Computational Aesthetics in Graphics, Visualization, and Imaging
Angus Forbes and Lyn Bartram (Editors)

EXPRESSIVE 2016

Barycentric Shaders: Art Directed Shading Using Control Images

E. Akleman, S. Liu and D. House

(a) by Jay Jackson and Tim Swartz, after an illustration by Goro Fujita.

(b) by Yinan Xiong, after an illustration by Artyakov Artyom.

Figure 1: *Example frames from class projects created using our method that provides intuitive art-directed control of expressive style.*

Abstract

In this paper, we present Barycentric Shaders, a shading framework based on barycentric algebra, for the development of shader functions providing intuitive art-directed control. The framework guarantees that whatever style is desired will be consistently obtained regardless of the underlying rendering method and illumination model, since our shaders are used only to compute colors based on the incoming illumination. This property of our framework allows our shaders to be included in any rendering pipeline without major changes. We define barycentric operations over positive real colors to guarantee that results will always be positive real colors. With this formalization, we redefine shader functions as parametric functions that satisfy the partition of unity. This property supports an intuitive interaction mechanism for obtaining desired styles by guaranteeing that colors will always stay inside of the convex hull of a set of control colors. To obtain colored light effects, we extend our barycentric methods by allowing computation separately along each color channel, providing the convex hull property for each dimension independently. This results in a more relaxed rectangular box property without significantly changing visual style. This formalism can be particularly helpful to artists, who may not have mathematical training, by simplifying shader development to obtain a desired expressive style. This new approach also suggests a new rendering and shading architecture that provides a clear distinction between illumination and shading.

1. Motivation

In visual narrative, the term *look-and-feel* refers to the unique expressive style for defining the world of the story and its characters, and is one of the most important important attributes differentiating one narrative from another [Ols98,Blo14,pix15]. The identification of a desired art style, therefore, is one of the key artistic decisions that is made during the initial stage of the production process. Unfortunately, development of a rendering and shading framework to obtain a look-and-feel that is consistent with a desired art style is not an easy task.

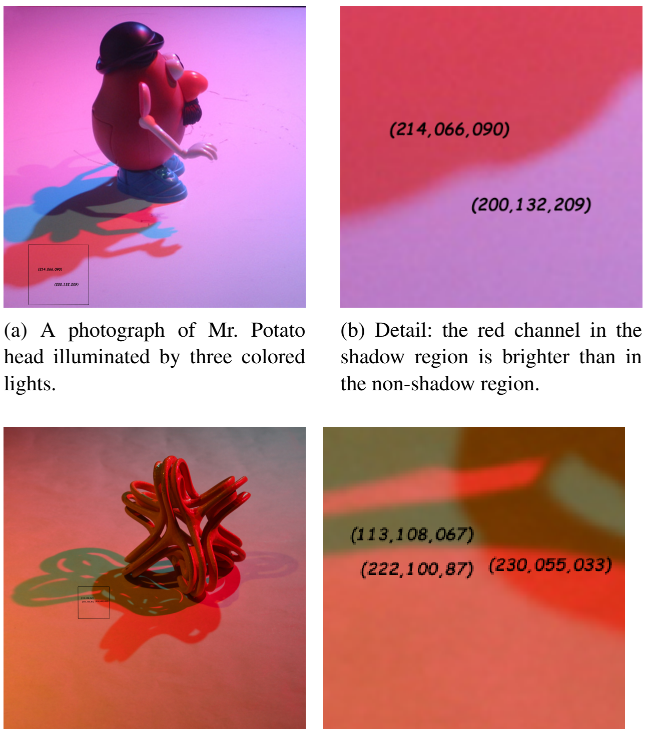
Providing support for the range of visual styles required to create any desired look-and-feel was one of the goals of the highly influential *Shade Trees* architecture [Coo84]. This architecture laid the foundations of the procedural shader concept, and quickly evolved into the *Renderman* shading language [CCC87, HL90, UAD*90, AGB00]. The main goal of such a shading language is to provide users easy control over the look-and-feel of the rendered image.

DOI: 10.2312/exp.20161062

The conceptual flexibility of the procedural shader, which can be developed with functional operators, has made *Renderman* an industry standard since the 1990's [Ebe03]. The shader concept is now ubiquitous in graphics, as it is central to the modern GPU architecture, which is structured around a sequence of programmable shaders [PMTH01, MGAK03, LKM01, RLKG*09].

Despite the enormous success of shading languages, in current film making practice highly qualified lighting technical directors regularly need to hand craft custom "one-off" shading solutions to obtain a desired style. Such work requires a high degree of experience and expertise, as solutions tend not to be intuitive. The need for considerable skilled custom work on shader development contributes to the high number of resources spent on art direction. Therefore, there exists a critical need for new approaches to simplify rendering and shading development through a more intuitive and streamlined process.

This problem has not been appropriately acknowledged until now, since there exists a false impression that obtaining a desired style is not that difficult. It is easy to understand where this impression comes from. There exists a large number of publicly shared shader examples for a wide variety of styles. Creation of a version of an existing shader is not that difficult. People can always create a new shader by making minor changes to a publicly shared shader that provides a style they like. It is also possible to obtain an interesting style by just randomly playing with functions. On the other hand, the look development phase of a project will typically rely on artists who use hand drawn and rendered images to convey their visual concepts. Thus, the real art direction problem is very close to that of matching the style of a painting or an illustration. This is a very different problem that cannot typically be solved by tweaking existing shaders, and is the demonstration problem that we set for ourselves in this paper.



(a) A photograph of Mr. Potato head illuminated by three colored lights.

(b) Detail: the red channel in the shadow region is brighter than in the non-shadow region.

(c) A photograph of a high-genus sculpture illuminated by three colored lights.

(d) Detail: both red and green channels in the shadow regions are brighter than in the non-shadow region.

Figure 2: *Two photographs that demonstrate conceptually inconsistent — yet real — shadows.*

2. Introduction

Our point of view is that the complexity of obtaining a desired style with shaders comes from the fact that the there is a mismatch between the underlying algebra of the shade tree framework with its elements. Abstractly, colors are the elements of the algebra, and are represented by n-tuples of positive real numbers. On the other hand, operations to create new colors are closed over all real numbers, not just positive real numbers. Because of this mismatch, shader networks do not guarantee mathematical consistency. To make things worse, the physical concepts rooted in shader development tend to cause us to ignore this mathematical inconsistency. Therefore, ad-hoc solutions (e.g. negative lights or material colors), introduced to circumvent problems resulting from this mathematical inconsistency, have become ingrained in the community as accepted practice.

Recent advances in physically based rendering [HPJ12, GKDS12] require a formal resolution of this inconsistency. For instance, in the new version of Renderman it is now possible to have negative colors [HJB*12]. Moreover, new color models such as projective alpha colors provide a theoretical foundation for a vector definition of colors [Wil06]. This new direction provides a solution for obtaining physically acceptable images. However, it does not provide the control necessary for obtaining a desired look-and-feel that may not necessarily be physically acceptable. Therefore, despite the recent advances in physically based rendering, the need for obtaining desired styles through a simple and streamlined process is still an important issue.

We observe that it is easy to obtain desired styles by maintaining colors as points, i.e. n-tuples of positive numbers, rather than vectors whose elements can be negative. This restriction allows artists to build color control in a shader using color points much like a modeler uses control points to define the shape of a surface. This approach provides an intuitive control mechanism for artists working to obtain a final look-and-feel. For mathematical consistency, all we need is a formal algebra that guarantees that we deal with and produce only n-tuples of positive real numbers. Fortunately, such algebras already exist, and are known as barycentric algebras [RS90, CR12]. These are well known and widely used in geometric modeling applications as barycentric coordinates.

In this paper, we provide a theoretical formalism to obtain any desired art style with relative ease by employing barycentric algebras. Based on this formalism, we present a simple mathematical approach to art directed shader development. We have tested this approach over three semesters in a graduate rendering and shading class. As a project in the course, the students each choose an artist's style to mimic, and then create rendered images strongly resembling that style. Our thesis is that the method we are proposing provides shader developers an intuitive process, giving them a high level of visual control in the creation of stylized depictions. The experience of the students on this project demonstrates the ease with which a particular visual style can be attained. Two examples of student work are shown in Figure 1, with additional examples shown at the end of the paper.

2.1. Restriction for Better Artistic Control

Although there has been a significant amount of research on non-(photo)realistic rendering (NPR), a theoretical formalism for shader development has not yet been developed that can provide a mathematically consistent non-realistic approach. Developing an algebraic formalism does not mean including more operations and a higher level of elements. In fact, in this case, it is the opposite. We actually restrict the operations to the ones that can guarantee obtaining only positive real numbers when applied to positive real numbers.

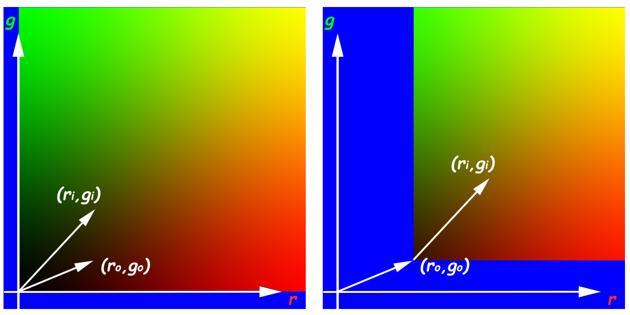
We can demonstrate why we need such a restriction by examining conceptual inconsistencies in shadow regions of real photographs. Consider Figure 2(b), which is a detail of the photograph in Figure 2(a). In this case, the red component of the shadow region is $r_0 = 214$ and the red component of the illuminated region is $r_1 = 200$. Let a variable t denote illumination, such that $t = 1$ means fully illuminated and $t = 0$ means not illuminated. Then, we can write a Blinn-Phong style shading equation in the form $r = a + bt$ to define the red component in the image. In this case, If we solve this using the known values, we have $r_0 = a$ and $r_1 = a + b$, so that $a = 214$ and $b = -14$. This is a problem, since b is expected to be a color. In other words, to replicate this shadow we need to introduce some negative numbers, which we would have to consider to be negative lights. The photographs in Figures 2(c) and (d) provide another example, with similar issues for both the red and green channels.

We call this a conceptual inconsistency, since this is a phenomenon that is not logically expected. Unfortunately for shader development, such inconsistencies frequently happen and they do not necessarily come from the non-linearity of real photographs in RGB space. In fact, any image, painted by an artist by hand, will be much more likely to have such conceptual inconsistencies. A shader designer's job is to replicate these inconsistencies to obtain the desired look-and-feel.

In current physically-based rendering and shading practice, the only recourse is to use ad-hoc work-arounds such as negative lights or negative material colors. Such ad-hoc solutions often create more problems for designers than they solve. Since negative lights or negative material colors are not really intuitive as control parameters, we end-up with a partial yet complicated solution to a very simple problem.

What we demonstrate in this paper is that it is possible to to obtain a solution to the illumination problem posed by Figure 2(b), if instead of addition and multiplication and their inverses, we chose operations that satisfy *partition of unity* such as the mixing operation $r = a(1-t) + bt$. In this case, by just choosing $a = r_0 = 214$ and $b = r_1 = 200$, we obtain the desired result using only colors defined with positive real numbers. Let us also compare and contrast the partition of unity or Barycentric formulation with the existing vector algebraic approach using a color space with only two color channels, red and green as shown in Figures 3 and 4.

Figure 3 shows an affine vector algebraic equation in the form $r = r_0 + r_1 t_r$ and $g = g_0 + g_1 t_g$. Since (t_r, t_g) corresponds to the red and green channels of the incoming illumination, which are energies, both t_r and t_g must be positive real numbers. By varying t_r and t_g we can span an infinite range of colors based on the strength of light energy. There are two problems with this type of equation that are demonstrated by this simple case. (1) The color vector (r_1, g_1) does not provide much control over what types of results can be obtained, in fact the range of colors is infinite. (2) When there is no incoming illumination, we cannot get colors where either $r < r_0$ or $g < g_0$.



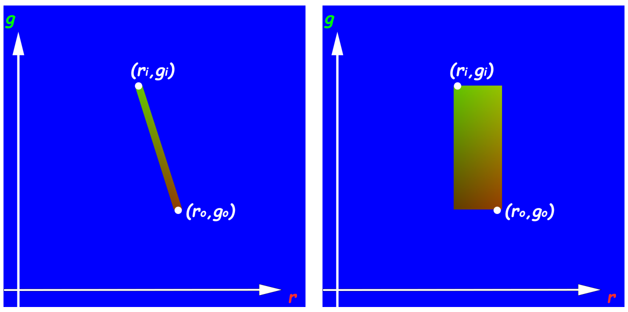
(a) Coefficients of vector algebraic equations in shaders are vectors with positive real elements.

(b) The parameters (t_r, t_g) can be any positive real numbers, permitting results over the entire non-blue region.

Figure 3: *An example of colors that can be obtained by vector algebraic equations in a red-green color space. In this case, we use the affine vector algebraic equation $r = r_0 + r_1 t_r$ and $g = g_0 + g_1 t_g$ with positive real terms. We can cover an infinite region, but we cannot get colors where either $r < r_0$ or $g < g_0$.*

On the other hand, Figure 4 shows a barycentric equation in the form $r = r_0 w_{r_0} + r_1 w_{r_1}$ and $g = g_0 w_{g_0} + g_1 w_{g_1}$. Since we work with energies, all terms are still positive reals. However, since the equation has to be barycentric, the weights are constrained such that $w_{r_0} + w_{r_1} = 1$ and $w_{g_0} + w_{g_1} = 1$. Therefore, (w_{r_0}, w_{g_0}) and (w_{r_1}, w_{g_1}) do not really represent incoming illumination, but instead, are parameters obtained by mapping incoming illumination between 0 and 1. If we choose $w_{r_1} = w_{g_1}$ this equation provides the convex hull property as shown in Figure 4(a). if w_{r_1} and w_{g_1} are not constrained to be equal, solutions can span the box defined by (r_0, g_0) and (r_1, g_1) and shown in Figure 4(b). There are two advantages to this type of equation that are demonstrated by this simple case. (1) We now have complete control over what colors can be obtained, i.e. they must lie within the convex hull or the rectangular region defined by the control colors. (2) The point (r_0, g_0)

does not have to be the lower left corner of the box, so that missing illumination can possibly produce larger numbers, as we wanted.



(a) Convex hull property obtained by $w_{r_1} = w_{g_1}$. (b) Box property obtained by non equal w_{r_1} and w_{g_1}.

Figure 4: *Colors that can be obtained by a barycentric equation in the form of $r = r_0 w_{r_0} + r_1 w_{r_1}$ and $g = g_0 w_{g_0} + g_1 w_{g_1}$, subject to $w_{r_0} + w_{r_1} = 1$ and $w_{g_0} + w_{g_1} = 1$, and where all terms are positive real numbers. The blue regions indicate colors that cannot be obtained by the equation.*

3. Historical Perspective

The problem encountered in working with colors as algebraic elements is not unexpected mathematically, since the addition operator and its inverse are not closed over the positive real numbers. This was a noted historical problem that was solved by the introduction of zero and negative numbers, and eventually lead to the familiar algebra of real numbers. Actually, real algebra was the first vector algebra, where numbers do not correspond to positions on an axis, but instead to the difference of two positions. This is true even in the one dimensional case.

If we relate this to shader development, when we use the addition operator, we cannot avoid negative numbers and all the terms are conceptually vectors, i.e. the difference between two positions, Therefore, none of the terms in the equation of $r = a + bt$ behave like colors, in the sense of being positive real values. In actuality, they are really the difference of two colors, even when there is only one color channel.

This issue is also reminiscent of another historical example; finding roots of cubic equations [Tal04]. By allowing square roots of negative numbers, Tartaglia was able to solve certain kinds of otherwise unsolvable cubic equations [Gui30]. That Cardano called complex numbers "fictitious" was not a problem, since a cubic equation with real coefficients always has at least one real root.

Analogously, in shaders as they are currently conceived, all of the energy related terms, such as colors, actually are relative entities that must be viewed as the difference between two positive real numbers. If we do not allow negative numbers, our solution space becomes quite awkward and, in most cases, we cannot even find a solution. This is equivalent to not allowing complex numbers in the solution process of cubic roots.

One legitimate approach would be to embrace the current underlying algebra of shaders by letting our algorithms manipulate negative numbers along the way. As we have described earlier, this is the current status resulting from physically based rendering approaches. In this case, there are no absolute colors corresponding to energies. What we currently call colors should be thought of as relative colors, which can be negative during the shading computation. But, once the computation is finalized we need to make sure to convert relative colors into absolute colors. This creates a conceptual confusion, as the tendency is for people to view all of these entities, and not just the final result, as absolute colors.

Fortunately, there is a simpler solution to this problem in this particular case: barycentric algebras [RS90, CR12]. There is precedent for the use of operators from a barycentric algebra in shaders. For instance, Gooch shading is based on a barycentric formulation [GGSC98]. Using a barycentric algebra in shaders does not even require a significant conceptual change. We must only restrict shader operators to be of the form

$$c = \sum_{i=0}^{M} \omega_i C_i \quad \text{where} \quad \sum_{i=0}^{M} \omega_i = 1 \quad \text{and} \quad \forall \omega_i \geq 0$$

where the C_i's are colors, i.e. n-tuples of positive real numbers. The restriction that the weights ω_i are all positive and sum to 1 is called the partition of unity property, which guarantees that solutions c stay inside of the convex hull defined by the colors C_i. This restriction does not impose any limit over the polynomials we can use. In fact, basis functions of most widely used parametric polynomials in geometric modeling, such as Bezier, B-splines or β-splines, satisfy this property. For instance, if we choose $\omega_0 = (1-t)$ and $\omega_1 = t$ with $0 \leq t \leq 1$, we obtain the mix operator, which utilizes basis functions of first degree Bezier curves. Similarly, it is easy to see that the degree zero B-spline basis functions, namely,

$$N_{i,0}(t) = \begin{cases} 1 & \text{if } t_i \leq t < t_{i+1} \\ 0 & \text{otherwise.} \end{cases}$$

satisfy partition of unity and each t_i is called a knot. Moreover, higher degree B-spline basis functions

$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t),$$

that are obtained by the Cox-de Boor formulation also satisfy partition of unity. More importantly, any parametric rational or irrational polynomial or piecewise polynomial can be converted to a form that satisfies the partition of unity property [BB87]. In addition, the convex hull property, which comes with partition of unity, is particularly useful in practical shader development applications, since it provides an intuitive control mechanism for obtaining desired results.

Based on this discussion, the extension may seem to be straightforward and it could be implemented by adapting primitives, like those used in geometric modeling, into shader development. This observation is not wrong, however there are practical aspects of shader development that differentiate this problem from modeling curves and surfaces. In this paper, we provide effective implementation approaches for these three aspects of shader development.

4. Rendering Architecture

With the Barycentric formalization, we cannot obtain non-polynomial functions such as exponential, logarithm or cosine. These functions are needed for shader implementations that are related to geometry such as implementation of displacements, manipulation of normal vectors or descriptions of procedural textures. This type of information is only needed when computing global or local illumination. Therefore, if we separate shaders into two types, which we will call front and back-end shaders, we can effectively develop a shading architecture that provides the best of both worlds.

- Front-end shaders: These are vector algebraic shaders that are used to compute and manipulate geometry related information such as displacements, normal vectors, and angles. These shaders can be constructed like classical shaders using any function or any operation. In other words, we can use the full power of the shade tree concept at this level and allow manipulation with negative and even complex numbers. For instance, if the result of an operation using $\cos\theta$ turns out to be negative, we can keep it negative. In our architecture, these shaders only produce parameters for use by back-end shaders, which actually compute colors. Only when we output these parameters do we need to make a conversion.
- Back-end shaders: These shaders are constructed using only barycentric operations, and are used to compute colors based on parameters that are passed from front-end shaders. Therefore they guarantee that from a color we can only obtain other colors. A back-end shader that is designed with a particular visual style in mind, guarantees consistent provision of that style, regardless of how parameters are computed in a front-end shader.

There can be a wide variety of parameters created by front-end shaders. For conceptual simplicity, it is better to constrain these parameters to be positive real numbers. In the rest of paper, without loss of generality, we will further restrict them to be n-tuples of positive real numbers between 0 and 1. Permitting only n-tuples is of practical use since common color formats, such as RGB, can be used as illumination parameters. For example, $c = (r, g, b) \in [0,1]^3$, can be used to represent incident energy at three primary wave lengths. The number of color channels used can be varied from three without changing the overall structure. Restriction of numbers to the range 0 to 1 is not a problem, since any set of real numbers can always be mapped onto $[0,1]$. The simplest of such mappings is *clamp*, which is defined in Renderman as:

$$clamp(t, \max, \min) = \begin{cases} 1 & \text{if } \max \leq t \\ \dfrac{t - \min}{\max - \min} & \text{if } \min \leq t \leq \max \\ 0 & \text{otherwise.} \end{cases}$$

For instance, the red color channel $R_0 = 214$ in our earlier example can be clamped to r_0 using

$$r_0 = \frac{R_0 - \min}{\max - \min} = \frac{214}{255} 0.84,$$

with $\min = 0$ and $\max = 255$. For high-dynamic range colors, of course, non-linear tone mappings are more appropriate [RPK*12]. Depths and distances are not colors and they can be significantly large. For such cases, another mapping such as $1/(d+1)$ can be used where d is either actual depth or distance. We assume these mappings are provided just to turn computed results into parameters. This conversion helps us to treat all parameters uniformly as if they are colors.

For conceptual simplicity, we will assume each parameter is created by a single front-end shader. This can help us to further classify front-end shaders in terms of the parameters they produce. In the following list, we give some examples:

- Diffuse Shader: A diffuse front-end shader provides a single parameter that is a lump sum of all diffusely reflected light that reaches a given point. This can include illumination coming from lights, ambient occlusion, environment illumination or final gathering.
- Specular Shader A specular front-end shader provides a single parameter that is a lump sum of all specularly reflected illumination for a given view direction. Note that Phong reflection and true mirror reflection are not really compatible. A shader designer must decide how to compute this lump sum parameter from these, not necessarily consistent, elements.
- Silhouette Shader: A silhouette front-end shader provides information about the distance from the shading point to the silhouette edge of an object from a given view direction.
- Depth Shader: A depth front-end shader provides information about the distance from a view-point to the shading point.

There can, of course, be many other styles of front-end shaders such as refraction, Fresnel, and caustic shaders. Note that specular and diffuse output parameters will be colors, at least 3-tuples of positive real numbers between 0 and 1. On the other hand, silhouette and depth output parameters are single dimensional.

This structure allows non-realistic global illumination with artistic control. For example, we can obtain reflection or refraction of a silhouette edge through ray tracing. Figure 5 demonstrates that we are able to obtain a specific type of mirror reflection by including shaders in the global illumination stage. Such global illumination effects cannot be obtained during a post-processing stage. This approach is not limited to only one type of rendering. For instance, it is possible to include non-realism into particle systems by controlling the colors of the particles using barycentric shaders. For instance, particles reflected in the view direction from a shading point closer to a silhouette edge most likely carry the color of the silhouette edge that is described by a back-end shader.

An advantage of this shading architecture is that it can be re-used for other scenes to obtain the same visual style without major changes. Figure 6 shows two well-known computer graphics objects rendered in a Chinese Painting style using the same shader network used to produce Figure 5. In the next section, we present the usage of our barycentric framework in practice.

5. Barycentric Shaders

Although Barycentric shaders defining colors and parametric shapes defining geometry originate from the same theoretical framework, there are significant practical differences between

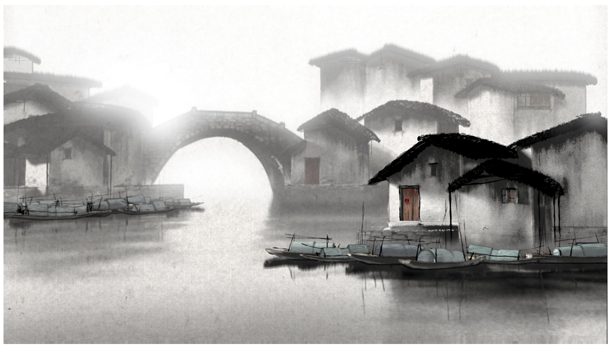
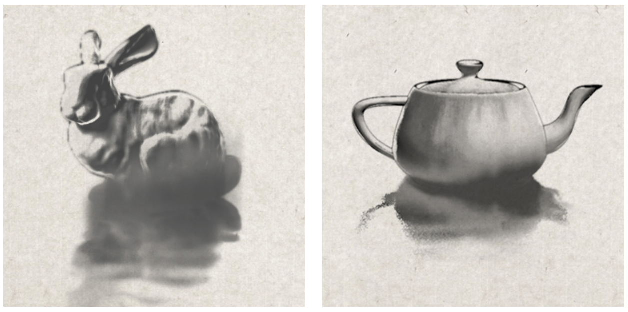


Figure 5: *A Chinese Painted 3D Scene with river reflections obtained using our shading concept. The visual style is inspired by the paintings of Yang Ming-Yi, a contemporary Chinese landscape painter.*



(a) Chinese painted Stanford Bunny. (b) Chinese painted Teapot.

Figure 6: *Two examples of reusing the shader in Figure 5 for rendering well-known computer graphics objects.*

them. We can classify these differences in the following broad categories:

1. In shaders, every surface or volume point behaves differently. Therefore, shader functions must be parameters of either texture coordinates (u,v), that correspond to surface positions, or volumetric positions (x,y,z).
2. In shape modeling, parameters are single dimensional. For instance, a parametric surface is defined by two single parameters. On the other hand, shading parameters are usually colors, i.e., n-tuple positive numbers.
3. For modeling objects, the entire parameter space is used to compute shapes. On the other hand, in rendering we construct only a subset of allowable colors.
4. In shape modeling, the number of parameters can be at most three; one for curves, two for surfaces and three for volumes. On the other hand, the number of independent shading parameter can be much higher than three.

Therefore, parametric curve and surface methods in geometric modeling cannot be directly used in the implementation of barycentric shaders. For implementation there is a need for practical solutions resolving these differences. In the rest of this section, we present these solutions.

5.1. Style Control with Control Images

We view barycentric shaders as if they are barycentric operations on texture images. This view is necessary since every shading point can have a different material property. These material properties can be described by a set of texture images that are used as control images of barycentric shaders. The function $I : [0,1]^2 \rightarrow [0,1]^3$ will denote a texture image as $c = I(u,v)$ and it will simply be denoted as I. Here, “two” corresponds to the dimension of the texture space without loss of generality; it can be increased to three or decreased to one without changing the overall idea.

We assume that the final assignment of a color for a given point, for a given view direction, is done by functions that provide the convex hull property. The resulting colors are always computed as a weighted average of control colors. As a result, a function that describes shading, i.e. how colors must be computed in every point on a given surface, can simply be given as a weighted average of a set of control images as

$$I = \sum_{i=0}^{M} \Omega_i I_i,$$

where I_i's are control images and Ω_i's are weight images that satisfy partition of unity, i.e

$$\sum_{i=0}^{M} \Omega_i = \mathbf{1},$$

where $\mathbf{1}$ is a white image and I is the final rendering. Note that this is usually a five dimensional formula that includes two texture dimensions and three color dimensions. Figure 12 provides a simple example how final rendering is computed using the weighted average of two control images as I_0 and I_1. Note that in this case if we choose one of the weight images, say Ω_1, the other one is well defined as $\Omega_0 = \mathbf{1} - \Omega_1$. In this case, obtaining $\mathbf{1} - \Omega_0$ is easy, since it is just the inverse image of Ω_0. Note that since Ω_0 can be a color image it is not possible to obtain this using Ω_0 as an opacity map. The two corresponding pixels in these two control images in this example are not simply scaled versions of each other. Their hues are also different. White backgrounds in both images are also part of the computation, but, since the colors are the same we always get a white background.

To control style, the most important component is the choice of control images. As shown in Figure 12, the two control images define how the final rendering will look. The weight images only define the level of illumination. If the control images are meaningful, any set of weight images that satisfy partition of unity can result in an image in the same visual style. This is demonstrated in Figure 8. Although, the final rendering in Figure 8 does not look as three-dimensional as in Figure 12, the resulting image still appears to be illuminated. In this case, we did not use any special treatment to obtain an acceptable result. Any image that we used as a weight provided similar visual results for these control images.

The Figure 8 demonstrates the critical importance of control images for obtaining a consistent color scheme in the final image.

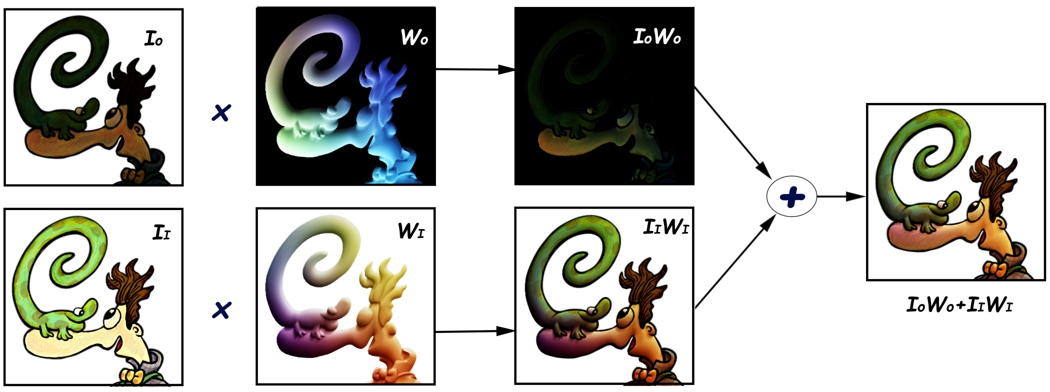
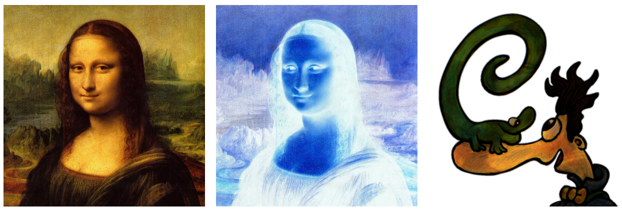


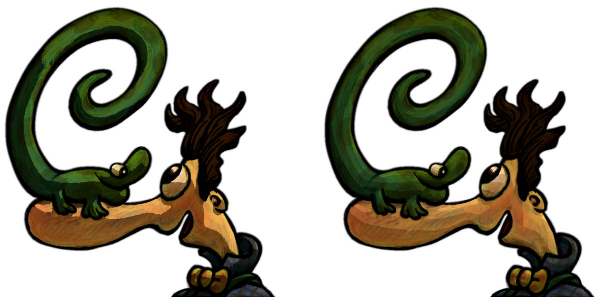
Figure 7: *An example demonstrating the concept of control and weight images. I_0 and I_1 are control images, Ω_0 and Ω_1 are weight images that satisfy partition of unity. $I = I_0\Omega_0 + I_1\Omega_1$ is the final rendering obtained by taking a weighted average of the two control images.*

Styles can further be controlled by basis functions that are used to compute weight images. Basic functions control how colors are distributed across the final image.



(a) Ω_1: Mona Lisa as a weight image. (b) Ω_0: Inversion of Mona Lisa. (c) Final rendering.

Figure 8: *An example that shows the importance of control images in obtaining a style. Even a meaningless illumination, in this case we used the Mona Lisa as a weight image, can result in an acceptable image in the same visual style.*



(a) Two zero degree B-spline basis functions with uniformly distributed knots at $t_0 = 0.0$, $t_1 = 0.5$ and $t_2 = 1.0$. (b) Four zero degree B-spline basis functions with uniformly distributed knots between 0 and 1.

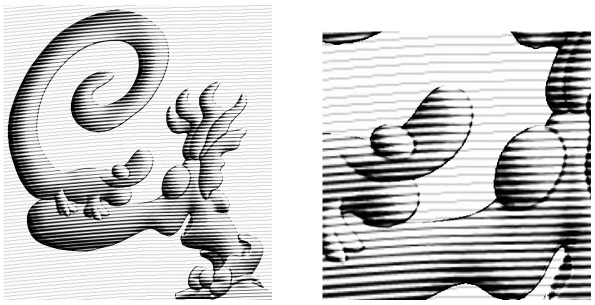
Figure 9: *A comparison of the effect of the number of discontinuities in changing the final style. To create consistency, we obtained the two new control images needed in (b) by using linear interpolation of the original control images.*

5.2. Style Control with Basis Functions

Similar to weights ω_i, weight images Ω_i are computed from basis functions and parameters. For instance, first degree Bezier basis functions are $\omega_0 = 1-t$ and $\omega_1 = t$. If we replace, 1 with a white image **1** and t with a parameter image T, we obtain the first degree Bezier basis functions for shading as $\Omega_0 = \mathbf{1} - T$ and $\Omega_1 = T$. In this equation T can be any image that describes a particular illumination effect such as diffusely reflected illumination that reaches every point on the surface.

We have observed that some important elements of visual style are the functional continuities underlying the parametric function. For instance, an everywhere G^2 continuous function appears significantly different "in style" than a function that is G^0 discontinuous in some regions. Similarly, the number of discontinuous regions, their shapes and sizes also changes our perception of style. Since our control texture images can be discontinuous everywhere, we cannot really evaluate the continuity of control images. On the other hand, we can always analyze properties of the basis functions that are used to create the final images. For instance, consider the two renderings presented in Figure 9. Although these images are discontinuous everywhere, it is easy to see in (a) there are only two and in (b) there are four distinct regions. These perceived regions result from the number of zero-degree B-splines. Note that when the number of zero degree basis functions increases, the final image approaches a linear interpolation.

This particular example demonstrates that discontinuities introduced by basis functions are helpful in designing a style, since discontinuities and continuities introduced by basis functions can be more visible than texture discontinuities in control images. In other words, artists can simply control style by changing the formulation of the underlying parametric function. Choice of basis functions is also important for some applications that require discontinuity, such as the crosshatching example shown in Figure 10.



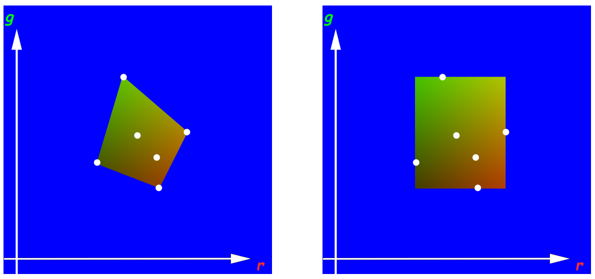
(a) Crosshatching with zero degree B-splines.

(b) Detail.

Figure 10: *Nine zero degree B-spline basis functions with uniformly distributed knots using nine control images consisting of thicker and thicker lines.*

5.3. Rectangular Box Property

If weights for a given shader point (u,v) are unsaturated, i.e. greyscale colors, then we obtain a convex hull property exactly like that in geometric modeling (See Figure 11(a)). On the other hand, if weights are just random colors, each dimension gives a convex hull property independently from each other. This results in a richer range of color possibilities, as shown in Figure 11(b). We call this the rectangular box property.



(a) Convex hull property obtained by choosing unsaturated colors as ω values.

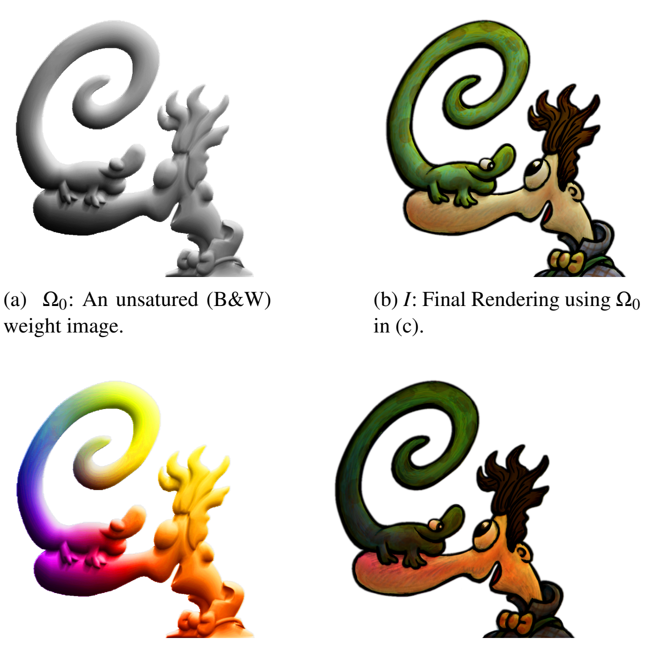
(b) Box property obtained by using random colors as ω values.

Figure 11: *White points show control colors for a given shader point (u,v). Blue regions are the ones that cannot be obtained by the equation.*

Figure 12 provides a comparison of results using unstaturated vs. saturated weight images. As can be seen in this example, saturated weights provide a wider distribution of colors without sacrificing visual style.

5.4. Painter's Hierarchy

In this paper, we also introduce a painter's hierarchy for handling a high number of shading parameters. As discussed earlier, the number of parameters can be very high (e.g. Diffuse, Specular, Refraction, Fresnel, Caustic, Depth and Silhouette). If we consider all of these parameters in a general parametric function, it will be very hard to keep track of control images since there would be a need for too many of them. Our solution to this problem is to create a hierarchy similar to a painter's order. A painter first creates a base image that corresponds to the most essential part of the painting, its system of values. This corresponds roughly to diffuse reflection. The painter then adds other effects such as specular highlights, silhouette edges, outlines, caustics or shadows. We also create a hierarchy of parameters, starting with the diffuse parameter. Let us denote the diffuse rendering as I_0, the next rendering I_1 is computed as a weighted average of I_0 and another effect image I_{F_1}, e.g. reflection and refraction combined by Fresnel, as



(a) Ω_0: An unsatured (B&W) weight image.

(b) I: Final Rendering using Ω_0 in (c).

(c) Ω_0: A very saturated weight image.

(d) I: Final Rendering using Ω_0 in (c).

Figure 12: *A comparison of convex hull vs. box property. A greyscale weight image provides a convex hull. On the other hand, a very saturated weight image extends color possibilities to a bounding box.*

$$I_0 = I_0\Omega_{1,0} + I_{F_1}\Omega_{1,1}.$$

We can continue this way to obtain final image I_n as

$$I_n = I_{n-1}\Omega_{n,0} + I_{F_n}\Omega_{n,1}.$$

If all effects I_{F_n}'s are computed by a barycentric formula, combination of all these equations is guaranteed to provide a barycentric formula.

The painter's hierarchy significantly reduces the complexity coming from a high number of parameters. One property that can further simplify shader development is that, except for diffuse, most other parameters do not really require full textures. Most of them can be described using a single color. For instance, we do not usually have to change the color of a silhouette along the silhouette boundary. If we want to draw silhouette edges like cartoonists, this color is usually a single dark color. If we want to get a rim light effect, the resulting silhouette brightening is usually a single light color. Some parameters, such Fresnel, do not need their own color, they are used to mix the others, namely refraction and specular reflection.

6. Design Process and Results

We have tested this approach over three semesters in a graduate rendering and shading class. Each student in the class chooses an artist's painting to mimic, and then creates still-life animations strongly resembling that painting. Examples have been shown in Figure 1 and additional examples are in Figure 13). As demonstrated in these examples, it is possible to obtain a wide variety of styles using this approach. The process of converting a style in a painting into a Barycentric shader requires analyzing the chosen painting carefully.

The most critical issue is to identify the painter's hierarchy along with the effects used in the painting. For instance, one painter might have first drawn outlines and later added colors in such a way that the outlines are still visible under the paint. Another painter might first paint the diffuse illumination, add specular highlights, and then add silhouette edges and outlines. The structure of the shader must follow exactly the hierarchy of painting the painter employed. Some painters may not use certain effects. For instance, many painters do not emphasize outlines and do not draw silhouette edges. Leonardo Da Vinci's *sfumato* style did not use specular highlights. If any particular effect is not used in a desired style, our shader must not provide it. This process of analysis and construction results in a simple tree structure, in which each leaf node corresponds to a particular effect and other nodes correspond to a Barycentric operation that simply mixes colors.

Once the painter's hierarchy is established, the next issue is to establish barycentric operations and control colors. To make the process simple, we assume that all surface colors result from material properties, i.e. all lights are white. Each barycentric operation is a mixture of n colors. Therefore, the first step is the identification of the number n, which is usually 2, but in some cases can be 3 or higher in diffuse reflection effects. The second step is to identify appropriate barycentric functions. For cartoon shading, we use zero-degree B-splines to get flat shading. For smoother color changes, we use first-degree B-splines, quadric or cubic Bezier curves. The third step is the identification of control colors. Since all other key decisions have already been made, this identification is done relatively easily by carefully checking the colors that have been used in painting every object in the painting.

Using this process, it is easy to obtain the overall look-and-feel of the painting being used to guide visual style. Of course, the other issues such as shapes of the objects, positions and colors of lights and patterns of textures, must be continuously improved until a close resemblance to the example painting is obtained. Copyright protection prevents us from including the original paintings used by the students to develop their animations, however we have provided the names of the artists and their paintings to facilitate an internet search to make a qualitative visual comparison with their examples.

It is, of course, hard to convince all students to follow this procedure initially. Typically, students who have some previous experience with shader development try to follow a physically based approach. Since there are always such students in a class, we always have an unofficial control group. We show them how to use negative materials and negative colors to obtain a desired style using physically based shaders. It is our observation that despite very hard work, they fail to obtain the desired styles. After this experience, they eventually begin to use our method. We believe that this approach, if it is adopted by the industry, can save a significant amount of time and resources.

To make adoption easier, there is a need for the development of some off-the-shelf basic shader structures that will hide all of the mathematical complication from users. Users should only need to provide control textures or colors. This approach is reminiscent of parametric curves and surfaces. Most users that draw a curve in a vector drawing package do not have any idea about the underlying mathematics. On the other hand, once they choose a tool they quickly start to feel how the control points shape the curve. Therefore, the most important issue in practice is to turn these ideas into an intuitive user interface for novice users.

7. Conclusion and Future Work

In this paper, we have presented a simple mathematical approach to art directed shader development. This approach provides shader developers an intuitive process, giving them a high level of visual control in the creation of stylized depictions. In our approach, the shader functions are parametric functions that satisfy the partition of unity, a concept that is widely used in Computer Aided Geometric Design. The paper makes three contributions:

(1) We define barycentric operations over positive real colors to guarantee that results will always be positive real colors. With this formalization, we redefine shader functions as parametric functions that satisfy the partition of unity, a concept that is widely used in geometric modeling. In geometric modeling, parametric functions with this property, such as Bezier or B-spline functions, guarantee that shapes stay inside of the convex hull defined by control vertices. This property provides an intuitive interaction mechanism for obtaining desired shapes. Similarly, with our shader functions, we can provide intuitive interaction by guaranteeing that colors (or texture images) will always stay inside of the convex hull of control colors (or texture images).

(2) To obtain colored light effects, we allow computation separately along each color channel. This extension provides the convex hull property for each dimension independently, and results in a more relaxed, rectangular box property without significantly changing visual style. This formalism can be particularly helpful to artists, who may not have mathematical training, by simplifying shader development for obtaining desired expressive styles. The new approach also provides a clear differentiation between illumination and shading. The terms that are related to local illumination such as surface normals and angles conceptually move into the illumination part of the rendering and shading pipeline.

(3) We have also provided a well-tested procedure to obtain desired styles. To develop a specific shader for a given shader function, all artists need to provide a set of texture-mapped control images,

which act like control points of parametric functions. With these shader functions, final renderings are guaranteed to be weighted averages of the control images. The resulting styles are also controlled by the choice of shader functions. For instance, zero-degree B-splines are appropriate for styles such as cartoon or crosshatching, since they can support discontinuity. Similarly, first-degree B-splines support linear color changes with derivative discontinuities. Given this underlying structure, it is easy to provide an intuitive interface that allows the artist choose their style. As a result, the artist's work is simply deciding on a shader function to obtain the desired style and providing control images or textures to obtain the desired color distribution.

We have focused our attention on the actual working process that lighters use in film production. They really do use negative lights and other hacks like that, and are invariably working in RGB space. On the other hand, it is possible to use these ideas in other color spaces designed to support interpolation, such as Lab space. We want to emphasize that no matter what color space one is working in, this method can be used to guarantee that colors stay in the display gamut, as long as the convex hull vertices lie in the display gamut. Although RGB is just one example space, it is particularly easy to work with, since the display gamut is a unit cube. For other color spaces, there is no guarantee that colors will project back into the display gamut when we convert back to RGB in order to build an image.

In this framework, what is usually considered as shader development goes into computing shading parameters. This approach guarantees that the overall look of resulting 3D renderings are not significantly affected by how the shading parameters are computed. For instance, if shadows are not included in the computation of the diffuse parameter, shadows disappear but the overall look and feel of the rendering does not really change. This predictability is especially useful for artists, enabling them to be able to focus mainly on final appearances.

This paper only deals with non-realistic illumination and shading. We assume that the images are still representational and the shapes and cameras are still realistic. For truly non-realistic paintings and illustrations, we would also need non-realistic shapes and cameras. We are currently investigating the shape problem and are developing a new shape representation for modeling impossible or non-realistic shapes. There is also need for a formalization of non-realistic cameras, which we are also currently working on. It is our hope that these three approaches will provide a solid foundation for the formalization of non-realism in 3D modeling, rendering, and shading.

References

[AGB00] Apodaca A. A., Gritz L., Barzel R.: *Advanced RenderMan: Creating CGI for motion pictures*. Morgan Kaufmann, 2000. 1

[BB87] Beatty J. C., Barsky B. A.: *An introduction to splines for use in computer graphics and geometric modeling*. Morgan Kaufmann, 1987. 4

[Blo14] Block B.: *The visual story: creating the visual structure of film, TV and digital media*. CRC Press, 2014. 1

[CCC87] Cook R. L., Carpenter L., Catmull E.: The reyes image rendering architecture. In *ACM SIGGRAPH Computer Graphics* (1987), vol. 21, ACM, pp. 95–102. 1

[Coo84] Cook R. L.: Shade trees. *ACM Siggraph Computer Graphics 18*, 3 (1984), 223–231. 1

[CR12] Czédli G., Romanowska A. B.: An algebraic closure for barycentric algebras and convex sets. *Algebra universalis 68*, 1-2 (2012), 111–143. 2, 4

[Ebe03] Ebert D. S.: *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003. 2

[GGSC98] Gooch A., Gooch B., Shirley P., Cohen E.: A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM, pp. 447–452. 4

[GKDS12] Georgiev I., Křivánek J., Davidovic T., Slusallek P.: Light transport simulation with vertex connection and merging. *ACM Trans. Graph. 31*, 6 (2012), 192. 2

[Gui30] Guilbeau L.: The history of the solution of the cubic equation. *Mathematics News Letter* (1930), 8–12. 4

[HJB*12] Hachisuka T., Jarosz W., Bouchard G., Christensen P., Frisvad J. R., Jakob W., Jensen H. W., Kaschalk M., Knaus C., Selle A., et al.: State of the art in photon density estimation. In *Acm Siggraph 2012 Courses* (2012), ACM, p. 6. 2

[HL90] Hanrahan P., Lawson J.: A language for shading and lighting calculations. *ACM SIGGRAPH Computer Graphics 24*, 4 (1990), 289–298. 1

[HPJ12] Hachisuka T., Pantaleoni J., Jensen H. W.: A path space extension for robust light transport simulation. *ACM Transactions on Graphics (TOG) 31*, 6 (2012), 191. 2

[LKM01] Lindholm E., Kilgard M. J., Moreton H.: A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 149–158. 2

[MGAK03] Mark W. R., Glanville R. S., Akeley K., Kilgard M. J.: Cg: a system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)* (2003), vol. 22, ACM, pp. 896–907. 2

[Ols98] Olson R.: *Art direction for film and video*. CRC Press, 1998. 1

[pix15] Pixar's animation process, 2015. 1

[PMTH01] Proudfoot K., Mark W. R., Tzvetkov S., Hanrahan P.: A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 159–170. 2

[RLKG*09] Rost R. J., Licea-Kane B. M., Ginsburg D., Kessenich J. M., Lichtenbelt B., Malan H., Weiblen M.: *OpenGL shading language*. Pearson Education, 2009. 2

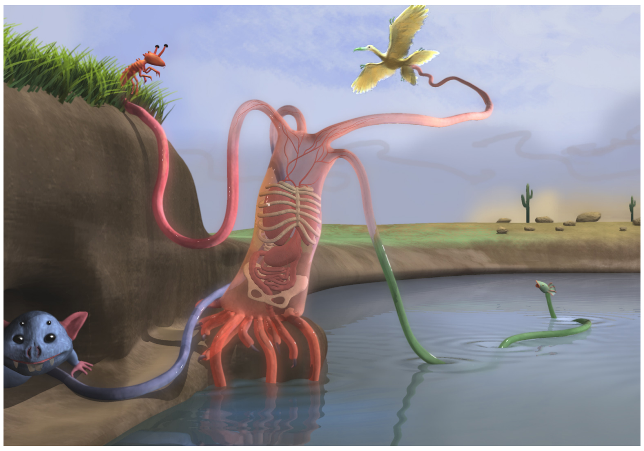
[RPK*12] Reinhard E., Pouli T., Kunkel T., Long B., Ballestad A., Damberg G.: Calibrated image appearance reproduction. *ACM Trans. Graph. (Proceedings of SIGGRAPH Asia) 31*, 6 (2012). 5

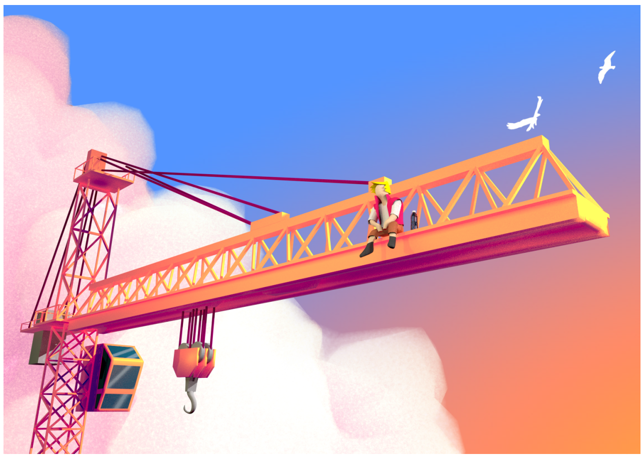
[RS90] Romanowska A. B., Smith J.: On the structure of barycentric algebras. *Houston, Journal of Mathematics 16*, 3 (1990), 431–448. 2, 4

[Tal04] Tall D.: Building theories: The three worlds of mathematics. *For the Learning of Mathematics* (2004), 29–32. 4

[UAD*90] Upstill S., Achjadi J., Damais A., Jumena N. S., Hardjonagoro K., Ichsan F., Pieters P., Siswandi R., Sosrowardoyo T., Tirta I., et al.: *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, vol. 226. Addison-Wesley, 1990. 1

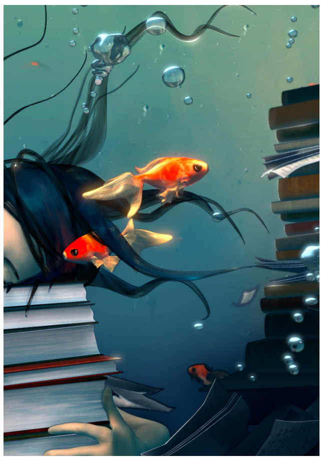
[Wil06] Willis P.: Projective alpha colour. In *Computer Graphics Forum* (2006), vol. 25, Wiley Online Library, pp. 557–566. 2

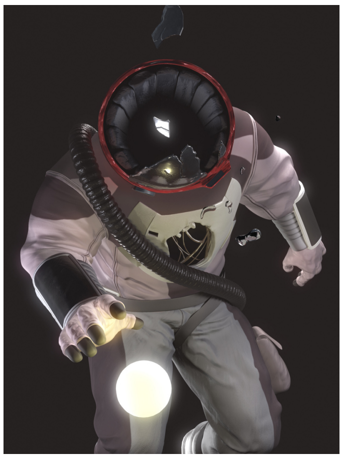

(a) by Chethna Kabeerdoss after an illustration by Rachel Cunningham Wang.


(b) by Justin Hollis after an illustration by Sylvain Sarrailh (aka Tohad).

(c) by Dustin Han after an illustration by Sam Nielson.

(d) by Matthew Suarez after a still Life painting by Pieter Van Claesz.


(e) by Yolanda Cheng after a painting by Cyril Rolando.


(f) by Egan Conrad after an illustration by Chun Lo.

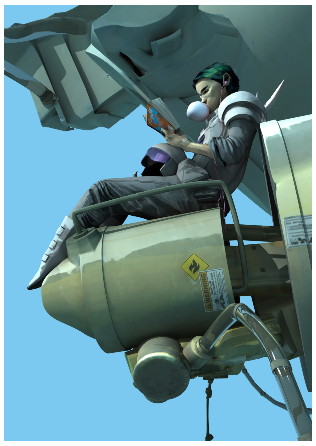

(g) by Schaefer Mitchell after an illustration by Fiona Staples.

Figure 13: *More example frames from class projects that were created using our method for providing intuitive art-directed control to obtain a wide variety of expressive styles. Each of these are from short animations.*