Shader Reparameterization
Colin Braley*
University of Virginia

Abstract

Digital artists spend much of their time manipulating real-valued GUI sliders in computer animation software. Each slider corresponds to some parameter of an analytical model that determines the appearance of a virtual object. For a given object, we call the set of all parameters controlling its visual appearance a parameter space.

Shaders are a common tool in computer graphics to model the visual characteristics of a virtual object. Shaders are small programs which determine the color of each pixel on an object. The resulting image of a shader is driven by a parameter space controlled by an artist. The goal of the artist is to search the high dimensional parameter space of a shader until the desired look is achieved.

Searching this parameter space may be very time consuming due to its high dimensionality. Furthermore, shaders are not always parameterized in a way that is meaningful to the artist. For example, shaders are often written by engineers with a background vastly different than the artists who will use them. An engineer might think it is natural to parameterize a noise shader in terms of “noise frequency” whereas an artist might prefer this parameter to be called “smoothness.”

We present a system and an analysis of the problem of reparameterization. Our system allows users to add their own, personally meaningful parameters to shader and then be able to modify the result based on their own parameterization. This has the benefits of allowing users to specify meaningful parameter names, as well as potentially reducing the dimensionality of the parameter space.

In our system we implement and analyze a new novel technique called linear shader reparameterization. We also discuss (but do not implement) perceptually uniform shader reparameterization and non-linear shader reparameterization.

Source code for our system can be checked out from SVN via https://subversion.assembla.com/svn/ezshade/.

Keywords: shaders, programming languages, awesome

1 Introduction and Problem

1.1 Problem and Motivation

Trained digital artists spend much of their time searching parameter spaces. In 3D modeling and animation software, engineers typically provide parameterized analytical appearance models which artists can adjust to change the visual output. For example, a 3D rendering program might expose two numerical parameters controlling the reflectance of an object: glossiness and matte-ness. An artist would then adjust the numbers corresponding to these parameters until the desired visual result is achieved. Artists colloquially refer to this as slider twiddling, coming from the slider bars provided in many GUIs to help them tweak numeric parameters.

This process is slow and painstaking for 2 reasons. First, many of these parameter spaces have high dimensionality which makes the searching process slow. Shader programmers seek to create very general highly-parameterized models, and the resulting high dimensional parameter space may not be intuitive for the artist end-user. Secondly, many shader parameters are not named intuitively. This stems from the fact that programmers write shaders, and may find mathematically oriented parameter names suitable, whereas an artist might prefer a different naming scheme. Often, artists spend hours adjusting these numeric parameters until the desired visual result is achieved. This is a frustrating for the artist, and expensive for the artist’s employer. Clearly, if digital artists were able to search this parameter space faster they would be happier and companies would save money.

We propose a new technique family of techniques called Shader Reparameterization to solve the aforementioned problems. Our techniques allows an end-user to create new shader parameters with names that are meaningful to them. Furthermore, our technique allows artists to reduce the dimensionality of the parameter space and eliminate superfluous parameters.
1.2 Related Work

Authors in computer graphics have previously recognized the problem that searching high dimensional parameter spaces is difficult and time consuming for artists. In the seminal paper of Marks et al. [Marks et al. 1997], the concept of design galleries is presented. This fully automatic approach presents a series of images to the user, and the user at each step selects the image they want to “move towards” in parameter space. The main technical challenge here is solving what the authors term as the dispersion problem. In design galleries, it is necessary to choose an initial set of $K$ vectors from $\mathbb{R}^N$ that represent all of the potential different types of outputs that shader is capable of. A lot of insight could have been gleaned from this work, but the author unfortunately only found out about it very near to this project’s deadline.

The more recently work of Matusik et al [Matusik et al. 2003] on generative models for BRDFs. Here, the author allows the user to go through a set of measured BRDF data, and mark each BRDF with a scalar value of some quantity they name. For example, a user might look through all BRDFs and give a number for how “glossy” they are. Using this data, the author then can synthesize new BRDFs based on the users input for how glossy they would like they BRDF to be. This system is similar to our own in that the user applies what Matusik terms annotations to a high dimensional space. While this work is solving a very different problem and is only tangentially related, this is the work that inspired our own.

1.3 Mathematical Formulation

Consider a shader with $N$ parameters. The names of the parameters are given by $s_1, \ldots, s_N$. The $i$th parameter has a name, given by $s_i$. For example, in figure 1, we see that $s_1 = “\text{blueVal}”. The numeric values of the parameters are given by $x_1, \ldots, x_N$ where each $x_i \in \mathbb{R}$. In figure 1, we see that $x_0 = 0.9$. We can consider the parameter values as a vector $x \in \mathbb{R}^N$.

Considering a shader’s parameter settings as a vector in $\mathbb{R}^N$, we can think of the shader as a function $S : \mathbb{R}^N \rightarrow I$, where $I$ is the set of all images. Our goal is to allow the user to create a meaningful $M$ parameter reparameterization $F$, where $F : \mathbb{R}^M \rightarrow \mathbb{R}^N$ for $M \leq N$. In our reparameterization, we denote the new parameters values by $x_1', \ldots, x_M'$ and the new parameter names by $s_1', \ldots, s_M'$. Once the reparameterization has been given, for some input values $x \in \mathbb{R}^M$, we compute the final image from the reparameterized shader as $S(F(x))$.

2 Reparameterization Techniques

We consider several techniques for creating a reparameterization $F$. Our first technique, linear shader reparameterization has been implemented and tested. It works well for certain types of reparameterizations, but fails in some cases. We next present a technique for making a reparameterizations perceptually uniform, in that each slider is scaled appropriately. Finally, we present a technique for non-linear reparameterization. We did not implement this technique due to time constraints.

2.1 Linear Reparameterization

For linear reparameterization, we allow the user to first specify a number $M \in \mathbb{Z}^+$. This is the number of parameters in the reparameterization. The users also specify names of each new parameter: $s_1', \ldots, s_M'$. Next, the users specify $M$ unit vectors in $\mathbb{R}^N$, $v_1, \ldots, v_M$. These vectors are found by subtracting two points in $\mathbb{R}^N$ and normalizing the to unit length. Motion along a particular direction in $\mathbb{R}^N$ is associated with a new parameter name $s_i'$. This process is done manually by the user. In my software, this can be achieved by pressing “p” in the main window, and then by following the directions found when right clicking on the image. Once this reparameterization has been setup, the user can alter the new shader parameters, and ignore the old ones.

In a linear reparameterization, our new parameters are a linear combination of a set of $M$ basis vectors from $\mathbb{R}^N, v_1, \ldots, v_M$. The point in the original $N$ dimensional parameter space is given as a linear combination of $M$ vectors in that original parameter space. Each of the $M$ vectors corresponds to a quality that is meaningful to the user. This is formalized in equation 1, which tells us how to evaluate shader $S$ given a new reparameterization with $M$ parameters. This linear combination is shown in equation 1.

$$S(\sum_{i=1}^{M} x_i' v_i)$$ (1)

When written this way, it is clear that for linear shader reparameterization $F$ is just a linear map from $\mathbb{R}^N$ to $\mathbb{R}^M$. We know from linear algebra that any linear map between finite dimensional vector spaces can be written as a matrix. Therefore, we write $F$ as follows:

$$F = \begin{bmatrix} v_1 & \cdots & v_M \end{bmatrix}$$ (2)

Consider a vector $x \in \mathbb{R}^M$ of parameter values from the reparameterization. In order to evaluate the shader, and produce an image,
we simply do the following computation:

$$S(Mx)$$  (3)

where $Mx$ is a matrix-vector multiplication.

This approach works well for many cases. For an example, see the simple case in figure 1. We also tested linear reparameterization on a Phong shader and got useful results. However, there are limitations to linear reparameterization. The first limitation is that one of the new parameters may have a stronger effect on the resulting image as compared to other parameters. This is resolved with perceptually uniform shader reparameterization. Secondly, a larger problem arises then our desired reparameterization is not linear in terms of the original parameterization. The implications of this issue, along with possible solutions, are discussed in section 2.3.

### 2.2 Perceptual Shader Reparameterization

Our ideas for perceptual reparameterization are motivated from the success of perceptually uniform color spaces [Meyer and Greenberg 1990]. We will briefly review perceptual uniformity in terms of color spaces. A $K$ dimensional color space specifies the sensation of color through a $K$-tuple of real numbers. Ideally, the Euclidean distance between two $K$-tuples in a color space should correspond to a human’s response to how perceptually different the two colors are. Colorspace in which Euclidean distance is similar to human perceptual differences are called perceptually uniform color spaces. The most common perceptually uniform color spaces are the Lab and Luv color spaces. More common colorspace, like RGB and XYZ, are not perceptually uniform. There are many reasons for which perceptually uniform colors was designed. One of the foremost of these reasons is for simple color specification. When a user is altering sliders that correspond to one of the $K$ parameters, ideally a change of $\epsilon$ in parameter 1 will alter the image by a perceptually similar amount to a change of $\epsilon$ in parameter 2. Our goal is to reparameterize shaders in a similar way.

When the user moves each slider by a specified amount, the image should change by just as much regardless of which slider the user is changing.

Shaders produce images. To compare two images, we can use an image comparison metric. Consider two images, $A$ and $B$. The simplest image comparison metric is to consider each image as a vector of pixels, $v_A$ and $v_B$. To compare them, we simply take the $L^2$ norm of the difference between these two vectors. This is commonly called the $L^2$ image difference. Analogous schemes exist for other vector norms, but $L^2$ is the most popular. The issue here is that this image comparison metric does not align with what humans think are similar images. Consider 2 high frequency images that are the exact same with the exception of one image being shifted over by 1 pixel. Most humans would consider these images highly similar, but the $L^2$ metric would not. To combat these issues, perceptually meaningful image comparison metrics have been developed. Two such metrics are Earth-Mover’s distance (EMD) [Rubner et al. 2000], and the Structural-Similarity Metric (SSIM) [Wang et al. 2004]. These image comparison metrics are in some sense analogous to Euclidean distance in a uniform color space, like Lab or Luv. For our purposes it is sufficient to state that these metrics exist, as a full description of how they work (and when they don’t work) is outside the scope of a course project.

In an ideal reparameterized shader, altering the shader parameter values by $\epsilon$ should produce the same amount of change, regardless of which parameters were altered. If this is true, we say that a shader has a perceptually uniform parameterization. One could leverage techniques from the creation of uniform color spaces to create uniform shader reparameterizations. Uniform color spaces are created using what is often called the MacAdam ellipse data set [Wyszecki et al. 1968]. This dataset is shown in figure 3. Each ellipse refers to areas in which the colors shown are indistinguishable to a typical human observer. Since these ellipses are known, nonlinear functions can be computed that remap this color space such that all of these ellipses are circles. This remapped space is perceptually uniform.

Figure 3: MacAdam ellipse data. Ellipses correspond to regions where colors are visually indistinguishable. A small subset of the available ellipses are shown for the sake of clarity.

The MacAdam ellipse data is for a 2D colorspace (XY chromaticity). For a shader with an $N$ dimensional parameter space, an analogous process could be used to determine hyperellipses that correspond to indistinguishable output images with respect to an image comparison metric like SSIM or EMD. This process could be fast, since no human needs to be in the loop. Next, a nonlinear remapping could be computed that takes these hyperellipses to hyperspheres. Values specified in this remapped space would allow a shader reparameterization to be perceptually uniform.

While I think this idea is theoretically interesting, I am not sold as to its practical value. Implementation of this technique would be time consuming, and I am unsure as to how much value it would add to a final system.

### 2.3 Nonlinear Shader Reparameterization

Consider a shader like that shown in figure 2. This shader is extremely simply and allows the user to specify RGB values for each pixel. Using linear shader reparameterization, we can reparameterize this shader into a single parameter shader where the user controls the darkness of the output pixel. This is because darkness is linear in terms of the original shader’s parameters. However, many mappings $F$ from one parameter space to another are nonlinear! For example, you will never be able to reparameterize the RGB shader as HSV (hue, saturation, value). This is because the mapping from RGB to HSV is nonlinear. Ideally, shader reparameterizations would not have to be linear, since this would allow for more flexible reparameterizations. One way to specify a nonlinear reparameterization would be through the use of curvilinear coordinates. Instead of specifying pairs of points in $\mathbb{R}^N$ and then computing a direction vector, the user could specify a list of points, and then a spline function could be fit to this list of points. This spline could be used as a basis element in a curvilinear local basis.
2.4 System and Implementation

A prototype system demonstrating linear shader reparameterization is available with read-only svn access at the following location: https://subversion.assembla.com/svn/ezshade/. Our system is written in C++, and tested on 64 bit Ubuntu Linux. A modern graphics card is required for operation. The following libraries are also required: GNU scientific library(GLSL), GLUI, GLEW, GLUT, and libpng. CMake is used for compilation.

Screenshots from our prototype are visible in figures 1 and 2. For usage on how to launch the program, see the --help argument. Different 3D models (.obj format) and shaders(GLSL) can be specified as command line parameters. Once the program is running, press h to see information on usage, or just press p to begin reparameterization.

Our system supports a subset of valid GLSL shaders. In GLSL, variables are declared with the uniform modifier if they are modifiable on a global level(as opposed to a per vertex level). These “uniforms” are what the artist has to spend time twiddling the value of, and are what form our shader’s parameter space. In our system, only uniforms of type float are supported. Furthermore, min, max, and initial values can be given for a uniform via special syntax in the GLSL file. To do this a uniform must be declared like this: uniform float x; /\min, max, initial/.

I would like to emphasize that this project was done entirely for this course, and beyond being related to graphics, has no crossover with my actual research. I wrote a large amount of of code to support this system, not including the hundreds of lines that went into some things I didn’t have time to finish. The only code inside src that I did not write this semester, for this project is the cmd code(it was open source), the cimg code(this is a library). and some of the string handling code in Utils.h (I reused this from an old project).

2.5 Proposed Evaluation

While a full user-study based evaluation is outside of a class project, we briefly describe the evaluation that we would like to do if we could. Ideally, we would enlist the assistance of W users, each with a varying level of skill in computer graphics. Each user would be given an goal image that they are to match. The task of the user would be to tweak shader parameters over time until they feel that they have matched the input image. Each user would do this process using design galleries, shader reparameterization, and using the original shader with no modifications. We could then plot how close the user was to the goal image(in terms of an image comparison metric) as a function of time. We might learn that, while design galleries allows the user to get in the ballpark of the goal image quickly, the process is unable to ever reach the exact final image. This experiment would be useful in that it would shed light not just on which technique is “best” but on which technique is best within a given time budget.

3 Future Work

There is still much work to be done before shader reparameterization is a solved problem. The curvilinear basis approach in section 2.3 could likely be extended with an approach inspired by manifold learning. It is likely that many useful shader reparameterizations could be envisioned as movement along a manifold embedded in \( \mathbb{R}^N \).

Another problem waiting to be solved is that of “discontinuous” shaders. Consider a shader with a parameter named “z”. If z has a value < 0, the shader returns the color (1, 0, 0), otherwise, it returns (0, 0, 1). This type of shader is not well suited to reparameterization, because its output varies discontinuously with its input. To solve this problem, we think a clustering technique might be useful. For many different parameter values of a shader, this technique would evaluate the shader and get a resulting image. Using an image comparison metric, this image would be compared to images at all other points. Using this information, one could create a symmetric similarity matrix for the shader. Using this matrix one could perform spectral clustering [Ng et al. 2001], and split the shader into 2 separate shaders: one where “z” is always less than 0, and another where “z” is always greater than 0.

Following the ideas of Marks et al. [Marks et al. 1997], it would be interesting to extend our ideas to non-shaders. Many tasks in graphics correspond to tweaking parameters until an output images looks correct. In many cases, the exact techniques developed here for shaders could be applied.

Finally, it would be interesting to automate the process of searching parameter spaces. Perhaps, if the user could draw a goal image of what they want the end result to be, an hill-climbing algorithm could automatically search the parameter space until the resulting shader image matches the goal image.

4 Conclusions

In summary, we presented 2 new techniques for shader reparameterization. The first, linear shader reparameterization, we implemented and tested. Our implementation is publicly available. Next, we presented perceptually uniform shader reparameterization, but did not implement it. This technique has the advantage of making adjusting parameters more intuitive since all parameters have the same scale in terms of their perceptual impact on the resulting image. Finally, we presented some initial thoughts and first steps on how to achieve nonlinear shader reparameterization.

References


