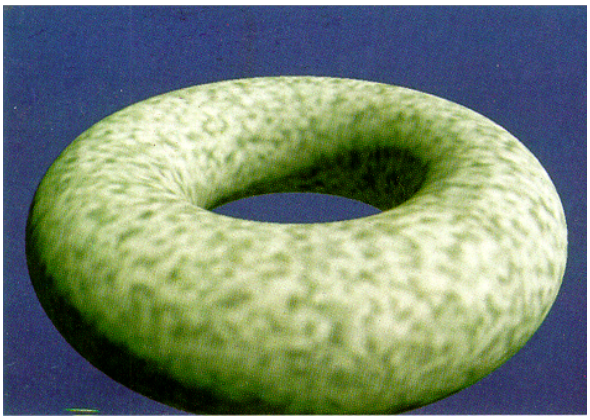
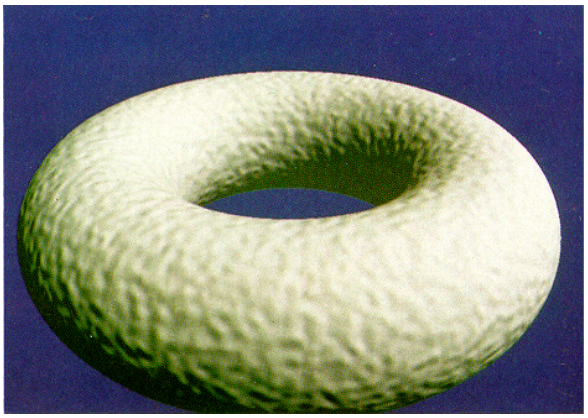
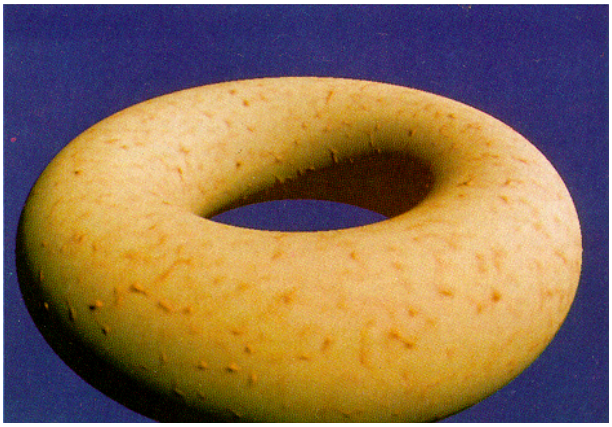
Abstract

We introduce the concept of a Pixel Stream Editor. This forms the basis for an interactive synthesizer for designing highly realistic Computer Generated Imagery. The designer works in an interactive Very High Level programming environment which provides a very fast concept/implement/view iteration cycle.
Naturalistic visual complexity is built up by composition of non- linear functions, as opposed to the more conventional texture mapping or growth model algorithms. Powerful primitives are included for creating controlled stochastic effects. We introduce the concept of "solid texture" to the field of CGI.
We have used this system to create very convincing representations of clouds, fire, water, stars, marble, wood, rock, soap films and crystal. The algorithms created with this paradigm are generally extremely fast, highly realistic, and asynchronously parallelizable at the pixel level.
CR CATEGORIES AND SUBJECT DESCRIPTORS: 1.3.5 [Computer Graphics]: Three-Dimensional Graphics and Realism
ADDITIONAL KEYWORDS AND PHRASES: pixel stream editor, interactive, algorithm development, functional composition, space function, stochastic modeling, solid texture, fire, waves, turbulence
Introduction
This work arose out of some experiments into developing efficient naturalistic looking textures. Several years ago we developed a simple way of creating well behaved stochastic functions. We found that combinations of such functions yielded a remarkably rich set of visual textures. We soon found it cumbersome to continually rewrite, re.compile, and rerun programs in order to try out different function combinations.
This motivated the development of a Pixel Stream Editing language (PSE). Cook [1] has proposed an expression parser for this purpose. We have taken the same idea somewhat farther by providing an entire high level programming language available at the pixel level. Unlike [1], The PSE contains, general flow of control structures, allowing arbitrarily asynchronous operations at different pixels.
With the PSE we may interactively compose functions defined over modeling space. By starting with the right choice of primitive functions we can build up some rather convincing naturalistic detail with surprisingly simple and efficient algorithms.
We will first describe the PSE language and environment. Then we will introduce the concept of solid texture, together with our well behaved stochastic functions. Finally we will give some examples of how these concepts work together in actual practice.
A PIxelStream Editing Language
Consider any list of variable names. We will call any list of corresponding values for these variables a “pixel”. For example, one possible pixel for the variable list [red green blue] is [0.5 0.3 0.7]. We will call any list of names together with a two dimensional array of pixels an "image".
A Pixel Stream Editor (PSE) is simply a filter which converts input images to output images by running the same program at every pixel. We always read and write image pixels in some canonical order. At any one pixel, all that the program "knows" about each image are its variable names and their current values.
We interpret color as a [red green blue] vector (?)

Spotted Donut
Bumpy Donut

Stucco Donut
Disgusting Donut


288
Bozo's Donut
Wrinkled Donut

Space Functions and Solid Texture

A number of researchers have proposed procedural texture, notably [3], [5], and [6]. As far as we know all prior work in this direction has been with functions which vary over a two dimensional domain.
Suppose we extend this to functions which vary over a three dimensional domain. We call any function whose domain is the entirety of (x,y,z) space a "space function".
Any space function may be thought of as representing a solid material. If we evaluate this function at the visible surface points of an object then we will obtain the surface texture

that would have occurred had we "sculpted" the ob~.~'t out of the material. We will call a texture so formed a “solid texture”.
This approach has several advantages over texture mapping :

1. Shape and texture become independent. The texture does not need to be "fit" onto the surface. If we change the shape or carve a piece out of it, the appearance of the solid material will accurately change.
2. As with all procedural textures, the database is extremely small.

Although it is not immediately obvious, this paradigm is a superset of conventional texture mapping techniques. Any stored texture algorithm may be cast as a table lookup function composed with a projection function from three dimensions to two.
We will use solid texture repeatedly over the course of this paper to simulate a variety of materials.

Noise()

In order to get the most out of the PSE and the solid texture approach we have provided some primitive stochastic functions with which to bootstrap visual complexity. We now introduce the most fundamental of these.
Noise() is a scalar valued function which takes a three dimensional vector as its argument. It has the following properties :

Statistical invariance under rotation
(no matter how we rotate its domain, it has the same statistical character)
A narrow bandpass limit in frequency
(its has no visible features larger or smaller than within a certain narrow size range)
Statistical invariance under translation
(no matter how we translate its domain, it has the same statistical character)

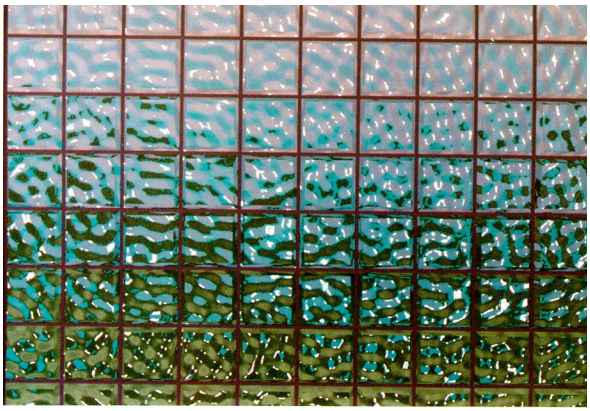
Noise() is a good texture modeling primitive since we may use it in a straightforward manner to create surfaces with desired stochastic characteristics at different visual scales, without losing control over the effects of rotation, scaling, and translation. This works well with the human vision system, which tends to analyze incoming images in terms of levels of differently sized detail [4].
The author has developed a number of surprisingly different implementations of the Noise() function. Some real tradeoffs are involved between time, storage space, algorithmic complexity, and adherence to the three defining statistical constraints. Because of space limitations, we will describe only the simplest such technique. Although generally adequate, this procedure only approximately conforms to the bandwidth and rotational invariance constraints.

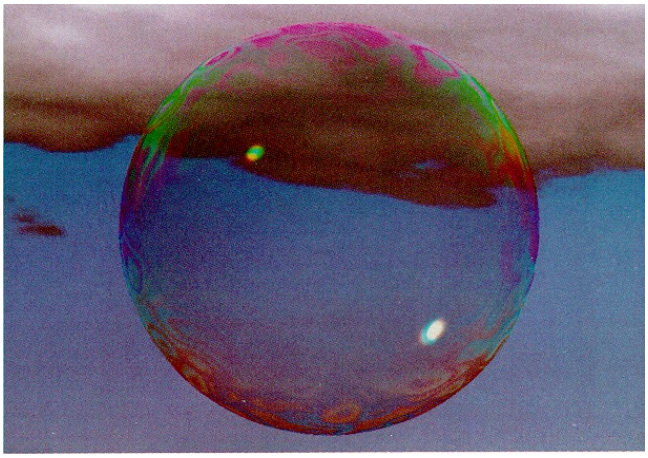
1. Consider the set of all points in space whose x, y, and z coordinates are all integer valued. We call this set the integer lattice. Associate with each point in the integer lattice a pseudo- random value and x, y, and z gradient values. More precisely, map each ordered sequence of three integers into an uncorrelated ordered sequence of four real numbers: [a,b,c,d] = H([x,y,z]), where [a,b,c,d] define a linear equation with gradient [a,b,c] and value d at [x,y,z]. H() is best implemented as a hash function.
2. If [x,y,z] is on the integer lattice, we define Noise([x,y,z]) = d[x,y,z]. If [x,y,z] is not on the integer lattice we compute a smooth (eg. cubic polynomial) interpolation between lattice equation coefficients, applied first in x (along lattice edges), then in y (within lattice z-faces), then in z. We then evaluate this interpolated linear equation at [x,y,z].

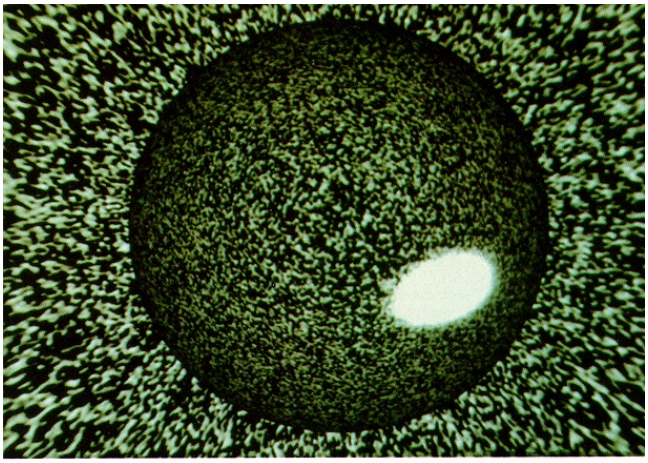
We will now show some of the simpler uses of Noise(). We will assume that "'point" and "normal" are vector valued input image variables.
By evaluating Noise() at visible surface points of simulated objects we may create a simple "random" surface texture (figure Spotted.Donut) :
color = white * Noise(point)

The above texture has a band-limited character to it; there is no detail outside of a certain range of size. This is equivalent to saying that the texture's frequency spectrum falls off away from some central peak frequency.
Through functional composition we may do many different things with the value returned by the Noise() function. For example, we might wish to map different ranges of values into different colors (figure Bozo's.Donut) :
color = Colorful(Noise(k * point))
In the above example we have scaled the texture by multiplying the domain of Noise() by a constant k. An nice feature of the functional composition approach is the ease with which such modifications may be made.
Another convenient primitive is the vector valued differential of the Noise() signal, defined by the instantaneous rate of change of Noise() along the x, y, and z directions, respectively. We will call this function Dnoise().

Art Glass







291

Dnoise() provides a simple way of specifying normal perturbation [7] (figure Bumpy.Donut) :

normal + = Dnoise(point)

By using functions of Noise() to control the amount of DnolseO sPerturbation, we may simulate various types of surface (figure tucco.Donut), and use these in turn to design other types of

surface (figure Disgusting.Donut).

AS another example, a 1/f signal over space can be simulated by looping over octaves (powers of 2 in frequency) :

∑Noise(point* (2^i)) / (2^i)

In order to create 1/f texture we observe that the differential of a function with a 1/f frequency spectrum is a vector valued function with a fiat frequency spectrum (ie.

gradients of 1/f functions are similar at all scales). This means that we must create similar normal perturbation in all octaves (figure Wrinkled.Donut) :

```
f=l
while f < pixel_freq
        normal + = Dnoise(f * point)
        f*=2
```

Note that the calculation stops at the pixel level. In this way unwanted higher frequencies are automatically clamped.
Unlike subdivision based [5] or Fourier space [14] fractal simulations, the above algorithm proceeds independently at all sample points. There is no need to create and modify special data structures in order to provide spacial coherence, This results m a considerable time savings. As with 'all of the algorithms we will present, the calculation at different pixels can be done in any order, in parallel, or even on different machines.

Marble - An Example of a Solid Texture

We can use Noise() to create function turbulence() which gives a reasonable visual appearance of turbulent flow (see Appendix). We may then use turbulence() to simulate the appearance of marble.
We observe that marble consists of heterogeneous layers. The "marble" look derives from turbulent forces which create deformations before these layers solidify.
The unperturbed layers alone can be modeled by a simple coler filtered sine wave :

```
        function boring_marble(point)
                x = point[l]
                return marble_color(sin(x))
```

where point[1]denotes the first (ie. x) component of the point vector and marble_color() has been defined as a spline function mapping scalars to color vectors. To go from this to realistic marble we need only perturb the layers :

```
function marble(point)
        x = point[1] + turbulence(point)
        return marble_color(sin(x))
```

By invoking this procedure at visible surface]points we can create quite realistic simulations of marble ob~cts (figure Marble,Vase).

Fire

We can create fire using turbulence() whenever we have a well defined flow.
For example, suppose we wish to simulate a solar corona. We will assume that the following entities :
norm() - scalar length (ie. norm) of a vector
direction() - the (unit length) direction of a vector
frame - global lime variable (ie. one frame dick)

have already been defined.
A corona is hottest near the emitting sphere and cools down with radial distance from the sphere center. At any value of radius, and hence of temperature, a particular spectral emission is visible. Assume we have defined a function color..of_emission0 which models emission color as a function of radius.
Modeled as a smooth flow, the corona would be implemented by :

```
        smooth_corona(point - center)
```

```
function smooth_corona(v)
        radius = norm(v)
        return color_of_emission(radius)
```

By adding turbulence to the radial flow we can turn this into a realistic simulation of a corona (figure Corona) :

```
function corona(v)
        radius = norm(v)
        dr = turbulence(v)
        return color_of_corona(radius + dr)
```

To animate this we linearly couple the domain of turbulence to time :

```
function moving_corona(v)
        radius = norm(v)
        dr = turbulence(v - frame * direction(v))
        return color_of_corona(radius + dr)
```

Water

Suppose we wish to create the appearance of waves on a surface. To simplify things we will use normal perturbation [7] instead of actually modifying the surface position.

Max [8] approached this problem by using a collection of superimposed linear wave fronts. Linear fronts have a notable deficiency - they form a self.replicating pattern when viewed over any reasonably large area.
To avoid this we use spherical wave fronts eminating from point source centers [17]. More precisely, suppose at a given pixel a particular surface point is visible. For any wave source center, we will perturb the surface normal towards the center by a cycloidal function of the center's distance from the surface point :

```
normal + = wave(point - center)
function wave(v)
        return direction(v) * cycloid(norm(v))
```

We can create multiple centers, let's say distributed randomly around the unit sphere, by using the direction of Dnoise() over any co]leetion of widely spaced points. This works because (by definition) the value of Dnoise() is uncorrelated for any two points which are spaced widely enough apart :

```
function makewaves(n)
        for i in [1 .. n]
                center[i] = direction(Dnoise(i * [100 0 0] ))
        return center
```

To make a wave model with 20 sources we would enter :

```
if begin_frame
        center = makewaves(20)
for c in center
        normal + = wave(point- c)
```

Note that the surface need not be planar. By making our wave signal defined over 3-space we have ensured shape independence. This means that we can run the above procedure on any shape. The illustration "Water Crystal" was made using 20 sources (figure Water.Crystal). A similar procedure was used to simulate an "Art Glass" partition (figure Art.Glass).
Waves of greater realism are created by distributing the wave- front spacing frequencies using a 1/f relatiouship of amplitude to frequency. If we assign a random frequency f to each center, the last line of tiw procedure then becomes :

```
normal + = wave((point - c) * f) / f
```

Using this refinement (again with 20 sources) we can realistically simulate ocean surfaces (figure Ocean.Sunset).
Since each wave front moves outward linearly with time we may animate these images by adding a linear function of time to the argument passed to cycloid():

```
function moving.wave(v, Dphase)
        return direction(v) * cycloid(norm(v) - frame * Dphase)
```

where Dphase is the rate of phase change. For greatest realism we make Dphase proportional to f^0.5 [9]. The wave images pictured are actually stills from such animations.

Other Examples - Clouds and Bubbles

The two bubble images were designed by Carl Ludwig using the PSE. The various elements were all created and assembled by functional composition in the PSE.
For example, in the topmost bubble image the background clouds were created by composing a color spline function with turbulence(). The reflection and refraction from the bubble surface were done by using simple vector valued functions to modify an incoming direction vector in accordance with the appropriate physical laws. These were composed with the cloud function and added together.
In the center image, a function corresponding to the shape of an illuminated window was composed with reflection and refraction functions.
The appearance of variable bubble thickness was simulated by multiplying turbulence() by each of a red, 8teen, and blue frequency and using sin() of this to create constructive and destructive interference fringes. In the PSE this looks like :

```
color *= 1 + sin([rfreq gfreq bfreq] * turbulenca(point))
```

Compositing

We can use the PSE simply as a digital image compositor, in which case it functions as a generalization of [10]. We can also use it to combine and modify images in more unusual ways.
Suppose for example that we wish to synthesize some flame on the PSE, knowing that later we will race;re some other animation to be composited with our synthetic flame.
We may defer the aesthetic decision of how to color the flame until after looking at this footage. We do this by computing the flame in two passes. The first pass outputs only a scalar flame
value. The second and simpler pass maps this scalar quantity to the appropriate color vector.

Note that this process involves no recalculation of the flame itelf. The second pass through the PSE is being used only as a general color splining filter, at a small fraction of the total computing COSt.
In an actual commercial production .this ability to split computation costs and defer post-production oeasions adds enormously to throughput.
In more unusual cases we may use the scalar flame to modulate the frequency distribution or height of water waves, or the amount of rocklike character to give to a surface. In this context our approach is similar to that of [1] and [10], the difference being the extra flexibility we gain by the ability to specify arbitrary asynchronous pixel operations.

Consideration of Efficiency

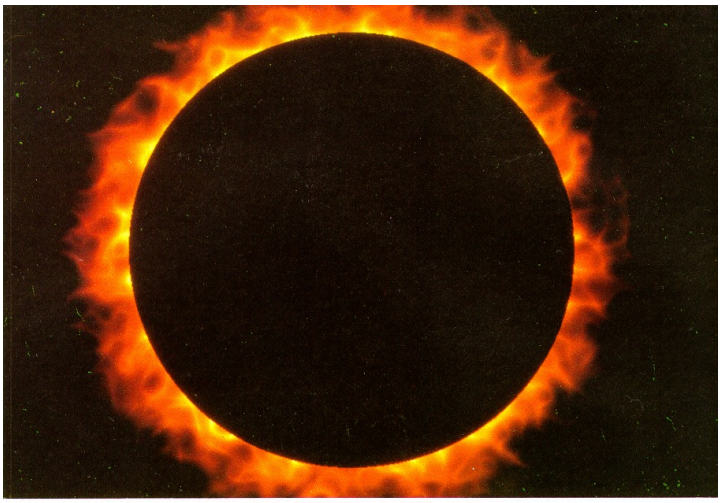
The efficiency of an implementation is a rather elusive thing. This is because it consists of three fairly different considerations.
Most familiar is time efficiency. There is also space efficiency, which often is inversely proportional to time efficiency (as m "should we use a procedure or a lookup table?").
The third consideration, often overlooked, is flexibility. Many of us are familiar with archaic and monolithic "dinosaur" programs that nobody dare modify lest they fall apart altogether. Such programs must be used "as is" or else scrapped and rewritten from scratch.
The approach we offer here does not always produce the most efficient algorithms. What it does offer is the opportunity to try out new approaches quickly and painlessly. For CGI in particular this is of the utmost importance. We generally want to see what the picture looks like before proceeding with optimization. Once implemented, PSE algorithms lend themselves readily to optimization by virtue of their simplicity and high degree of modularity.
In addition, a number of effects are ideally suited to a functional composition paradigm; generally when there is interplay between a simple regular structure and a complex stochastic structure. This is because we can use nonlinear functional composition to model the stochastic part of the structure. This will result in both good time efficiency and good space efficiency.
The flame model constitutes such a "best case" for our approach. The final motion picture quality animation ran in about 10 minutes a frame, written entirely in an unoptimized interpreted pseudo-code implementation of the design language on a Gould SEL 3287 Minicomputer. This appears to be much faster than the particle system approach of Reeves [11]. With optimization and true compilation a speedup of a factor of 5 is indicated. The marble vase, with twice as large an area of visible turbulence, took about 20 minutes to compute.
In all cases, the low resolution interactive design loop took between 15 seconds and 1 minute per iteration.

Now What?

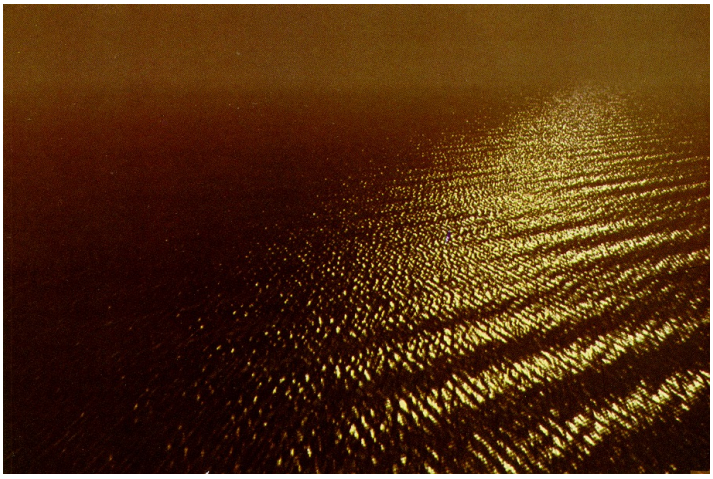
We plan to make a number of improvements m the system. We are developing an optimized compiler for the design language which recognizes quantities that vary slowly over the linage stream and computes quantities dependent these only as necessary. We are also adding a general facility for direct insertion of large data bases rote the image prior to pixel streaming.
We are currently using the same paradigm of composition with stochastic functions for motion and shape modeling.
We have applied our approach to modeling stochastic motion not only for continuous turbulence models, but also for such things as falling leaves, swaying trees, flocks of birds, and muscular

293

294



Corona
:\
Occan Sunset



295

rippling. In general the paradigm is appropriate whenever a regular, well defined macroscopic motion contains some stochastic component.
To create interesting stochastic shapes, we have generalized on the work of Bliun [151. Given any space filling scalar valued function, we may consider the shape formed by any isosurface (surface of constant value) of the function. It turns out that a very rich class of shapes may be created in this manner (for example, we can actually build the three dimensional structure of the flame shown in figure Corona). We understand that Lance Williams of NYIT [16] is pursuing a similar line of research.

Conclusions

We have shown a new approach to the design of realistic CGI algorithms. We have introduced the concepts of the Pixel SUream Editor and of solid texture. We have demonstrated a number of effects which would have been considerably more difficult and expensive, and in some cases impossible, to generate by previously known techniques.

Appendix- Turbulence

A suitable procedure for the simulation of turbulence using the Noise() signal is :

```
function turbulence(p)
        t=0
        scale = 1
        while (scale > pixelsize)
                t + = abs(Noise(p / scale) * scale)
                scale/= 2
        return t
```

This is actually a simplified approximation to the magnitude of the deformation which results from swirling around the isosurfaces of the Noise() domain along the instantaneous vector field :
e^(-Noise(point)^2)*(normal X Dnoise(point))
This formulation is part of a synthetic turbulence model developed by the author [12]. We use the simplified turbulence() procedure because it is fast and the pictures it produces look good enough.
Even so it is interesting to examine, with only minimal comment, the algorithmic structure of turbulence(). Note the expression

Noise(p / scale) * scale

inside the loop. This says that at each scale the amount of Noise() added is proportional to its size. Thus we obtain a self-similar, or 1/f, pattern of perturbation. This will give a visual impression of brownian motion. Also, while the deformation is continuous everywhere, the abs() at each iteration assures that its gradient will have discontinuous boundaries at all scales. This will give a visual impression of discontinuous flow, which will be interpreted by the viewer as turbulent.