User-Centered Design of GPU-Based Shader Programs

Kraus, Martin

Published in:
Proceedings of the International Conference on Computer Graphics Theory and Applications

Publication date:
2012

Document Version
Early version, also known as pre-print

USER-CENTERED DESIGN OF GPU-BASED SHADER PROGRAMS

Martin Kraus
Department of Architecture, Design and Media Technology
Aalborg University, Niels Jernes Vej 14, 9220 Aalborg Øst, Denmark
martin@create.aau.dk

Keywords: GPU, shader, user-centered design, user interface

Abstract: In the context of game engines with graphical user interfaces, shader programs for GPUs (graphics processing units) are an asset for game development that is often used by artists and game developers without knowledge of shader programming. Thus, it is important that non-programmers are enabled to explore and exploit the full potential of shader programs. To this end, we develop principles and guidelines for the design of user-centered graphical interfaces for shaders. With the help of several examples, we show how the requirements of a user-centered interface design influence the choice of widgets as well as the choice of the underlying shader parameters.

1 INTRODUCTION

In recent years, game engines with graphical user interfaces (e.g. Blender (Blender Foundation, 2011) or Unity (Unity Technologies, 2011)) gained popularity and enabled non-programmers to develop full-fledged, three-dimensional games with a minimum of scripting. In this context, vertex and pixel shaders for GPUs (graphics processing units) are just one kind of assets among others (e.g. meshes, animations, texture images, etc.), which are combined and manipulated by game developers and artists who often have very little (if any) knowledge of shader programming. In order to allow non-programmers to take full advantage of these GPU-based shaders, it is therefore important that shader programmers implement appropriate user interfaces for their shaders. This aspect of shader programming has received very little attention until recently although it is presumably crucial for the usability and commercial success of shader programs.

This work focuses on the user-centered design of interfaces for GPU-based vertex and pixel shaders in game engines and is, therefore, partly based on the principles for artist-friendly controls of Renderman shaders proposed by Sadeghi et al. (Sadeghi et al., 2010), which are reviewed in Section 2. However, we also take the actual user interface (including the choice and design of widgets) into account. This results in additional requirements, which are based on principles of user-centered design. We discuss these requirements and principles in Section 3.

Based on these principles, several more specific guidelines for the design of user interfaces of shader programs are developed in Section 4 — specifically for the choice and design of widgets.

In order to illustrate the principles and guidelines and to compare the resulting shader parameters with more traditional parameters, several examples are discussed in Section 5. In Section 6, conclusions and future work are described.

2 RELATED WORK

Surprisingly little work has been published on the design of user interfaces for shader programs. One reason is that the interface between the main application and shader programs often is an internal, technical interface that is only necessary because of the graphics API (application programming interface) to program GPUs, e.g. OpenGL or Direct3D; therefore, both sides of the interface are usually implemented by one and the same programmer. If a user interface is necessary to control a shader program, this interface is usually considered part of the application's user interface; therefore, it is considered an issue of user interface design instead of an issue of graphics programming.

On the other hand, some game engines, e.g. Unity (Unity Technologies, 2011), automatically generate graphical user interfaces for shader parameters from the source code of the shader definition. For example,

Figure 1: User interface for shader parameters in Unity.

the following shader code defines the user interface shown in Figure 1 for three shader parameters in the game engine Unity:

```
Properties {
   _MyFloat ("my number", Float) = 0.5
   _OtherFloat ("my slider", Range(0,1)) = 0.5
   _MyColor ("my color", Color) = (1,1,1,1)
}
```

Thus, parameters of shaders can be manipulated by game developers and technical artists with little or no knowledge of shader programming. In contrast to traditional systems, this approach puts the responsibility for the design of the shaders' interface on the shader programmer, who might lack the knowledge and skills to design an appropriate user interface.

One of the very few publications concerned with the design of user interfaces for shaders is the work by Sadeghi et al. (Sadeghi et al., 2010), who identified three requirements for artist-friendly control parameters:

- intuitive behavior, i.e. "control parameters should correspond to visually distinct features and behave predictably".
- decoupling, i.e. "changes to one visually distinct feature (e.g. brightness of primary highlight, color of the secondary highlight) should not affect other visually distinct features".
- going beyond reality, i.e. "controls which cover the physically correct domain, but which extend seamlessly into the non-physical domain".

However, Sadeghi et al. mention cases where the coupling of parameters is desirable to define one parameter relative to others. Furthermore, they describe one case where the coupling of parameters is useful but should optionally be broken later in the artistic process to allow for the independent manipulation of all parameters.

It is notable that these principles are only concerned with the choice of numeric parameters while the design of the actual user interface (i.e. the widgets to specify these numeric parameters or their coupling) are of no concern. This might be due to the fact that Sadeghi et al. are only concerned with Renderman shaders.

Sadeghi et al. also discuss a procedure to define artist-friendly controls for the specification of pseudo scattering functions; however, several of these steps are specific to the particular shaders. The generally applicable steps are: examination of the physical model; decomposition into visually distinct features (e.g. highlights, glints, and rim light); and definition of artist-friendly controls (e.g. color, intensity, size, shape, and position) for each distinct feature.

3 PRINCIPLES OF USER-CENTERED DESIGN OF SHADER INTERFACES

In this section, we apply results of user-centered design as presented by Norman (Norman, 2002, Chapter 7), who proposed seven principles for transforming difficult tasks into simple ones (Norman, 2002, pages 188-189): 1) use both knowledge in the world and knowledge in the head; 2) simplify the structure of tasks; 3) make things visible: bridge the gulfs of execution and evaluation; 4) get the mappings right; 5) exploit the power of constraints, both natural and artificial; 6) design for error; 7) when all else fails, standardize.

Based on these principles, we extended the three principles proposed by Sadeghi et al. (Sadeghi et al., 2010) and include three additional principles. Thus, our six principles for the user-centered design of interfaces to control parameters of shaders are:

1. **Intuitive behavior:** parameters should control visually distinct features, which can be identified by the user by means of comprehensible labels; changes of quantitative parameters should correspond to appropriately scaled changes in the appearance of features; the behavior of widgets to manipulate parameters should be known to users.
2. **Decoupling:** each parameter should control one and only one visually distinct feature; (optional) exceptions should be communicated to the user.
3. **Artistic freedom:** parameters should neither be constrained by the limits of physical models nor should their availability be constrained, i.e. any parameter may be manipulated at any time.
4. **Visible parameter values:** the current parameter settings, and — for quantitive parameters — default values, extreme values, and all possible values should be represented graphically.
5. **Interactive feedback:** the effect of user-specified parameters should be displayed interactively while the parameters are manipulated in order to encourage interactive exploration; users should be able to apply shaders to any appropriate model or system-provided default models; furthermore,

feedback should be given in the case of "implicit" couplings where one parameter becomes ineffective for specific settings of other parameters.

6. **Design for error:** changes of parameters should be reversible at any time by manipulation, undoing the most recent manipulation, or reverting to a saved set of parameters.

In the following, we discuss how Norman's general principles of user-centered design manifest themselves in the proposed principles.

External and internal knowledge: External knowledge is employed by the labeling of parameters in Principle 1, the communication of coupled parameters in Principle 2, the visual representation of default and extreme values in Principle 4, and the feedback in Principle 5. Principle 1 also employs internal knowledge since the labels rely on knowledge of the denoted features. Furthermore, internal knowledge is also required by Principle 1 to use the widgets that control the shader parameters.

Simplified tasks: Practically all principles are designed to simplify various tasks associated with the manipulation of shader parameters; however, Principles 1 and 2 are particularly important simplifications since the systematic manipulation of unintuitive, coupled parameters can be an extremely difficult and cumbersome process that mainly consists of trial and error. A further fundamental simplification can be achieved by offering a library of ready-made parameter presets. The possibility of such presets is required by Principle 6.

Visibility for execution: Users often require support to determine which actions are required to achieve certain intentions. The visual feedback of the applied shader in Principle 5 helps users to determine which visual feature they want to change. The correspondence of visual distinct features to a single parameter (Principles 1 and 2) helps them to identify the parameter that should be manipulated to change the feature. Principle 3 makes sure that this parameter can be manipulated at any time and Principle 4 supports users in determining what kind of change is likely to be appropriate by communicating the current value, the default value and extreme values. Note that the appropriate rate of change of appearance required by Principle 1 also helps to estimate the required change. In case the change turns out to be ineffective, the feedback required by Principle 5 should support the user to identify the parameter that has to be changed to achieve the intended effect.

Visibility for evaluation: Apart from Principles 1 and 2, the main means to support visibility for evaluation is interactive visual feedback by applying the parameterized shader to a user-specified model in Principle 5; however, the graphical representation of current values of parameters in Principle 4 and the feedback about ineffective parameters is also crucial to enable users to evaluate their actions.

Good mappings: Norman summarizes different kinds of mappings (Norman, 2002, page 199). In our case, Principles 1 and 3 ensure a good mapping between intentions to change specific visual features and possible actions. Furthermore, Principles 1, 2, and 4 ensure a good mapping between actions and their effects. Moreover, the graphical representation of the current settings in Principle 4 ensures a good mapping between the actual system state and what is displayed. Finally, the visual feedback required by Principle 5, ensures that the system state can be compared to the intentions and expectations of the user.

Constraints: Principles 1 and 2 constrain the user to changes of parameters that are limited to certain visually distinct features. Furthermore, Principle 4 will not only constrain users to the appropriate range of parameter values but usually also to an appropriate minimum change since each allowed parameter value is often represented by at least one pixel.

Design for Error: Principle 6 was included particularly to support design for error.

Standards: Principle 1 — and in practice also Principle 4 — encourages the use of standard widgets. Moreover, Principle 1 results in a set of standard features such as identified by Sadeghi et al.: color (including opacity), intensity, shape, position, etc.

4 GUIDELINES FOR USER INTERFACES OF SHADERS

The six principles that were proposed in the previous section are illustrated by several examples in Section 5. It is, however, useful to develop some more specific guidelines based on these principles — in particular with respect to the choice and design of the widgets to manipulate shader parameters, ways to communicate couplings of parameters, and the computation of parameters of GPU-based shaders from user-specified parameters.

4.1 Color Selection

One or more colors are usually among the visual distinct features of a surface shader. Appropriate colors should potentially (i.e. under very specific circumstances) appear in a rendered image exactly as specified by the user. This is implied by Principle 1, which requires parameters that control visual features: a color that cannot appear in a rendered image would not be a visual feature. Moreover, this requirement tends to make it possible to pick specific colors in photos or artwork.

Principle 1 encourages the use of standard widgets in order to guarantee an intuitive behavior of widgets. Thus, the standard color selector is usually the best way to specify colors. This includes opacities of features if their specification is supported by the standard color selector; otherwise a slider should be used as discussed below.

If a default value of a color is available, it should not only be used as initial value for the color selector but it should always be represented graphically in the widget as suggested by Principle 4. Alternatively, a button might be added that allows the user to select the default color.

4.2 Sliders

Sliders are often the best widget to interactively specify scalar numbers such as intensities, opacities, sizes, translations in a certain direction, etc. The behavior of a standard slider widget is very likely to be familiar to users in accordance with Principle 1.

Moreover, most sliders visualize the current setting as well as the extreme values and, in fact, all possible values as required by Principle 4. If the default value is not one of the extreme values, it is often possible to define the parameter such that the center position of the slider represents the default value, which is conveyed to users even without any explicit labels if the default value is also the initial value of the widget.

In Western culture, Principle 1 suggests that a slider movement to the right or up should correspond to an increase of the parameter. Furthermore, changes should correspond to appropriately scaled changes in appearance. Specifically, the perceived "sensitivity" of the slider (i.e. the change of appearance per distance moved by the slider) should be appropriate over the whole range of possible values, i.e. the sensitivity should not be too high, which would make it difficult — or even impossible — to choose a sufficiently good approximation to a particular value, nor should it be too low, which would require large movements of the slider for visible changes of the appearance of the shader and therefore might appear to the user as if the slider was ineffective. In order to satisfy this requirement, it is often necessary to introduce a non-linear mapping from user-specified parameters to low-level GPU-shader parameters. For example, the range from 0 to 1 of a user-defined parameter can be mapped to the range from 0 to infinity of a low-level parameter by the mapping $x \mapsto ((1-x)^{-c_1} - 1)c_2$ with two positive real numbers c_1 and c_2, which can be adjusted by the programmer to provide an appropriate sensitivity.

4.3 Couplings of Parameters

While Principle 2 encourages decoupled parameters, there are various cases in which coupled parameters are actually more user-friendly. One reason for the coupling of parameters is to offer users alternative ways of specifying the same (low-level) parameters. An example is the specification of a color either in the RGB or the HSV space. In these cases, the coupling should be clearly communicated to the user, for example by displaying only one set of independent parameters (e.g. either the RGB coordinates or the HSV coordinates) at a time.

As discussed by Sadeghi et al. (Sadeghi et al., 2010), it is sometimes useful to optionally couple parameters; for example, in order to allow users to find an initial solution by manipulating only a subset of all parameters and automatically computing further, coupled parameters. Novice users who are not familiar with all parameters would particularly benefit from such a coupling. However, later in the artistic process, some users might want to fine tune these additional parameters separately without the coupling between parameters. Some expert users might even prefer to always use the decoupled version.

Thus, a shader interface should allow the user to enable or disable the coupling of shader parameters. This can be achieved with the help of a checkbox or similar widgets. In fact, the concept of optionally coupled parameters is familiar to many users; for example, several image editing tools allow to fix the aspect ratio of an image when specifying a new width or height.

In some cases, couplings are unavoidable; for example, highlights become invisible if their color is set to black; thus, other parameters, e.g. the size of the highlight, become ineffective as the highlight no longer exists. This is problematic if users don't understand the reason for the apparent ineffectiveness of a parameter. A possible solution is to add a text label next to the control of an ineffective parameter that refers to the parameter setting that caused the parameter to become ineffective.

4.4 Preprocessing of Parameters

There are several situations that require preprocessing of parameters outside of GPU-based shaders; for example, the computation of low-level shader parameters from artist-friendly parameters; see Section 5. Similarly, some shaders require the computation of textures (e.g. for lookup tables) based on user-specified parameters. Therefore, it is often useful to include an additional preprocessing step to map user-specified parameters to the low-level shader parameters of a GPU-based shader. In some game engines, this can be accomplished with the help of a script that processes user-specified parameters and sets shader parameters accordingly.

5 EXAMPLES

This section discusses the application of the proposed principles and guidelines to actual examples.

5.1 Phong Reflection Model

Most shader implementations of the Phong reflection model (Phong, 1975) employ color parameters for the reflectance coefficients of diffuse and specular reflection. These colors correspond to actual visible colors under particular lighting conditions. Moreover, they are decoupled and they are not restricted by a physical model. In other words, they fulfill Principles 1, 2, and 3, which probably has contributed to the tremendous success of the Phong reflection model in computer graphics.

Furthermore, Phong introduced an exponent n between 0 and infinity, which models the dependency of the specular reflection on the angle s between the direction of the reflected light and direction from the surface point to the camera by the term $(\cos(s))^n$. The parameter n is sometimes described as "shininess" since larger values correspond to smaller highlights, i.e. shinier surfaces. Unfortunately, it is difficult to control this parameter by a slider since the sensitivity of the changes in appearance vary dramatically: values between 0 and about 20 correspond to significantly different sizes of highlights while larger values have less and less influence.

In order to determine the low-level parameter n by a slider parameter x for the size of highlights, we could employ a non-linear mapping, e.g. $n = (x^{-2} - 1) \times 20$. This mapping allows users to change n between infinity and 0 with reasonable sensitivity by manipulating the slider parameter x between 0 and 1. The center value $x = 0.5$ corresponds to a reasonable default value $n = 60$. Using an interactive slider with this mapping to control the size of highlights would satisfy Principles 1 to 5 and therefore is very likely to provide a significantly improved experience for users of the shader.

It should also be noted that this mapping is a good example for the benefits of a preprocessing step since n has to be computed only once for each material instead of once per vertex or fragment.

5.2 Schlick's Fresnel Factor

For many materials, the specular reflectance coefficients depend on the angle between incident and reflected light. For wavelength λ, the Fresnel factor $F_\lambda(u)$ models this dependency on the cosine u of the angle between the direction to the camera and the halfway vector (i.e. the normalized sum of the normalized directions to the camera and the light source). Schlick (Schlick, 1994) proposed the approximation

$$F_\lambda(u) = f_\lambda + (1 - f_\lambda)(1 - u)^5 \qquad (1)$$

for this factor. In actual shader implementations, three values f_λ for red, green, and blue wavelengths are usually combined in one color parameter. Since u is not a parameter, Schlick's Fresnel factor has only one color parameter. It is noteworthy that this color parameter is in fact visible in images if the camera, the light source and the surface normal vector are aligned; thus, Principle 1 is satisfied. The factor is also decoupled from other parameters and therefore satisfies Principle 2. However, Schlick's approximation does not provide any artistic freedom beyond the specification of f_λ.

In order to provide artists with more control over the Fresnel factor as suggested by Principle 3, the power 5 in Equation 1 could be replaced by another decoupled, user-specified shader parameter p. Since the effect becomes very subtle for larger powers than 5, a suitable range for the parameter p is from 0 to 10 such that the value at the center of the slider corresponds to the value 5 in Schlick's approximation. In this way, the default value and reasonable extreme values are communicated to the user in accordance with Principle 4. Furthermore, since the effect becomes less visible for larger powers p, a mapping $p = 10 \times (1 - x)$ for a slider parameter x between 0 and 1 is useful such that a larger slider parameter corresponds to a stronger visual effect in accordance with Principle 1.

5.3 Ward's Anisotropic Reflection

Ward's anisotropic reflection model (Ward, 1992) introduces two parameters α_x and α_y (see Equation 5a

in (Ward, 1992)), which are the standard deviations of the surface slope in an x and y direction, which are orthogonal to the surface normal direction. The parameters are sometimes described as the "roughness" of the surface in the two directions and therefore determine the extent of specular highlights in those directions. As noted by Ward, the two parameters are uncorrelated, i.e. decoupled. Moreover, the extent of highlights in two directions can be considered visually distinct features.

However, both parameters are coupled with a third parameter, namely the specular reflectance coefficient, which is divided by α_x and α_y in the cited equation. Together, these three parameters determine the maximum intensity of specular highlights in a way to guarantee energy conservation. This coupling is particularly problematic because it makes it extremely difficult to change the size of highlights without changing their maximum intensity since this task would require users to manipulate these three parameters at the same time in a nonlinear way. On the other hand, the task of changing the size of a highlight without changing its color is easily accomplished in the Phong reflection model.

Apart from avoiding the most important couplings, it is also necessary to identify those parameters which are most easily understood and controlled by users. As mentioned in Section 4, there might be more than one set of decoupled parameters, e.g. either HSV or RGB coordinates. However, in the case of Ward's anisotropic reflection model, the most artist-friendly set of parameters probably consists of the maximum color of the specular highlight (corresponding to the specular reflectance coefficient but without the division by α_x and α_y in Equation 5a of (Ward, 1992)) and two parameters to control the size and shape of highlights. Specifically, the two parameters could correspond to the sum $\alpha_x + \alpha_y$ and the difference $\alpha_x - \alpha_y$. The latter is a good measure of the anisotropy of the reflection as a difference of 0 corresponds to an isotropic highlight (which is therefore a good default value). If the difference is kept constant, the sum $\alpha_x + \alpha_y$ controls only the size of highlights (in both directions).

Note that it would be possible to compute the original parameters of Ward's anisotropic reflection model from the proposed artist-friendly parameters in a preprocessing step and then implement Ward's model in a shader. However, this would not result in the most efficient solution since it would include the division by α_x and α_y in the shader.

6 CONCLUSIONS

The examples in the previous section illustrate how to construct artist-friendly shader parameters from the six principles and the guidelines proposed in Sections 3 and 4. Not only can these principles and guidelines serve as a checklist for specific sets of parameters, but they can also guide shader programmers to find more artist-friendly parameters in order to provide a better user experience.

The principles and guidelines were derived by applying results from research on user-centered design to GPU-based shader programming. One of the important results is how user-centered design influences the choice and mappings of shader parameters. Furthermore, the examples show that an additional preprocessing step is useful in many cases in order to map user-specified parameters to low-level shader parameters. For best performance, however, changes to the shader programs are inevitable even if such a preprocessing step is employed.

Future work has to experimentally validate the proposed principles and guidelines by applying them to more shaders and conducting appropriate user tests.

REFERENCES

Blender Foundation (2011). Blender 3d content creation suite. Web site: http://www.blender.org.

Norman, D. A. (2002). *The Design of Everyday Things*. Basic Books, New York, reprint paperback edition.

Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18:311–317.

Sadeghi, I., Pritchett, H., Jensen, H. W., and Tamstorf, R. (2010). An artist friendly hair shading system. *ACM Trans. Graph.*, 29:56:1–56:10.

Schlick, C. (1994). An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246.

Unity Technologies (2011). Unity game engine. Web site: http://www.unity3d.com.

Ward, G. J. (1992). Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph.*, 26:265–272.