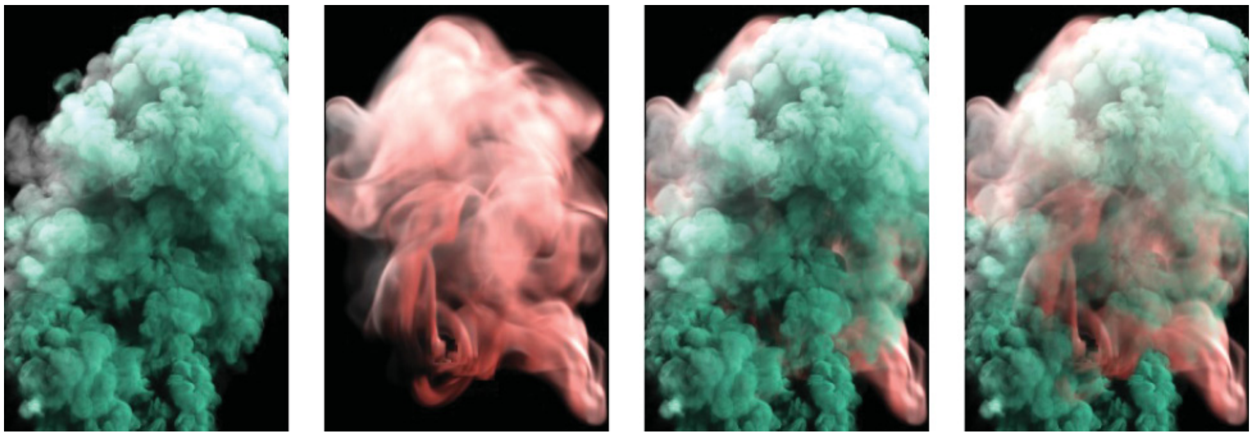
By analogy with the deep shadow situation, when A and B are pixel values (rather than just transmittances), we will refer to the pixel value splitting function as A^{λ} and the mixing function as $A \otimes B$.

2. CONTRIBUTIONS

We start by showing that pixel values together with the **over** operator form a group, which we call **Ov**, and that **Ov** is a Lie group. There are many good introductions to Lie groups and algebras, including Tapp [2016]. **Ov**, like every Lie group, has an associated Lie algebra, $\mathfrak{ov}$, with an exponential map

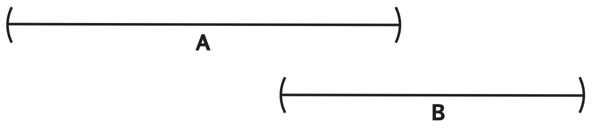
$$\exp : \mathfrak{ov} \rightarrow \mathbf{Ov}$$

relating the two. Using exp and its inverse log map, we can easily write the splitting and mixing functions that *OpenEXR* deep images require and prove that they are, under reasonable assumptions, unique. The splitting and mixing functions themselves are the same as those described in Hillman [2012]. The proof of uniqueness is

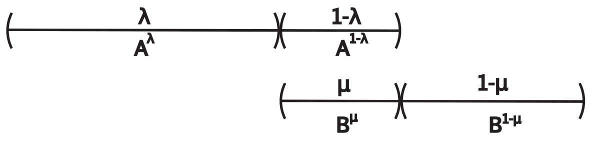


(a) A bluish-green cloud. (b) A reddish-orange cloud. (c) Incorrectly mixed (bluish-green cloud **over** reddish-orange). Being in front, the bluish-green cloud unsurprisingly overwhelms the reddish-orange one. (d) Correctly mixed. Neither cloud dominates the other.

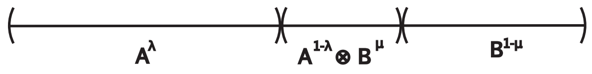
Fig. 1. Incorrect and correct mixing of two clouds. © Disney/Pixar 2017.



(a) Two overlapping intervals, one with value A, the other with value B.



(b) After reconciliation, interval A has two subintervals with lengths in the ratio $\lambda : 1 - \lambda$, with pixel values A^λ and $A^{1-\lambda}$, and B has two subintervals divided $\mu : 1 - \mu$, with pixel values B^μ and $B^{1-\mu}$.



(c) After combining, the region where A and B overlap is assigned pixel value $A^{1-\lambda} \otimes B^\mu$. The meanings of A^λ and $A \otimes B$ are discussed in section 1 and defined mathematically in section 5.

Fig. 2. Reconciling and combining deep pixels.

new, as is the observation that they are particularly simple when expressed in $\mathfrak{ov}$.

Additionally, we use the Lie group structure to define new pixel interpolation functions that are invariant with respect to the splitting function, allowing us to, for example, change the (transverse) resolution of an image in a splitting-independent way. That is, subdividing a volumetric image in Z and then resampling in X and Y produces the same result as resampling and then subdividing.

We can use the same interpolation functions to approximate deep image functions in a way that gives better compression than the piecewise-constant *OpenEXR* approach.

Finally, we show that in some cases, we can bypass deep images altogether. In Appendix B, we will demonstrate a use of **Ov** to correctly composite scenes with nonopaque surfaces and participating media, where the contribution of the medium need only be evaluated at the surfaces.

3. NOTATION, REPRESENTATION, AND INTERPRETATION

A pixel value is a pair of channels $A = (a, \alpha)$, where a is the amount of new light emitted from A and α is the fraction of light that A blocks from shining through it from behind. In Porter and Duff [1984], this representation is referred to as premultiplied color. A pixel value blocks no incoming light when $\alpha = 0$ and all of it when $\alpha = 1$. When we need to talk about more than a single pixel value, we will refer to $B = (b, \beta)$, $C = (c, \gamma)$, and so forth.

In this article, we will, for simplicity, consider both a and α to be real numbers, representing either monochrome emission and opacity or the behavior in a single color channel. Vectors of (a, α) pairs can be used for color imaging—for example, *Renderman* shaders use three emission components and three opacity components. In practice, α will often be the same over all channels, in which case the four-component (r, g, b, α) scheme used in Porter and Duff [1984] is appropriate. For more details about pixel value representations, see Smith [1995] and Blinn [1994].

Pixel values form a vector space, so we can take linear combinations of them:

$$\begin{aligned} u\,A + v\,B &= u\,(a, \alpha) + v\,(b, \beta) \\ &= (u\,a + v\,b,\ u\,\alpha + v\,\beta). \end{aligned}$$

To simplify many expressions, we use the notation

$$\overline{\alpha} = 1 - \alpha.$$

If α is an *opacity* value (0 for transparent, 1 for opaque), $\overline{\alpha}$ is the corresponding transmittance, indicating how much of a more distant image α allows to show through.

$\overline{\alpha}$ satisfies two important identities:

$$\overline{\overline{\alpha}} = \alpha$$
$$\alpha + \overline{\alpha}\,\beta = \overline{\overline{\alpha}\,\overline{\beta}} = \beta + \overline{\beta}\,\alpha.$$

The first of these just says that this operation is its own inverse—the same computation effects conversion from opacity to transparency or vice versa. The second says that computing the α value in the **over** operation (see the next section) is the same thing as multiplying transmittances.

There are at least two different interpretations of pixel values, which we call the *geometric interpretation* and the *optical interpretation*. In the geometric interpretation, scene elements may cover part of the pixel, allowing light from farther away to shine through uncovered parts. In the optical interpretation, scene elements are considered to be partially transparent and allow some of the light from more distant elements to pass through them. The two interpretations are usually interchangeable but can differ in complex situations. For example, in the optical interpretation, A **over** A ought to describe a situation where two identical layers combine, reinforcing their emission and attenuation, whereas in the geometric interpretation, both copies of A should block exactly the same parts of each pixel and we ought to have A **over** $A = A$. So the former A should completely obscure the latter, and neither A should affect any of the background that the other does not. Porter and Duff's algorithm for evaluating complicated composites uses the geometric interpretation, and for them, indeed, A **over** $A = A$. This article is concerned with the optical interpretation, which is appropriate when dealing with layers of clouds and other participating media whose geometry is poorly delineated, and whose effects, when stacked up, are cumulative. Most commercial compositing software, like *Nuke*, *Shake*, and the compositing tools bundled with Pixar's *Renderman*, implicitly uses the optical interpretation. They do none of the analysis described by Porter and Duff, treating A **over** A as if the two As were unrelated. Glassner [2015] goes into more detail about how the geometric and optical interpretations interact.

4. OVER

The **over** operator describes what happens when we stack two pixels and look through the nearer at the farther. What we see is the emission of the nearer pixel in the stack combined with the amount of the farther pixel that the nearer's α allows to shine through from beyond. So we get

$$A \textbf{ over } B \overset{\text{def}}{=} A + \overline{\alpha}\, B.$$

Equivalently,

$$(a,\ \alpha)\,\textbf{over}(b,\ \beta) = (a + \overline{\alpha}\, b,\ \alpha + \overline{\alpha}\,\beta).$$

Pixel values with $\alpha \neq 1$ form a group with **over** as the group operator. That is, the following four easy-to-verify properties all hold:

—Closure: $\alpha + \overline{\alpha}\,\beta$ cannot be 1 if $\alpha \neq 1$ and $\beta \neq 1$.
—Associativity: $(A \textbf{ over } B) \textbf{ over } C = A \textbf{ over}(B \textbf{ over } C)$.
—Identity: there is an identity element, which we call

$$\textbf{clear} = (0,\ 0),$$

with

$$A \textbf{ over clear} = \textbf{clear over } A = A.$$

—Invertibility: every pixel value has an inverse

$$A^{-1} = -A/\overline{\alpha}, \tag{1}$$

satisfying

$$A \textbf{ over } A^{-1} = A^{-1} \textbf{ over } A = \textbf{clear}.$$

Opaque pixel values (those with $\alpha = 1$) are excluded from the group because we cannot compute their inverses. With $\alpha = 1$, we have A **over** $B = A$, in which case we cannot expect to be able to put some magical A^{-1} over A to recover the vanished B. (While opaque pixels are excluded from the group, they are not banned from our computations. Whenever a computation requires an inverse, the $\alpha = 1$ case must often be considered separately, as we will see. This is the compositing analog of being careful about dividing by zero.)

Normally we think of $a \geq 0$ and $0 \leq \alpha \leq 1$, in which case A^{-1} has $-a/\overline{\alpha} \leq 0$ and $-\alpha/\overline{\alpha} \leq 0$. It is hard to attach a physical meaning to pixel values with negative components, but they are mathematically unproblematic.

Note: There are actually at least two different groups with **over** as their operation. Combining pixel values with $\alpha < 1$ (either by **over** or by taking inverses) never produces a result with $\alpha > 1$, so the $\alpha < 1$ case forms a subgroup of the larger group that includes all pixel values with $\alpha \neq 1$. Pixel values with $\alpha > 1$ have no use, except in peculiar circumstances not touched on herein, so we will disregard them in the following discussion.

We will use the name **Ov** to refer to the group of pixel values with $\alpha < 1$ and group operator **over**.

5. SPLITTING AND MERGING

Having introduced the group structure of **Ov**, we now have enough mechanism in place to define the splitting and merging functions required to do deep compositing.

The splitting function A^λ has to satisfy the identities

$$\begin{aligned} A^0 &= \textbf{clear} \quad \text{(but undefined if } \alpha = 1) \\ A^1 &= A \\ A^u \textbf{ over } A^v &= A^{u+v} \\ A^{uv} &= (A^u)^v. \end{aligned} \tag{2}$$

The derivation is easy if we start by considering integer exponents and then extend to arbitrary real exponents. From the properties listed in Equation (2), we can immediately write down

$$\begin{aligned} A^n &= \underbrace{A \textbf{ over}\,(A \textbf{ over}\,(A \textbf{ over} \cdots \textbf{over } A)\cdots))}_{n \text{ terms}} \\ &= A + \overline{\alpha}(A + \overline{\alpha}(A + \cdots + \overline{\alpha}\,A)\cdots)) \\ &= \left(\sum_{i=0}^{n-1} \overline{\alpha}^i\right) A. \end{aligned}$$

This last formula is just A multiplied by a geometric series, which gives

$$A^n = \begin{cases} \frac{1-\overline{\alpha}^n}{\alpha} A & \text{if } \alpha \neq 0, \\ nA & \text{if } \alpha = 0. \end{cases} \tag{3}$$

We can extend this to work for arbitrary real exponents in three steps. First, we can use Equation (3) to compute $A^{1/n}$ by solving $A = B^n$ for B. The result is

$$A^{1/n} = \begin{cases} \frac{1-\overline{\alpha}^{1/n}}{\alpha} A & \text{if } \alpha \neq 0, \\ \frac{1}{n} A & \text{if } \alpha = 0. \end{cases} \tag{4}$$

That is, exactly the same formula works. Now we can extend this to arbitrary rational exponents by noting that $A^{m/n} = (A^{1/n})^m$.

Again, the same formula works. Finally, by continuity, the same formula can be applied to nonrational real exponents.

It is easy to verify that Equation (3) satisfies each of the identities in Equation (2).

Note: In general,

$$A^\lambda \textbf{ over } B^\lambda \neq (A \textbf{ over } B)^\lambda$$

because **over** is not commutative.

The merging function $A \otimes B$ must work correctly even if deep pixels are diced into smaller segments before merging. That is, we must have

$$A \otimes B = (A^\lambda \otimes B^\lambda) \textbf{ over} (A^{1-\lambda} \otimes B^{1-\lambda}),$$

where $0 \leq \lambda \leq 1$. In fact, we can use this idea to *define* $A \otimes B$. First split A and B into n laminae:

$$\begin{aligned} A &= (A^{1/n})^n = A^{1/n} \textbf{ over} \ldots \textbf{over } A^{1/n} \\ B &= (B^{1/n})^n = B^{1/n} \textbf{ over} \ldots \textbf{over } B^{1/n}. \end{aligned}$$

Then intersperse the laminae and take the limit as n approaches infinity:

$$\begin{aligned} A \otimes B &\overset{\text{def}}{=} \lim_{n\to\infty} A^{1/n} \textbf{ over } B^{1/n} \textbf{ over} \ldots \textbf{over } A^{1/n} \textbf{ over } B^{1/n} \\ &= \lim_{n\to\infty} (A^{1/n} \textbf{ over } B^{1/n})^n. \end{aligned}$$

This limit is harrowing to evaluate by hand, but any good computer algebra system should make short work of it (we used *SymPy* [2015]), giving

$$A \otimes B = \frac{\overline{\overline{\alpha}\,\overline{\beta}}}{\log \overline{\alpha} + \log \overline{\beta}} \left(\frac{\log \overline{\alpha}}{\alpha} A + \frac{\log \overline{\beta}}{\beta} B \right), \tag{5}$$

as long as $\alpha, \beta \neq 0$ or 1.

In the cases where at least one of α or β is zero or one, Equation (5) tries to divide by zero. But if we separately simplify the expression under the limit for each case, the limits all converge, unless $\alpha = 1$ and $\beta = 1$:

$$\begin{array}{lll} \alpha = 0, & \beta \neq 0,\ \beta \neq 1, & A \otimes B = \frac{\beta}{\log \overline{\beta}} A + B \\ \alpha \neq 0,\ \alpha \neq 1, & \beta = 0, & A \otimes B = A + \frac{\alpha}{\log \overline{\alpha}} B \\ \alpha = 0, & \beta = 0, & A \otimes B = A + B \\ \alpha = 1, & \beta \neq 1, & A \otimes B = A \\ \alpha \neq 1, & \beta = 1, & A \otimes B = B. \end{array} \tag{6}$$

In the $\alpha = \beta = 1$ case, we can get different values by having α and β approach 1 at different rates, so we must either leave the mixing function undefined in that case or assign a value to it some other way. When $\alpha \neq 1$ or $\beta \neq 1$, $A \otimes B = B \otimes A$, as we would hope. It would be nice if $A \otimes B = B \otimes A$ everywhere. One possible choice is to set $(a, 1) \otimes (b, 1) = (a + b, 1)$, which is both commutative and associative, but there are other possibilities that may behave better in some circumstances. See the discussion at the end of Section 8.

6. MATRIX REPRESENTATION AND GROUP STRUCTURE

We can learn a lot about the structure of **Ov** by looking for other well-known groups that might contain it as a subgroup.

The set of all invertible 2×2 upper-triangular real matrices is a group under matrix multiplication—the product of two such matrices has the same form, as does the inverse. Now, let ρ be the mapping of pixel values to matrices, and thus:

$$\begin{gathered} \rho : \textbf{Ov} \mapsto \textbf{GL}_2(\mathbb{R}) \\ \rho(A) = \begin{bmatrix} \overline{\alpha} & a \\ 0 & 1 \end{bmatrix}, \end{gathered} \tag{7}$$

and then we have

$$\begin{aligned} \rho(A)\rho(B) &= \begin{bmatrix} \overline{\alpha} & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \overline{\beta} & b \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \overline{\alpha + \overline{\alpha}\beta} & a + \overline{\alpha} b \\ 0 & 1 \end{bmatrix} \\ &= \rho(A + \overline{\alpha} B) \\ &= \rho(A \textbf{ over } B) \end{aligned}$$

and

$$\begin{aligned} \rho(A)^{-1} &= \begin{bmatrix} \overline{\alpha} & a \\ 0 & 1 \end{bmatrix}^{-1} \\ &= 1/\overline{\alpha} \begin{bmatrix} 1 & -a \\ 0 & \overline{\alpha} \end{bmatrix} \\ &= \begin{bmatrix} 1/\overline{\alpha} & -a/\overline{\alpha} \\ 0 & 1 \end{bmatrix} \\ &= \rho(-A/\overline{\alpha}) \\ &= \rho(A^{-1}). \end{aligned}$$

So, under the mapping ρ, matrix multiplication corresponds to **over** and the matrix inverse corresponds to the pixel inverse.

Ov can be decomposed into two interesting subgroups. Pixel values with $\alpha = 0$ (those that contribute color without attenuation) form an additive subgroup isomorphic to $(\mathbb{R}, +)$, and those with $a = 0$, which attenuate without contributing color, form a multiplicative subgroup isomorphic to $(\mathbb{R}_{>0}, \cdot)$. This subgroup is represented oddly because the quantities to be multiplied are $\overline{\alpha}$, not α. **Ov** is not composed using the familiar direct product, $(\mathbb{R}, +) \times (\mathbb{R}_{>0}, \cdot)$, because then we would have

$$(a, \alpha) \textbf{ over}' (b, \beta) = (a + b, \overline{\overline{\alpha}\,\overline{\beta}}),$$

but it is rather a *semidirect* product, $(\mathbb{R}, +) \rtimes (\mathbb{R}_{>0}, \cdot)$ [Lang 2002, p. 76], in which b must be attenuated by $\overline{\alpha}$ before adding to a, yielding the correct

$$(a, \alpha) \textbf{ over} (b, \beta) = (a + \overline{\alpha} b, \overline{\overline{\alpha}\,\overline{\beta}}) = (a + \overline{\alpha} b, \alpha + \overline{\alpha} \beta).$$

This is analogous to the situation when combining affine transformations. Computer graphics practitioners will already be familiar, at least informally, with $\textbf{Aff}_3(\mathbb{R})$, the group of invertible 4×4 matrices whose last row is [0 0 0 1], widely used to transform points in three dimensions. An affine transformation can be decomposed into a pair (M, v) of a rotation/scaling matrix and a translation vector. To properly compose two affine transformations $(M_1, v_1) \circ (M_2, v_2)$, we don't just compute $(M_1 M_2, v_1 + v_2)$ but rather $(M_1 M_2, v_1 + M_1 v_2)$, transforming v_2 by M_1 before adding it to v_1.

Ov is isomorphic to (in fact, diffeomorphic, when the group is considered as a manifold, see later) to the subgroup of $\textbf{Aff}_1(\mathbb{R})$ that replaces $(\mathbb{R}_{\neq 0}, \cdot)$ with its positive subgroup $(\mathbb{R}_{>0}, \cdot)$ (i.e., the numbers $\overline{\alpha} > 0$) in the semidirect product.

Note: For simplicity of exposition, this article mostly talks about a monochrome world where pixels have a single color component. It is worth developing our notation to deal with color versions of **Ov**. For pixels with m emissive color components, we can have either a single alpha channel (the case discussed in Porter and Duff [1984]) or a separate alpha channel for each color component. We can denote these $\mathbf{Ov}_{m,n}$ with $n = 1$ or $n = m$. We use **Ov** unadorned by subscripts when it is obvious or doesn't matter which $\mathbf{Ov}_{m,n}$ we are referring to. (In this article, **Ov** usually stands for $\mathbf{Ov}_{1,1}$.) $\mathbf{Ov}_{m,m}$ is just the direct product $\mathbf{Ov}_{1,1}^m = \mathbf{Ov}_{1,1} \times \mathbf{Ov}_{1,1} \times \cdots \times \mathbf{Ov}_{1,1}$, which is the subgroup of $\mathbf{Aff}_m(\mathbb{R})$ that includes translation and positive scaling transformations but not rotations or shears, with a separate scale factor for each axis. $\mathbf{Ov}_{m,1}$ is the subgroup of $\mathbf{Ov}_{m,m}$ that includes only translation and uniform positive scaling. In each case, the group has no surprising new behavior that is not apparent from the study of $\mathbf{Ov}_{1,1}$.

Note also that it makes sense to talk about $\mathbf{Ov}_{0,n}$, isomorphic to $(\mathbb{R}_{>0}, \cdot)^n$, which describes Lokovic and Veach's [2000] deep shadows, which have alpha channels but no emission color.

7. OV IS A LIE GROUP

An important property of **Ov** (beyond its *group*ness) is that the **over** operator and the inverse are both smooth. That is, small changes in their operands produce correspondingly small changes in their results.

Groups in which the group operator and the inverse are smooth functions of their operands are called *Lie groups*. They combine the ideas of *smooth manifold* and *group* in a way that makes them extremely useful in, for example, differential geometry and physics [Arfken et al. 2013]. (Lie groups were first investigated by the Norwegian mathematician Sophus Lie, whose surname is pronounced as though it were spelled *Lee*.)

Every Lie group has an associated linearized structure called a *Lie algebra*, which is the tangent space at the identity. Lie algebras are usually named using the lowercase Fraktur version of the Lie group's name, so we will call ours $\mathfrak{ov}$.

For our purposes, the most important feature of Lie algebras and groups is the existence of a mapping from the Lie algebra to the Lie group called the *exponential map*, as many operations in a Lie group are more easily carried out by mapping them into the appropriate Lie algebra, where everything is linear.

Groups of square matrices under multiplication are Lie groups, provided that, as manifolds, they are closed in the set of all invertible square matrices. Their exponential map is just the usual matrix exponential, defined by the familiar Taylor series with a matrix argument:

$$\exp M = \sum_{n=0}^{\infty} \frac{M^n}{n!}.$$

For 2×2 upper triangular matrices, the matrix exponential can be evaluated in closed form [Moler and Van Loan 2003]:

$$\exp \begin{bmatrix} p & q \\ 0 & r \end{bmatrix} = \begin{cases} \begin{bmatrix} \exp p & \frac{\exp p - \exp r}{p-r} q \\ 0 & \exp r \end{bmatrix}, & \text{if } p \neq r \\ \begin{bmatrix} \exp p & q \exp p \\ 0 & \exp p \end{bmatrix}, & \text{if } p = r. \end{cases} \tag{8}$$

Now we can derive **Ov**'s exponential map and its inverse. First, the inverse. Let

$$\log A = \begin{bmatrix} p & q \\ 0 & r \end{bmatrix}$$

and let us solve

$$\exp \log A = \begin{bmatrix} \overline{\alpha} & a \\ 0 & 1 \end{bmatrix}$$

for $\log A$. In both cases of Equation (8), $\exp r = 1$, so $r = 0$. When $p \neq r$, we have

$$\exp \log A = \begin{bmatrix} \exp p & \frac{\exp p - 1}{p} q \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \overline{\alpha} & a \\ 0 & 1 \end{bmatrix}. \tag{9}$$

Pulling Equation (9) apart componentwise, we get

$$\exp p = \overline{\alpha}\,, \tag{10}$$

which means

$$p = \log \overline{\alpha}\,. \tag{11}$$

Here and elsewhere in this article, log of a number is the natural logarithm.

Substituting Equations (10) and (11) back into the upper right component of Equation (9), we have

$$a = \frac{(\exp p) - 1}{p} q = -\frac{\alpha}{\log \overline{\alpha}} q,$$

so

$$q = -\frac{\log \overline{\alpha}}{\alpha} a.$$

The other case, when $p = r$, is much easier. Since $r = 0$, we have $p = 0$, and immediately we get $q = a$. Summing up,

$$\log A = \begin{cases} \begin{bmatrix} \log \overline{\alpha} & -\frac{\log \overline{\alpha}}{\alpha} a \\ 0 & 0 \end{bmatrix}, & \text{if } \alpha \neq 0 \\ \begin{bmatrix} 0 & a \\ 0 & 0 \end{bmatrix}, & \text{if } \alpha = 0. \end{cases}$$

Note that $\log A$ does not have the form of a pixel value because its lower right element is 0, not 1. Remember that it is a member of the Lie algebra $\mathfrak{ov}$, which is not isomorphic to the Lie group **Ov**. For ordinary invertible pixel values, those with $\alpha < 1$, $\log A$ is a well-behaved one-to-one function. Now we can read the exponential map from Equation (8):

$$\exp \begin{bmatrix} p & q \\ 0 & 0 \end{bmatrix} = \begin{cases} \begin{bmatrix} \exp p & -\frac{\overline{\exp p}}{p} q \\ 0 & 1 \end{bmatrix}, & \text{if } p \neq 0 \\ \begin{bmatrix} 1 & q \\ 0 & 1 \end{bmatrix}, & \text{if } p = 0. \end{cases}$$

Notation: We will write members of $\mathfrak{ov}$ as vectors in double square brackets, like this:

$$[\![q,\ p]\!] \stackrel{\text{def}}{=} \begin{bmatrix} p & q \\ 0 & 0 \end{bmatrix}.$$

So

$$\exp[\![q,\ p]\!] = \begin{cases} \left(-\frac{\overline{\exp p}}{p} q,\ \overline{\exp p}\right), & \text{if } p \neq 0 \\ (q,\ 0), & \text{if } p = 0 \end{cases} \tag{12}$$

and

$$\log(a,\ \alpha) = \begin{cases} [\![-\frac{\log \overline{\alpha}}{\alpha} a,\ \log \overline{\alpha}]\!], & \text{if } \alpha \neq 0 \\ [\![a,\ 0]\!], & \text{if } \alpha = 0. \end{cases} \tag{13}$$

Note that exp and log are bijections, so you can think of $\mathfrak{ov}$ as a different coordinatization of **Ov**.

8. SPLITTING AND MERGING IN **OV**

Working in $\mathfrak{ov}$ gives us greatly simplified equations for A^λ and $A \otimes B$.

It is easy to verify, using Equations (3), (12), and (13), that

$$A^\lambda = \exp(\lambda \log A). \tag{14}$$

Note that $\log A$ is undefined at $\alpha = 1$, but A^λ approaches A as α approaches 1 (as long as $\lambda \neq 0$), so it makes sense to define

$$A^\lambda = A \qquad \text{whenever } \alpha = 1 \text{ and } \lambda \neq 0,$$

which matches Equation (3).

Likewise, $A \otimes B$ is easily expressed in terms of exp and log. The Lie product formula [Trotter 1959] says that for any r and s in a Lie algebra,

$$\exp(r + s) = \lim_{N\to\infty} (\exp(r/N)\ \mathbf{over}\ \exp(s/N))^N.$$

Substituting $r = \log A$ and $s = \log B$ and using Equation (14), we get

$$\exp(\log A + \log B) = \lim_{N\to\infty} (A^{1/N}\ \mathbf{over}\ B^{1/N})^N.$$

That is,

$$A \otimes B = \exp(\log A + \log B). \tag{15}$$

It is straightforward, if tedious, to verify that this gives the same results as Equations (5) and (6).

Equation (15) makes it simple to see that $A \otimes B$ exhibits the qualitative behavior we expect from the mixing function. The order in which deep images are mixed does not matter. That is, $\otimes$ is commutative and associative:

$$A \otimes\ B = B \otimes A \tag{16}$$

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C. \tag{17}$$

The mixing function works if we increase the depth resolution with which we sample the image. Suppose we have two stacks of size n copies of A and B. It does not matter whether we create the stacks A^n and B^n and mix those or mix A and B and stack up n copies of the result. That is, the mixing function satisfies

$$A^n \otimes B^n = (A \otimes B)^n. \tag{18}$$

We call this requirement *splitting invariance*, and it is a generally useful correctness criterion for computations on pixel values. Informally, it captures the idea that the result of the computation should depend on the image being sampled, and that increasing the sampling rate should not cause the result to change.

Mixing powers of a given color works in the obvious way:

$$A^p \otimes A^q = A^{p+q}. \tag{19}$$

Equations (16) to (19) are almost enough to uniquely determine the mixing function. In fact, if we are given its value in an appropriate simple limiting case, Equations (16) to (19) are sufficient to determine the function for all other argument values. For example, imagine a full-intensity light of pixel value (1, 0) uniformly distributed in a volume of uniform nonemissive haze of pixel value $(0,\ \alpha)$. By Equation (15) (with Equations (12) and (13)), we have

$$(1,\ 0) \otimes (0,\ \alpha) = (-\alpha/\log\overline{\alpha},\ \alpha). \tag{20}$$

This makes physical sense. At every relative depth λ in the volume, the (infinitesimal) light emitted at that depth is attenuated by $\overline{\alpha}^\lambda$, so the total light emitted should be

$$\int_0^1 \overline{\alpha}^\lambda\, d\lambda = -\alpha/\log\overline{\alpha}.$$

In summary, Equation (15) satisfies Equations (16) to (20). But might there be some other (simpler?) formula that might work as well? As it turns out, Equation (15) is the only function that satisfies Equations (16) to (20). The proof is fairly simple and is presented in Appendix A.

As an example, Figures 1(a) and 1(b) show a pair of clouds, one greenish-blue and one reddish-orange. Imagine that the images only contain a single interval per pixel, all of identical extent, so splitting is not necessary. Figure 1(c) shows the greenish-blue **over** the red, and Figure 1(d) a correct mix. In Figure 1(c), the greenish-blue cloud (unsurprisingly) dominates the reddish-orange one. Figure 1(d) shows neither dominating the other.

Note that Equation (15) only defines $A \otimes B$ when A and B are **Ov** elements, that is, when they are not opaque. As with A^λ, we can fill in this gap by appealing to continuity. As just A approaches opacity, the answer is $A \otimes B = A$. If both A and B are opaque, there is no obvious answer; we can get different values by allowing α and β to approach 1 at different rates. The intuitive solution, letting $A \otimes B = (a+b,\ 1)$, satisfies Equations (16) to (19) and works well in practice. Hillman [2012] argues for using $A \otimes B = ((a+b)/2,\ 1)$ in this case, but notes that this sacrifices associativity. Another possibility, which we use in practice, is to keep track of the number of opaque pixel values contributing to a mix so that we can sum the pixel values up and divide through at the end to get the effect of $A_1 \otimes A_2 \otimes \cdots \otimes A_n = ((a_1 + a_2 + \cdots + a_n)/n,\ 1)$.

9. INTERPOLATION AND PIXEL VALUE SPLINES

Since operations in $\mathfrak{ov}$ are linear, one could define linear interpolation of pixel values by mapping the pixel values from **Ov** to $\mathfrak{ov}$, linearly interpolating the result and mapping back to **Ov**:

$$\mathrm{lerp}(A,\ B,\ t) = \exp((1-t)\log A + t\log B). \tag{21}$$

It is immediately obvious that this satisfies a subdivision identity analogous to Equation (18):

$$\mathrm{lerp}(A^n,\ B^n,\ t) = \mathrm{lerp}(A,\ B,\ t)^n. \tag{22}$$

Also,

$$\mathrm{lerp}(A^p,\ A^q,\ t) = A^{(1-t)p+tq}, \tag{23}$$

and if $\alpha = \beta$, we have

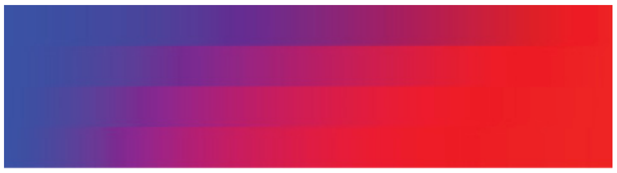
$$\mathrm{lerp}(A,\ B,\ t) = ((1-t)a + tb,\ \alpha). \tag{24}$$

Figure 3 shows $(rgb\alpha)$ pixel values interpolating from (.02 .05 1 .01) on the left to (1 .05 .02 .99) on the right. Figure 3(b) uses Equation (21), whereas Figure 3(a) uses componentwise linear interpolation. Each version is divided depthwise into four stripes, split as in Equation (22) into one, two, four, or eight identical layers (top to bottom). In Figure 3(b), the stripes are identical as per Equation (22), but there is a noticeable color shift from stripe to stripe in Figure 3(a).

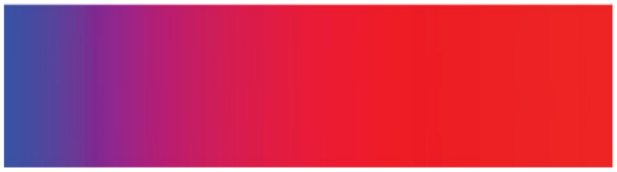
B-splines and Bezier curves are naturally defined by repeated lerps, so the linear interpolation formula immediately gives us well-behaved formulations of pixel-valued B-splines and Bezier curves. For example, a pixel-valued Bezier curve can be defined by

$$S(t) = \exp\sum_{i=0}^{n} b_{i,n}(t)\log B_i$$

where B_i is the Bezier curve's control value and $b_{i,n}(t)$ is the Bernstein polynomial of degree n.



(a) Componentwise linear interpolation



(b) Interpolation using equation (21)

Fig. 3. Pixel value interpolation. Each image is composed of four stripes, subdivided in depth 1, 2, 4, or 8 times (top to bottom) before interpolation. In Figure 3(b) the stripes are identical, but there is a noticeable color shift from stripe to stripe in Figure 3(a), because 3(b) is invariant under depth-splitting, while 3(a) is not.

This can be evaluated by de Casteljau's algorithm:

$$\begin{aligned} B_i^{(0)} &= B_i \\ B_i^{(j)} &= \operatorname{lerp}\left(B_i^{(j-1)}, B_{i+1}^{(j-1)}, t\right) \\ S(t) &= B_0^{(n)}. \end{aligned}$$

Note that many of the logs and exps hidden in the lerp cancel out. The general rule is: take the logs of the control points, make a componentwise spline from those, and take the exponential of the result. This schema works equally well for B-splines using de Boor's algorithm and for any other spline formulation in which the spline value is a linear combination of the control points, like Catmull-Rom [1974] splines or beta-splines [Barsky 1981].

All such schemes satisfy equations analogous to Equations (22) to (24). If we let

$$s(A_1, t_1, \ldots, A_n, t_n) = \exp \sum_{i=1}^{n} t_i \log A_i, \tag{25}$$

where A_i for $1 \leq i \leq n$ is a sequence of pixel values and t_i be a corresponding sequence of weights, then it is easy to verify the following subdivision invariance identity:

$$s\left(A_1^k, t_1, \ldots, A_n^k, t_n\right) = s\left(A_1, t_1, \ldots, A_n, t_n\right)^k \tag{26}$$

and

$$s\left(A^{k_1}, t_1, \ldots, A^{k_n}, t_n\right) = A^{\left(\sum_{i=1}^{n} t_i k_i\right)}. \tag{27}$$

Also, provided all the alphas are the same:

$$\alpha_i = \alpha,$$

and the weights produce an affine combination:

$$\sum_{i=1}^{n} t_i = 1,$$

then

$$s(A_1, t_1, \ldots, A_n, t_n) = \left(\sum_{i=1}^{n} t_i a_i,\ \alpha\right). \tag{28}$$

Furthermore, interpolants can be merged by merging their coefficients:

$$\begin{aligned} & s(A_1, t_1, \ldots, A_n, t_n) \otimes s(B_1, t_1, \ldots, B_n, t_n) \\ & \quad = s(A_1 \otimes B_1, t_1, \ldots, A_n \otimes B_n, t_n) \end{aligned}. \tag{29}$$

Thus, for any spline type that is linear in its coefficients, pairs of splines can be easily merged by first inserting knots to get their knot vectors to match and then merging corresponding coefficients using $\otimes$.

10. INTERPOLATION APPLICATIONS

10.1 Resampling Images with Alpha Channels

We can use Equation (25) to resample images with alpha channels. If we do, Equation (26) says that the resampled image is splitting invariant—that is, if we want to stack up multiple copies of a resampled image, it does not matter whether we stack first and then resample or vice versa. In contrast, this is not true if we interpolate componentwise, as we saw in Figure 3.

Equation (28) says that in regions of an image where α is unchanging, resampling using Equation (25) has the same effect as just resampling the color channels in the standard way and leaving alpha alone. Equation (27) says that in regions where α is changing (but a is a constant proportion of α), the effect of Equation (25) is just to resample the image's density (the logarithm of $\overline{\alpha}$) in the standard way, keeping the color proportional to α.

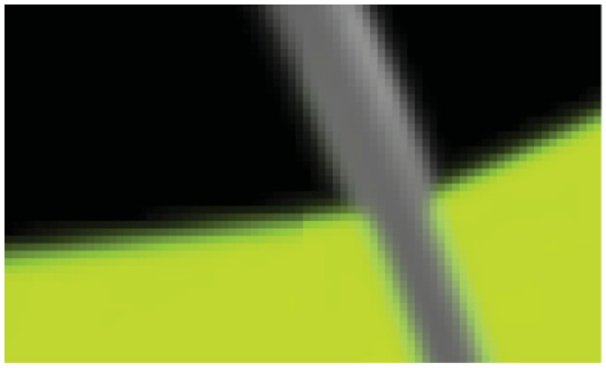
Figure 4 shows an example. Each of the two images is resampled from a detail of the RC car of Figure 8(a). Figure 4(a) is resampled componentwise in $(r\,g\,b\,\alpha)$ space, while Figure 4(b) is done by using the log map on each pixel, interpolating componentwise in $\mathfrak{ov}$ and converting back to $\mathbf{Ov}$ with the exp map.

The right half of each image is split depthwise into 100 laminae (i.e., we computed $A^{1/100}$ for each pixel). The laminae are resampled and then each stack is recomposited using **over**. The left half of each image is resampled directly, without depth splitting. As we would expect, the left and right halves of Figure 4(b) match seamlessly, but Figure 4(a) has a noticeable (if you look carefully) discontinuity between its subdivided and unsubdivided halves.

10.2 Deep Image Compression

Figure 5(a) is an image of a cloud reconstructed from a $154 \times 274 \times 154$ voxel array of pixel values. The final image is calculated from a stack of 154 layers composited each **over** the next. Recall that the *OpenEXR* deep image scheme stores a sequence of nonoverlapping intervals each with a single pixel value, and would effectively approximate this $154 \times 274 \times 154$ volume in a piecewise constant way. The value stored with each interval is the accumulated pixel value of the voxels it represents.

The deep image can be compressed by approximating the 154 intervals per pixel by fewer constant samples. One would expect that approximating using a higher-order interpolant would give better compression for the same error. We tested this by compressing the image using both a piecewise-constant-based compression and a linear interpolant-based compression, interpolated using Equation (21). In detail, we constructed interpolants to each pixel's voxel column by recursive binary subdivision. For a given range of voxels in the column, we find the voxel that deviates most from the interpolant determined by the range's endpoints. If that deviation is below some threshold, we stop subdividing. Otherwise, we add a control point at that voxel and recursively subdivide the two subranges.


(a) Componentwise resampled

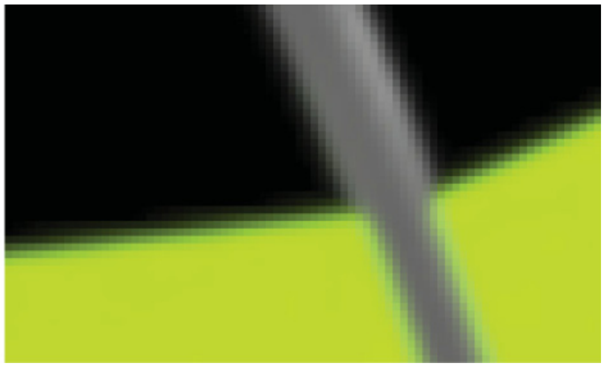
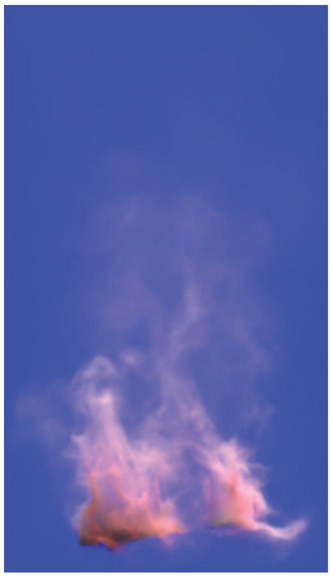
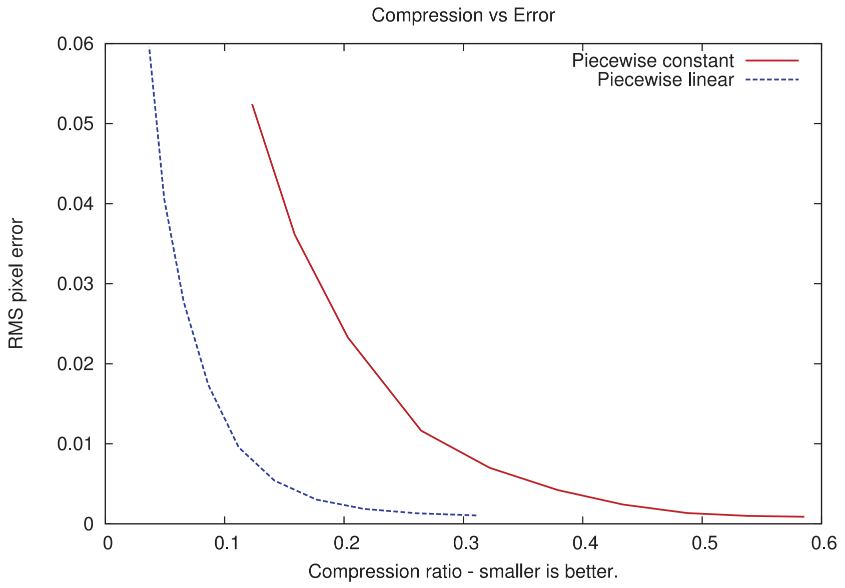

(b) Resampled using $\mathfrak{ov}$

Fig. 4. These pictures illustrate image resampling (magnification) using $\mathfrak{ov}$. The image being resampled is a detail of the wing and antenna of the RC car element used in Figure 8. The left half of each image is resampled directly. The right half of each is split depthwise 100 times, then each layer is resampled, and all 100 are recomposited. Figure 4(a) was resampled componentwise in **Ov**. You can see the discontinuity between the two halves if you look carefully. In Figure 4(b), which was done by first mapping from **Ov** to $\mathfrak{ov}$, resampling componentwise in $\mathfrak{ov}$, and mapping back to **Ov**, the join is invisible, as expected. © Disney/Pixar 2017.


(a) One of the test images used to evaluate deep image compression methods © Disney/Pixar 2017



(b) Plot of compression ratio vs RMS pixel error for the scene of figure 5a. Compression ratio is the number of control points in the compressed representation divided by the number of uncompressed voxels. The red curve corresponds to *OpenEXR*-like piecewise constant compression. The blue curve shows how our piecewise linear compression performs. For any particular RMS pixel error, the piecewise linear compression curve is considerably to the left (in the direction of better compression) of the piecewise constant curve.

Fig. 5. Deep image compression.

Figure 5(b) plots root-mean-square (RMS) error in the reconstructed image versus compression ratio (the number of control points divided by the number of uncompressed voxels) for both the piecewise constant and piecewise linear schemes. The piecewise linear scheme is a clear winner, giving better compression over the whole range of error values. The maximum tolerable RMS pixel error (determined by eye) for this test image is about 0.01, for which the piecewise constant scheme gives a compression ratio of 0.34, while our new piecewise linear scheme gives 0.11, roughly a 3 times improvement.

We ran the compression test on 159 production volumes of varying sizes. The graph of Figure 6 shows the relative performance of piecewise linear and piecewise constant compression at 0.01 RMS pixel error. Only on a very few of our test volumes (three out of 159) did piecewise constant compression outperform piecewise linear. The worst that the piecewise linear scheme did was compress

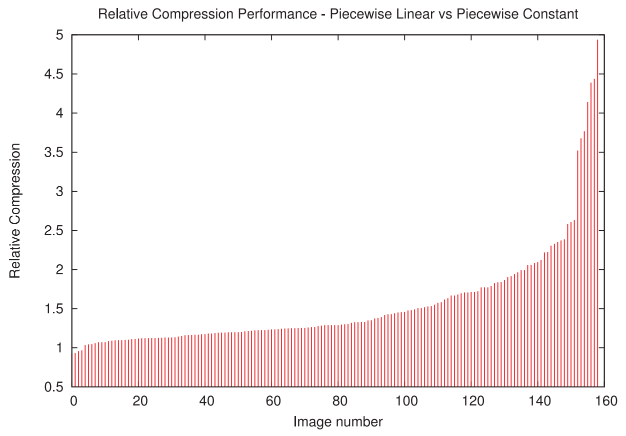


Fig. 6. Relative performance of piecewise linear and piecewise constant compression. For each of 159 test volumes, we plot the size (control point count) of the linear version divided by the size of the constant version. Ratios larger than 1 indicate that the linear version performs better than the constant version.

0.93 as well as the piecewise constant scheme. The best was 4.9 times better and the mean was 1.5 times better. That is, on average, linear-compressed images occupied 0.67 times as much space as constant-compressed ones.

11. SUMMARY

We have shown that pixel values with the Porter-Duff **over** operator form a Lie group, **Ov**. We derived the exponential map from the corresponding Lie algebra $\mathfrak{ov}$ to **Ov** and its inverse, the log map. We used them to provide simple rederivations of the splitting and mixing functions, A^λ and $A \otimes B$, required for merging *OpenEXR* deep images (originally derived in Hillman [2012]) and proved that the mixing function is uniquely determined by a small set of properties that any such function ought to have. The most important of these properties, splitting invariance, is an instance of a new general correctness criterion for computations on pixel values with alpha channels.

We hinted at the versatility of the Lie group framework by using it to formulate a splitting invariant interpolation function, lerp(A, B, t), and spline functions suitable for resampling images with alpha channels. We also showed how to use lerp(A, B, t) to produce piecewise linear approximations to volumetric images with substantially better compression than the existing piecewise constant approximant used in *OpenEXR*.

Additionally, we show in Appendix B how **Ov** can be used to correctly composite partially transparent surfaces in participating media, even without the machinery of deep images.

12. FUTURE WORK

The major message of this article is that the Lie group **Ov** and its Lie algebra $\mathfrak{ov}$ are powerful tools for attacking complicated questions about compositing. Previously known solutions, like the splitting and mixing functions, turn out to be exquisitely simple when viewed from the Lie algebra point of view, and their solutions immediately suggest approaches to other problems.

We do not yet understand in detail how the geometric and optical interpretations interact in deep compositing. Egstad et al. [2015] provide a good first step in that direction; we think the matter deserves closer investigation.

Another thing we should investigate is the use of higher-order approximating functions, which may offer even better compression of deep images. Methods that exploit pixel-to-pixel coherence would be of obvious benefit.

We would like to have a comprehensive way of fitting opaque pixel values into this framework. As we have seen, appealing to continuity works well in some circumstances, but that is not universal. Looking at Equation (13), the problem arises because $\log \overline{\alpha} = -\infty$ when $\alpha = 1$. For now, when computing $\log \overline{\alpha}$, our code substitutes a finite negative value large enough that in Equation (12), $\overline{\exp p} = 1$ after floating-point rounding. Effectively, we are pretending that opaque pixels are transparent, but not so transparent as to be noticeable to floating-point arithmetic. This is bound to be unsatisfactory in general but works well in all the cases we have seen so far.

$A \otimes B$ is a sort of volumetric union operator. That suggests looking for other related operators that might facilitate Constructive Solid Geometry-like manipulation of volumes. For example, volume-CSG subtraction might be implemented by subtraction in $\mathfrak{ov}$, appropriately clamped so as not to generate pixels with negative α.

APPENDIX

A. UNIQUENESS OF THE $\otimes$ OPERATOR

THEOREM. *Equations* (16) *to* (20)*, repeated here:*

$$A \otimes B = B \otimes A \tag{30}$$

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C \tag{31}$$

$$A^n \otimes B^n = (A \otimes B)^n \tag{32}$$

$$A^p \otimes A^q = A^{p+q} \tag{33}$$

$$(1,\ 0) \otimes (0,\ \overline{\alpha}) = (-\alpha/\log \overline{\alpha},\ \alpha) \tag{34}$$

uniquely determine that

$$A \otimes B = \exp(\log A + \log B).$$

PROOF. The exp and log mappings are 1-1 and onto. So every function

$$A \otimes B : \mathbf{Ov} \times \mathbf{Ov} \to \mathbf{Ov}$$

corresponds uniquely to a function

$$P \diamond Q : \mathfrak{ov} \times \mathfrak{ov} \to \mathfrak{ov},$$

with

$$A \otimes B = \exp(\log A \diamond \log B). \tag{35}$$

Using Equation (35), Equations (30) to (34) can be rewritten in terms of $\diamond$. For example, Equation (30) is the same as

$$\exp(\log A \diamond \log B) = \exp(\log B \diamond \log A).$$

Letting $\log A = P$ and $\log B = Q$, this is equivalent to

$$P \diamond Q = Q \diamond P. \tag{36}$$

Similarly, Equations (31) to (34) are equivalent to

$$P \diamond (Q \diamond R) = (P \diamond Q) \diamond R, \tag{37}$$

$$(nP) \diamond (nQ) = n(P \diamond Q), \tag{38}$$

$$(mP) \diamond (nP) = (m+n)P \tag{39}$$

and

$$[\![1,\ 0]\!] \diamond [\![0,\ p]\!] = [\![1,\ p]\!]. \tag{40}$$

So, if we have a unique function $\diamond$ satisfying Equations (36) to (40), Equation (35) gives us a unique $\otimes$, satisfying Equations (30) to (34).

The following lemma and a little algebra allow us to complete the proof:

LEMMA A.1.

$$[\![t,\ u]\!] = [\![t,\ 0]\!] \diamond [\![0,\ u]\!] \tag{41}$$

PROOF. If $t \neq 0$,

$$\begin{aligned} [\![t,\ u]\!] &= t[\![1,\ u/t]\!] \\ &= t([\![1,\ 0]\!] \diamond [\![0,\ u/t]\!]) && \text{by Equation (40)} \\ &= [\![t,\ 0]\!] \diamond [\![0,\ u]\!] && \text{by Equation (38).} \end{aligned}$$

Otherwise, $t = 0$, and we have

$$\begin{aligned} [\![0,\ u]\!] &= (0+1)[\![0,\ u]\!] \\ &= 0[\![0,\ u]\!] \diamond 1[\![0,\ u]\!] && \text{by Equation (39)} \\ &= [\![0,\ 0]\!] \diamond [\![0,\ u]\!]. \quad \square \end{aligned}$$

Now, letting $\log A = [\![p,\ q]\!]$ and $\log B = [\![r,\ s]\!]$, we are trying to derive a definition for $[\![p,\ q]\!] \diamond [\![r,\ s]\!]$:

$$\begin{aligned} [\![p,\ q]\!] \diamond [\![r,\ s]\!] &= ([\![p,\ 0]\!] \diamond [\![0,\ q]\!]) \\ &\quad \diamond ([\![r,\ 0]\!] \diamond [\![0,\ s]\!]) && \text{by Equation (41)} \\ &= (p[\![1,\ 0]\!] \diamond r[\![1,\ 0]\!]) \\ &\quad \diamond (q[\![0,\ 1]\!] \diamond s[\![0,\ 1]\!]) && \text{by Equations (36) and (37)} \\ &= [\![p+r,\ 0]\!] \diamond [\![0,\ q+s]\!] && \text{by Equation (39)} \\ &= [\![p+r,\ q+s]\!] && \text{by Equation (41)} \\ &= [\![p,\ q]\!] + [\![r,\ s]\!], \end{aligned}$$

and, using Equation (35), we get

$$A \otimes B = \exp(\log A + \log B),$$

as required. $\square$

B. PARTICIPATING MEDIA WITHOUT DEEP IMAGES

In some cases, deep images are more mechanism than we need in order to deal with participating media. For example, in a REYES renderer [Cook et al. 1987] like Pixar's *Renderman*, surfaces are shaded before their depth order is determined and there is no explicit representation of participating media. So atmospheric effects must be incorporated into surface shaders, and that calculation has no access to information about other surfaces that may obscure parts of the atmosphere. A similar situation can arise when elements are separately rendered to be combined in a compositing package like *Shake* or *Nuke*.

Figure 8(e) shows a partially transparent image of an RC car embedded in a cloud bank. Let us try to reproduce this image without using any deep compositing. Having no nondeep way to represent the cloud element, its effects must be taken into account by the surface shaders used to render the RC car and the background sky. Figures 8(a) and 8(b) show the results. When we composite the two naively, we get Figure 8(d). The part of the cloud that is in front of the RC car is also included in the sky element, and so is double-counted in Figure 8(d). Without making the sky's renderer aware of the RC car obscuring part of its cloud, there appears to be little we can do to get a correct final result, as in Figure 8(e).

Nevertheless, we can render scene components separately and composite the results correctly with the appropriate compositing math. Suppose that, along a particular eye ray, the colors and opacities of the surfaces (in depth order) are S_i, $1 \le i \le n$, and that the color and opacity of the part of the cloud between the eye position and S_i is F_i. F_{i+1} can be divided up into two parts: the contribution from the eye up to S_i (i.e., F_i) and the added contribution between S_i and S_{i+1} (let us call that C_{i+1}). See Figure 7. So the pixel value we want to compute is

$$\begin{aligned} &F_1 \textbf{ over } S_1 \textbf{ over} \\ &C_2 \textbf{ over } S_2 \textbf{ over} \\ &C_3 \textbf{ over } \cdots \textbf{ over} \\ &C_n \textbf{ over } S_n. \end{aligned}$$

Now at the time we render each separate element, we can compute F_i but not C_i. But

$$F_{i+1} = F_i \textbf{ over } C_{i+1},$$

so

$$C_{i+1} = F_i^{-1} \textbf{ over } F_{i+1},$$

and we can rewrite the compositing stack as

$$\begin{aligned} &F_1 \textbf{ over } S_1 \textbf{ over } F_1^{-1} \textbf{ over} \\ &F_2 \textbf{ over } S_2 \textbf{ over } F_2^{-1} \textbf{ over} \\ &F_3 \textbf{ over } \cdots \textbf{ over} \\ &F_n \textbf{ over } S_n. \end{aligned}$$

So if we have the renders of the various elements output F_i **over** S_i **over** F_i^{-1}, the compositing package will be able to compute a correct result.

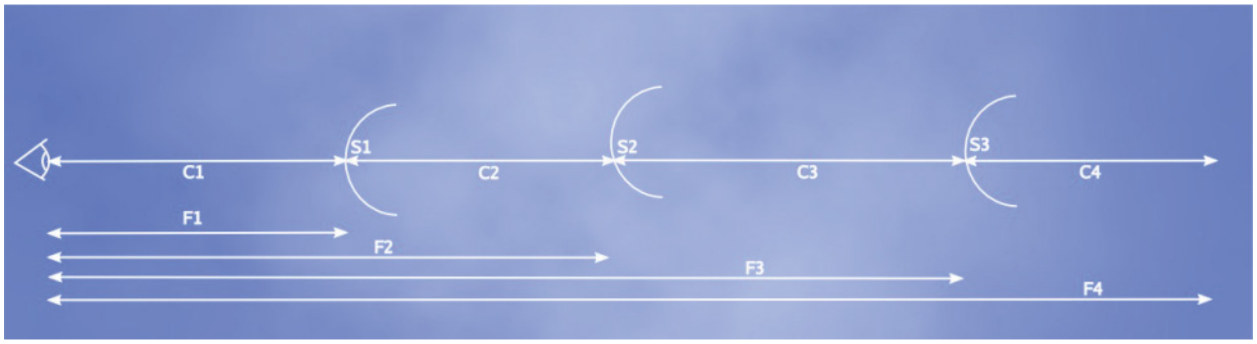
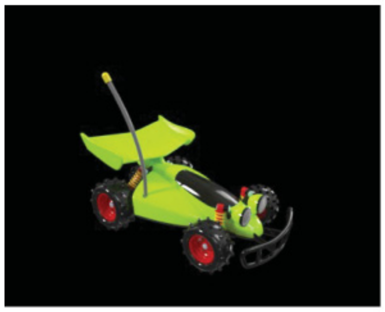
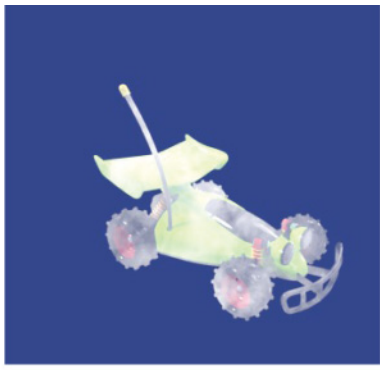


Fig. 7. Compositing with participating media and transparent surfaces. C_i is the part of the medium between surfaces S_{i-1} and S_i. F_i represents the part of the medium between the viewpoint and surface S_i.

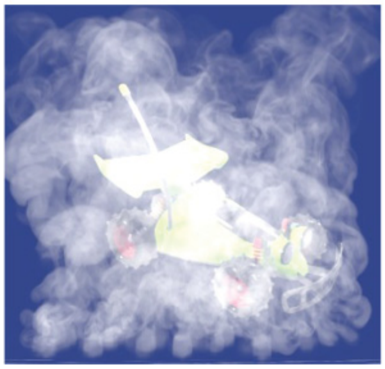


(a) An RC car element, before embedding in a cloud.

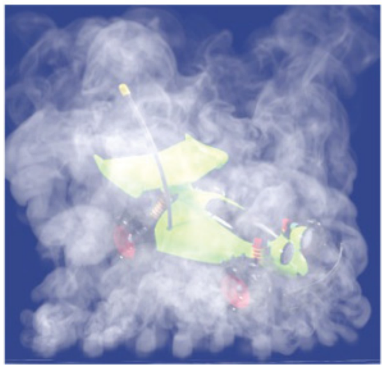
(b) Cloudy sky.



(c) The RC car element, after embedded in the cloud.



(d) The rc car of figure 8c **over** the sky of figure 8b. Because of double counting, the part of the cloud that is in front of the rc car is too bright.



(e) Correct composite, with double-counting compensation. The cloud is now the correct brightness everywhere, and the rc car, especially its spoiler, is not too bright.

Fig. 8. Two elements and their composites, without and with compensation for double-counting. © Disney/Pixar 2017.

The computation of F_i **over** S_i **over** F_i^{-1} is even simpler than it looks. Let $F_i = (f_i, \phi_i)$ and $S_i = (s_i, \sigma_i)$. If F_i is not opaque (i.e., $\overline{\phi}_i \neq 0$), then

$$\begin{aligned} F_i \textbf{ over } S_i \textbf{ over } F_i^{-1} &= F_i + \overline{\phi}_i (S_i - \overline{\sigma}_i\, F_i / \overline{\phi}_i) \\ &= (1 - \overline{\sigma}_i) F_i + \overline{\phi}_i\, S_i \\ &= \sigma_i F_i + \overline{\phi}_i\, S_i. \end{aligned}$$

Of course, if $\phi_i = 1$, F_i^{-1} does not exist. In that case, the appropriate answer is just F_i.

When we composite the layers, we will get the correct result, except for an extra F_n^{-1} at the end. If S_n is opaque, that is not a problem, because then F_n **over** S_n **over** $F_n^{-1} = F_n$ **over** S_n. If we cannot guarantee S_n's opacity, we can simply add an opaque black surface S_{n+1} behind an appropriate F_{n+1} behind the whole scene, which will cancel out the spurious F_n^{-1} and add nothing else to the image.

ACKNOWLEDGMENTS

Many colleagues, including Tony Apodaca, Tony DeRose, Matthias Goerner, Mark Meyer, Tom Nettleship, Mitch Prater, Rick Sayer, and Adam Woodbury, have directed and encouraged this work in conversation and careful readings of countless drafts of the article.

In particular, Rick, Adam and Mitch started me down this path when they showed me a shader implementing a close relative of the method of Appendix B, which Mitch had arrived at empirically but could not prove to be correct.

The referees gave thorough and insightful notes throughout the review process. Their advice has greatly improved the content and presentation, as well as my understanding of the underlying ideas.

REFERENCES

George B. Arfken, Hans-Jürgen Weber, and Frank E. Harris. 2013. *Mathematical Methods for Physicists: A Comprehensive Guide* (7th ed.). Academic Press, New York, NY, USA. xiii + 1205 pages.

Brian Andrew Barsky. 1981. *The Beta-Spline: A Local Representation Based on Shape Parameters and Fundamental Geometric Measures*. Ph.D. Dissertation. University of California at Berkeley.

Jim Blinn. 1994. Image compositing–theory. *IEEE Computer Graphics and Applications* 14, 5 (Sept. 1994), 83–87.

Edwin Catmull and Raphael Rom. 1974. A class of local interpolating splines. In *Computer Aided Geometric Design*, Robert E. Barnhill and Rich F. Reisenfeld (Eds.). Academic Press, New York, NY, 317–326.

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes image rendering architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'87)*. ACM SIGGRAPH, ACM, New York, NY, 95–102. DOI:http://dx.doi.org/10.1145/37401.37414

Jonathan Egstad, Mark Davis, and Dylan Lacewell. 2015. Improved deep image compositing using subpixel masks. In *Proceedings of the 2015 Symposium on Digital Production (DigiPro'15)*. ACM SIGGRAPH, ACM, New York, NY, 21–27. DOI:http://dx.doi.org/10.1145/2791261.2791266

Andrew Glassner. 2015. Interpreting alpha. *Journal of Computer Graphics Techniques (JCGT)* 4, 2 (May 2015), 30–44.

Peter Hillman. 2012. The Theory of OpenEXR Deep Samples. Retrieved from http://www.openexr.com/TheoryDeepPixels.pdf.

Florian Kainz. 2013. Interpreting OpenEXR Deep Pixels. Retrieved from http://www.openexr.com/InterpretingDeepPixels.pdf.

S. Lang. 2002. *Algebra*. Springer New York, New York, NY.

Tom Lokovic and Eric Veach. 2000. Deep shadow maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'00)*. ACM SIGGRAPH, ACM Press/Addison-Wesley Publishing Co., New York, NY, 385–392. DOI:http://dx.doi.org/10.1145/344779.344958

Cleve Moler and Charles F. Van Loan. 2003. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review* 45, 1 (2003), 3–49. DOI:http://dx.doi.org/10.1137/S00361445024180

Thomas Porter and Tom Duff. 1984. Compositing digital images. *Computer Graphics* 18, 3 (July 1984), 253–259. DOI:http://dx.doi.org/10.1145/964965.808606

Alvy Ray Smith. 1995. *Image Compositing Fundamentals*. Technical Report. Microsoft.

SymPy Development Team. 2015. SymPy Documentation. http://docs.sympy.org/latest/index.html.

Kristopher Tapp. 2016. *Matrix Groups for Undergraduates* (2nd ed.). American Mathematical Society, Providence, RI. viii + 239 pages.

H. F. Trotter. 1959. On the product of semi-groups of operators. *Proceedings of the American Mathematical Society* 10 (1959), 545–551.

Received September 2016; accepted December 2016