Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation

J. Marks* MERL
B. Andalman Harvard Univ.
P.A. Beardsley MERL
W. Freeman MERL
S. Gibson MERL
J. Hodgins Georgia Tech.
T. Kang CMU
B. Mirtich MERL
H. Pfister MERL
W. Ruml Harvard Univ.
K. Ryall Harvard Univ.
J. Seims Univ. of Washington
S. Shieber Harvard Univ.

Abstract

Image rendering maps scene parameters to output pixel values; animation maps motion-control parameters to trajectory values. Because these mapping functions are usually multidimensional, nonlinear, and discontinuous, finding input parameters that yield desirable output values is often a painful process of manual tweaking. Interactive evolution and inverse design are two general methodologies for computer-assisted parameter setting in which the computer plays a prominent role. In this paper we present another such methodology: *Design Gallery*™ (DG) interfaces present the user with the broadest selection — automatically generated and organized — of perceptually different graphics or animations that can be produced by varying a given input-parameter vector. The principal technical challenges posed by the DG approach are *dispersion*, finding a set of input-parameter vectors that optimally disperses the resulting output-value vectors, and *arrangement*, organizing the resulting graphics for easy and intuitive browsing by the user. We describe the use of DGs for several parameter-setting problems: light selection and placement for image rendering, both standard and image-based; opacity and color transfer-function specification for volume rendering; and motion control for particle-system and articulated-figure animation.

CR Categories: I.2.6 [Artificial Intelligence]: Problem Solving, Control Methods and Search—heuristic methods; I.3.6 [Computer Graphics]: Methodology and Techniques—interaction techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Keywords: Animation, computer-aided design, image rendering, lighting, motion synthesis, particle systems, physical modeling, visualization, volume rendering.

*Address: MERL – A Mitsubishi Electric Research Laboratory, 201 Broadway, Cambridge, MA 02139, U.S.A. E-mail: marks@merl.com.

1 Introduction

Parameter tweaking is one of the banes of computer graphics. Finding input parameters that yield a desirable output is difficult and tedious for many rendering, modeling, and motion-control processes. The notion of getting the computer to assist actively in setting parameters is therefore appealing. One such computer-assisted methodology is interactive evolution [11, 20, 22]: the computer explores the search space of possible parameter settings, and the user acts as an objective-function oracle, interactively selecting computer-suggested alternatives for further exploration. A more automatic methodology is inverse design, e.g., [10, 12, 13, 18, 21, 24, 26]: the computer searches for parameter settings that optimize a user-supplied, mathematically stated objective function.

Unfortunately, there are many interesting and important graphics processes for which interactive evolution and inverse design are not very useful. These processes share two common characteristics:

- High computational cost: if the process cannot be computed in near real time, interactive evolution becomes unusable.
- Unquantifiable output qualities: even though desirable graphics may be readily identified by inspection, it may not be possible to quantify a priori the qualities that make them desirable. This lack of a suitable objective function rules out the use of inverse design.

In this paper we present a third methodology for computer-assisted parameter setting that is especially applicable to graphics processes that exhibit one or both of these characteristics. *Design Gallery* (DG) interfaces present the user with the broadest selection — automatically generated and organized — of perceptually different graphics or animations that can be produced by varying a given input-parameter vector. Because the selection is generated automatically, it can be done as a preprocess so that any high computational costs are hidden from the user. Furthermore, the DG approach requires only a measure of similarity between graphics, which can often be quantified even when optimality cannot.

A DG system comprises several key elements. The *input vector* is a list of parameters that control the generation of the output graphic via a *mapping* process. The *output vector* is a list of values that summarizes the subjectively relevant qualities of the output graphic. The *distance metric* on the space of output vectors approximates the perceptual similarity of the corresponding output graphics. The *dispersion* method is used to find a set of input vectors that map to a well-distributed set of output vectors, and hence output graphics. The dispersed graphics are presented to the

user through a perceptually reasonable *arrangement* method that makes use of the distance metric. These six elements — input vector, mapping, output vector, distance metric, dispersion, and arrangement — characterize a DG system. The creator of a DG system chooses the input vector, output vector, and the distance metric for a specific mapping process. For particular instances of the process, the computer performs the dispersion, the mapping of input vectors to output vectors, and the arrangement of final graphics in a gallery. The end user need only recognize and select appealing graphics from the gallery.

We explain and illustrate the use of DGs for several common parameter-setting problems: light selection and placement for image rendering, both standard and image-based; opacity and color transfer-function specification for volume rendering; and motion control for particle-system and articulated-figure animation. During the discussion, we describe the input and output vectors for each mapping process, and present various methods for dispersion and arrangement that we have used in building DG systems.

2 Light Selection and Placement

Setting lighting parameters is an essential precursor to image rendering. Previous attempts at computer-assisted lighting specification have used inverse design. For example, the user can specify the location of highlights and shadows in the image [14], or pixel intensities [18], or subjective impressions of illumination [10]; the computer can then determine lighting parameters that best meet the given objectives, using geometric [14] or optimization [10, 18] techniques. Unfortunately, the formulation of lighting specification as an inverse problem has some significant drawbacks. High-quality image rendering (e.g., raytracing or radiosity) is costly; to make the computer's search task tractable, the user may have to fix the light positions [10, 18], thereby grossly limiting the illuminations that can be considered. A more intrinsic difficulty is that of requiring the user to quantify a priori the desired illuminative characteristics of the resulting image. This requirement may be satisfiable in an architectural context [10], but seems very challenging in a more general cinematographic context [8]. The most difficult lighting parameters to set are those relating to light type and placement, so they have been the focus of our efforts.

2.1 Input and Output Vectors

For the light selection and placement problem, we begin with a scene model comprising surfaces and viewing parameters. The goal is to explore different ways of lighting the scene, so the input vector includes a light position, a light type, and a light direction if needed. The light position is located somewhere on one of the surfaces distinguished as a *light hook* surface by the user. The light type comes from a user-defined group, and describes attributes of the light: its basic class (e.g., point, area, or spotlight); whether or not it casts shadows; its falloff behavior (e.g., none, linear, or quadratic); and class-specific parameters (e.g., the beam angle of a spotlight). Directional lights are aimed at randomly chosen points on designated *light target* surfaces.

The output vector should be a concise, efficiently computed set of values that summarizes the perceptual qualities of the final image. Thus, output vectors are based on pixel luminances from several low-resolution thumbnail images (e.g., 32×25 pixels and smaller). The luminances at resolution ρ are weighted by a factor $f(\rho)$. The distance metric on the output vector is the standard L^1 (Manhattan) distance. As a result, the distance between output vectors corresponding to images q and r is

$$\sum_{\rho \in \{1, \frac{1}{16}, \frac{1}{16^2}, \ldots\}} \sum_{x,y} f(\rho) |Y_q^\rho(x,y) - Y_r^\rho(x,y)| \qquad ,(1)$$

where $Y_q^\rho(x,y)$ is the luminance of the pixel at location (x,y) in image q at resolution ρ.[1]

2.2 Dispersion

The dispersion phase selects an appropriate subset of input vectors from a random sample over the input space. Specifically, T lights are generated at each of H positions distributed uniformly over the light hook surfaces. This gives a set L of $H \times T$ input vectors. Typical values are $H = 500$ and $T = 8$, in which case $|L| = 4000$.[2] For each input vector in L, thumbnail images are generated, and the corresponding output vector is determined as described above. The dispersion algorithm outlined in Figure 1 then finds a set $I \subset L$ with optimal spread among output vectors. The first step is the elimination of lights that dimly illuminate the visible part of the scene, e.g., because they are obscured: these lights are unlikely to be of interest to the user and can confound the rest of the dispersion process. Thumbnail images whose average luminance is less than a cutoff factor c are eliminated from the set L. (Typical useful values of c are in the range 1%–5% of the maximum luminance value.) The subset I is assembled by repeatedly adding to I the light in L whose output vector is most different from its closest match in the nascent I. The size of I is determined by the interface, described below; $|I| = 584$ for the examples we discuss in the paper.

2.3 Arrangement

We would like the set of lights I to be large, so that the user will have many complementary lights from which to choose. However, the greater the size of I, the more difficult it will be for the user to browse the lights effectively. We accommodate these contradictory requirements by arranging the set I in a hierarchy in which similar lights, i.e., those that produce similar illumination effects, are grouped together. This is the goal of the arrangement phase, and we accomplish it by graph partitioning. A complete graph is formed in which the vertices correspond to the lights in I, and edge costs are given by the inverse of the distance metric used in the dispersion phase. An optimal w-way partition of this graph would comprise w disjoint vertex subsets of equal cardinality such that the cost of the cut set, the total cost of all edges that connect vertices in different subsets, is minimized. Optimal graph partitioning is NP-hard [4], but many good

[1] Since we start with a low-resolution thumbnail (e.g., 32×25 pixels), the filtered images of even lower resolution called for in the expression will be truly tiny. Nevertheless, they do contain useful information: two barely nonoverlapping narrow-beam spotlights will generate a high (and somewhat misleading) difference score at the highest resolution, but smaller, more appropriate difference scores at lower resolutions because the beams will overlap in the lower-resolution images. The effect of the weighting function $f(\rho)$ is subtle, but we have found it preferable to weight higher-resolution images slightly more than lower-resolution ones.

[2] We picked these numbers to allow overnight batch processing of the entire DG process for one scene on a single MIPS R10000 processor.

Input:
L, a set of lights and corresponding thumbnail images.
$n < |L|$, the size of the selected subset.
c, an average-luminance cutoff factor.

Output:
$I \subset L$, a set of n dispersed lights and their images.

Procedure:

```
SELECTION_DISPERSE(L, n, c) {
    L ← L \ find_dims(c, L);
    I ← ∅;
    for i ← 1 to n do {
        p_score ← −∞;
        foreach q ∈ L do {
            q_score ← ∞;
            foreach r ∈ I do
                if image_diff(q, r) < q_score then
                    q_score ← image_diff(q, r);
            if q_score > p_score then {
                p_score ← q_score;
                p ← q;
            }
        }
        I ← I ∪ {p};
        L ← L \ {p};
    }
}
```

Notes:
find_dims(c, L) returns those lights in L with average luminance less than c.
image_diff(q, r) returns the value computed by Equation 1.

Figure 1: A selection-based dispersion heuristic.

heuristics have been developed for this problem [1]. Our partitioning code is based on an algorithm and software developed by Karypis and Kumar [9]. Once the initial w-way partition is formed, representative lights for each partition are selected, and installed in the hierarchy. The partitioned subsets, minus their representative vertices, are then processed recursively until a hierarchy with branching factor w and height h is completed.

The values for w and h are dictated by the user interface, whose structure is depicted in Figure 2. For each light in the final set I, medium- (128×100 pixels) and full-size (512×400 pixels) images are generated for use in the interface. The user is presented with a row of eight images that serve as the first level of the light hierarchy. Clicking on one of these images causes its eight children in the hierarchy to be presented in the next row of images. The third and final level in the hierarchy is accessed by clicking on an image from the second row. Thus $w = 8$ and $h = 3$. In turn, these parameters determine the cardinality of I: $|I| = \sum_{j=1}^{h} w^j = 584$. This particular interface provides additional application-specific functionality that exploits the additive nature of light [6]. Images can be dragged to the palette, where light intensity and temperature can be varied interactively. Multiple images are composited to form a full-size image in the lower left.

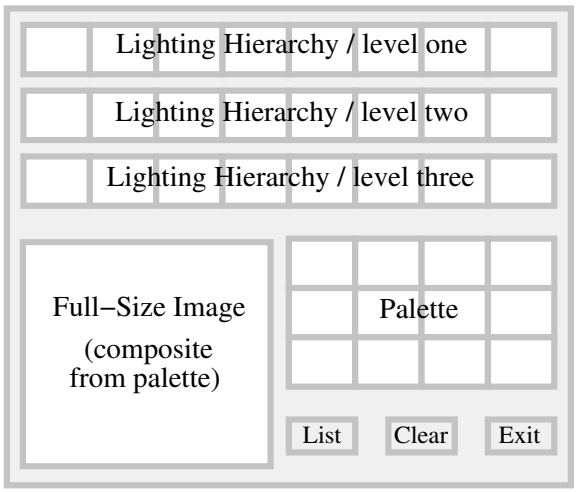

Figure 2: User-interface map.

2.4 Results

The DG in Figure 9 contains a scene inspired by an example from [8]. The floor, ceiling, and all four walls (only the rear one is visible) were designated light-hook surfaces. The surfaces comprising the figures were designated light-target surfaces, as was the back wall. The 584 lights in the gallery were selected from 5,000 randomly generated lights in the dispersion phase. The cost of computing this and the other light-selection-and-placement DGs shown here was dominated by the cost of raytracing the 584 full-size images used in the display, which took approximately five hours on a MIPS R10000 processor.

Figure 10 contains a scene with richer geometry. The ceiling, and the area around the base of the statue were designated light-hook surfaces. The surfaces of the two heads, the doors, the tree, and the statue were designated light-target surfaces. The gallery lights were selected from 3,000 randomly generated lights in the dispersion phase.

Finally, Figure 11 shows a DG for synthetic lighting of a photograph. A 3D model consisting of points and lines is recovered automatically from a triplet of images, each image being taken from a different viewpoint, as described in [2]. These points and lines are then aggregated into planes, and spurious geometry corrected or removed by hand. An illumination of the final recovered model is used to modulate intensity in one of the original photographic images.

3 Opacity and Color Transfer Functions for Volume Rendering

Choosing the opacity and color transfer functions for volume rendering is another tedious and difficult manual task amenable to a DG approach.[3] We developed DG interfaces for two data sets: the simulated electron density of a protein, and a CT scan of a human pelvis.

[3] The application of both interactive evolution and inverse design to this problem is the subject of [7].

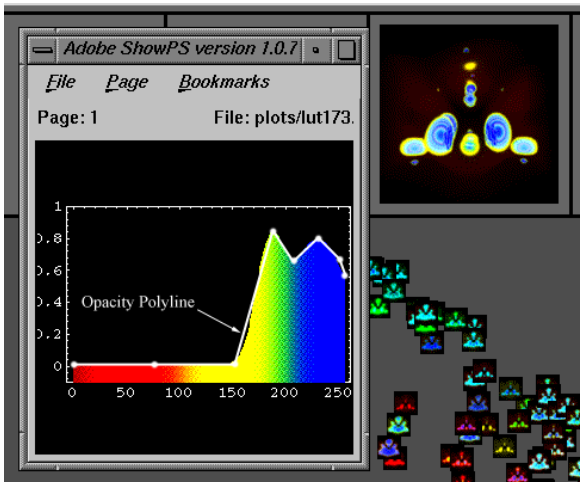

Figure 3: Pop-up display depicting opacity and color transfer functions.

3.1 Input and Output Vectors

The protein data set contains values in the interval $[0, 255]$. The opacity transfer function over this domain is parameterized by a polyline with eight control points, for a total of 16 values. The polyline is low-pass filtered before it is used. The color transfer function is parameterized by five values that segment the data into six subranges, which are arbitrarily assigned the colors red, yellow, green, cyan, blue, and magenta. Thus color is being used only to identify subranges of the data, and not to convey any quantitative relations among the data. Figure 3 illustrates a sample opacity and color transfer function. The complete input vector comprises 23 parameters.

For the scene-lighting DG, the output vector contains approximately 850 weighted pixel luminances. This kind of resolution is necessary because lights can cause completely local illumination effects in a synthetically rendered image, effects which should be representable in the output vector. By comparison, changes to transfer functions will generally affect many pixels throughout a volume-rendered image. We can take advantage of this homogeneity by including only a handful of pixels in the output vector. Currently we use eight pixels, selected manually for each data set. Representing all of their YUV values requires 24 values in the output vector, and standard Euclidean distance is used as the output-space metric. Dispersion on the basis of eight pixels from different parts of the image produces excellent dispersion of complete images at a much reduced computational cost.

3.2 Dispersion

The dispersion heuristic in Figure 1 works by distilling a set of randomly generated input vectors down to a well-dispersed subset. Although simple, this method has the drawback of not utilizing what is learned via random sampling about the mapping from input to output vectors. In contrast, the dispersion heuristic in Figure 4 uses an evolutionary strategy that adapts its sampling over time in response to what it implicitly learns, and consequently performs much better. It starts with an initial set of random input vectors. These vectors are then perturbed randomly. Perturbed vectors are substituted for existing vectors in the set if the substitution improves dispersion. The key notion of dispersion used is nearest-neighbor distance in the space of output vectors.

Input:
A random set of input vectors, I, and their corresponding output vectors, O. $|I| = |O| = n$.

A trial count, t.

Output:
Modified sets of input and output vectors, I and O.

Procedure:

```
EVOLUTION_DISPERSE(I, O, t) {
    for i ← 1 to t do {
        j ← rand_int(1, n);
        u ← perturb(I[j], i);
        map(u, v);
        k ← worst_index(O);
        if is_better(v, O[k], O) then {
            I[k] ← u;
            O[k] ← v;
        }
        else if is_better(v, O[j], O) then {
            I[j] ← u;
            O[j] ← v;
        }
    }
}
```

Notes:

$rand_int(1, n)$ returns a random integer in the range $[1, n]$.

$perturb(I[j], i)$ returns a copy of $I[j]$ in which all the elements have been perturbed. The magnitude of the perturbations is inversely proportional to i.

$map(u, v)$ maps input vector u to output vector v using an application-specific mapping process.

$worst_index(O)$ returns the index of the output vector in O with minimum nearest-neighbor distance. Ties are broken using the average distance to all other vectors in O.

$is_better(v, O[k], O)$ returns true if the nearest neighbor to v in $O \setminus \{O[k]\}$ is further away than the nearest neighbor to $O[k]$ in O. Ties are broken using average distance to all other vectors in the relevant set.

Figure 4: An evolutionary dispersion heuristic.

3.3 Arrangement

The arrangement method based on graph partitioning that was presented in §2.3 results in a simple and easy-to-use interface. Unfortunately, sometimes the partition contains anomalies, e.g., dissimilar lights placed in the same subset of the partition. This is due to limitations of the partitioning method (no heuristic partitioning strategy guarantees an optimal partition), and to the structure of the set of output vectors, which may not map well to any regular partition.

For the volume-rendering example, we used an alternative arrangement method that eschews a partition-based or hierarchical framework and instead illustrates the structure of the set of output vectors graphically in a 2D layout. An interface for this arrangement method is shown in Figure 5.

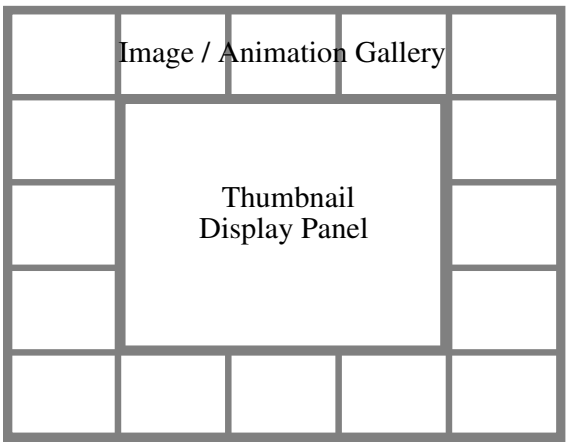

Figure 5: A more flexible user interface.

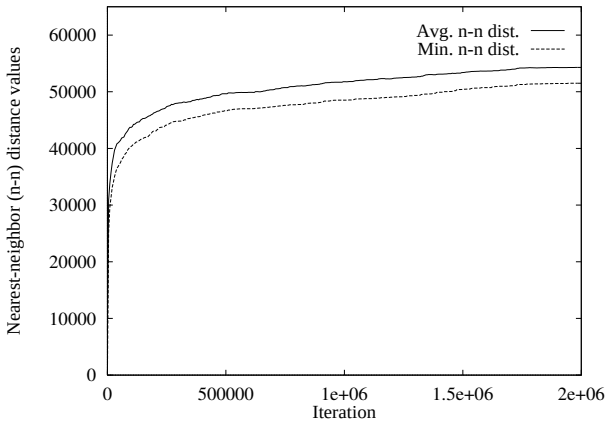

Figure 6: Nearest-neighbor distances using evolutionary dispersion.

A thumbnail, which in this case is a small, low-resolution volume-rendered image, is generated for each final output vector. The thumbnails are arranged in the center display panel, in a manner that correlates the distance between thumbnails with the distance between the associated output vectors. The thumbnail display panel can be panned and zoomed. Selecting a thumbnail brings up a full-size image, which can then be moved to the surrounding image gallery. Mousing on an image in the gallery highlights its associated thumbnail, and vice versa.

Thumbnail layout is accomplished using a multidimensional scaling (MDS) method due to Torgerson [3, 23].[4] Given a matrix of distances between points, MDS procedures compute an embedding of the points in a low-dimensional vector space (2D in our case) such that the interpoint distances in the embedding match closely those in the given matrix. Torgerson's "classical scaling" method, although less general and powerful than newer iterative methods, is fast and robust. When the interpoint distances come from an embedding of the points in a high-dimensional vector space (which is true for the applications we discuss here, although it need not be true in general), classical scaling is equivalent to an efficient technique for computing a principal-component analysis of the points [5].

The layouts computed by classical scaling are not without anomalies — as we are using it, this MDS method is a projection from a high-dimensional space onto a 2D space, which cannot be done without loss of information — but they do reflect the underlying structure of the output vectors well enough to allow effective browsing. One important practical detail: since full-size versions of all the images returned by the dispersion procedure must be rendered anyway, it is convenient and better to compute distances from these full-size images in the arrangement phase, instead of from the eight pixels used in the dispersion phase.

3.4 Results

Figure 12 illustrates the DG for the volume rendering of the protein data set. The dispersion procedure returned 256 dispersed input and output vectors. A selection of images is shown in the surrounding image galleries. The lines that connect images with their thumbnails give some indication of how images congregate in the thumbnail display. (During interactive use the association between thumbnails and images is done preferably by dynamic highlighting, as described above.) Figure 3 shows the result of clicking on one of the images in the image gallery: the corresponding opacity and color transfer functions are depicted in a pop-up window, allowing the user to see how image and data relate.

The performance of the dispersion heuristic from this experiment is documented in Figure 6; this data is representative of all of the DG experiments that use the evolutionary dispersion heuristic. The curves show how two values, the minimum and average nearest-neighbor distances in the set of output vectors, increase over time. Improvement is rapid at first: the minimum and average nearest-neighborhood distances in the initial random set are 184 and 7,789, respectively. However, the rate of improvement drops quickly. Although we used a trial count of $t = 2,000,000$ (see Figure 4), it is clear that relatively little improvement occurred after $t = 500,000$. To reach this point requires $8 \times 500,000 = 4,000,000$ raycast operations and takes less than 40 minutes on a single MIPS R10000 processor. This duration is roughly one-sixth of that needed to render the 256 full-size images (300×300 pixels) for the DG.

A second volume-rendering experiment was performed using a computed tomography (CT) data set for a human pelvis. These data values are presegmented into four disjoint subranges, one each for air, fat, muscle, and bone. The input vector specifies the y-coordinates of 12 opacity control points; the x-coordinates are held fixed. The input vector does not specify a color transfer function, since standard colors are used for the different tissue types. The output vector, distance metric, dispersion, and arrangement were identical to the protein-rendering experiment. Figure 13 illustrates the DG for the volume rendering of the pelvic data set.

4 Animation Applications

Motion control in animation is often fraught with parameter tuning, where the mapping from input parameters to graphical output is nonintuitive, unpredictable, and costly to com-

[4] The use of more sophisticated MDS techniques [3] for arranging a database of images is being investigated by Rubner et al. [17].

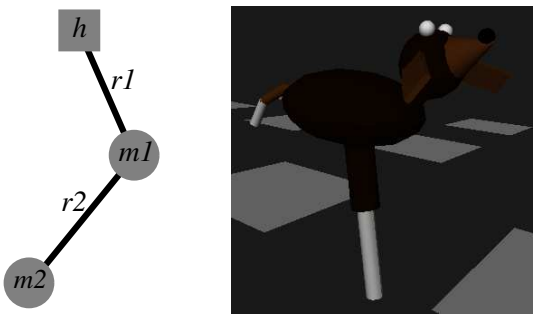

Figure 7: Articulated linkages.

pute.[5] For these reasons, motion control is very amenable to a DG approach. Building a DG interface for animation is similar to building one for still images (we reuse the dispersion and arrangement code from §3 virtually without change); the major differences are in computing the output-vector components. We now discuss three DG systems for animation tasks, focusing on this latter issue.

4.1 2D Double Pendulum

The 2D double pendulum is a simple dynamic system with rich behavior that makes it an ideal test case for parameter-setting methodologies.[6] A double pendulum consists of an attachment point h, two bobs of masses m_1 and m_2, and two massless rods of lengths r_1 and r_2, connected as shown in Figure 7. Our pendulum also includes motors at the joints at h and m_1 that can apply sinusoidal time-varying torques. The input vector comprises the rod lengths, the bob masses, the initial angular positions and velocities of the rods, and the amplitude, frequency, and phase of both sinusoidal torques, for a total of 14 parameters.

Choosing a suitable output vector proved to be the most difficult part of the DG process for the double pendulum, as well as for the other motion-control applications; several rounds of experimentation were needed (see §5 for more details). The output vector must capture the behavior of the system over time. For the double pendulum, the output vector has 12 parameters: the differences in rod lengths and bob masses, the average Cartesian coordinates of each bob, and logarithms of the average angular velocity, the number of velocity reversals, and the number of revolutions for each rod. Euclidean distance is used as the distance metric on this output space.

The mapping from input vector to output vector is accomplished by dynamically simulating 20 seconds of the pendulum's motion, and using the algorithm in Figure 4 for dispersion. Arrangement is accomplished using the MDS layout method of §3.3. The displayed thumbnails are static images of the final state of the pendulum, along with a trail of the lower bob over the final few seconds. We found that these images give enough clues about the full animation to enable effective browsing. Thumbnails can be dragged into gallery slots, all of which can be animated simultaneously by clicking on any occupied slot.

[5] Both interactive evolution [25] and inverse design [12, 13, 21, 24, 26] have been applied previously to motion control.

[6] Even without the application of external torques at its joints, the 2D double pendulum is provably chaotic [19].

Figure 7 shows the DG for the double pendulum. As before, the overlaid lines show where animations in the gallery are located in the thumbnail display. The plateau in nearest-neighbor distance is reached after 170,000 dispersion iterations, which take 6.5 hours on a single MIPS R10000 processor.

4.2 3D Hopper Dog

The previous DG is useful in finding and understanding the full range of motions possible for the pendulum under a given control regime. However, complete generality is not always a useful goal: the animator may have some preconceived idea of a motion that needs subtle refinement to add nuance and detail. The 3D hopper dog, shown in Figure 7, is an articulated linkage with rigid links connected by rotary joints. It has a head, ears, and tail, and moves by hopping on its single leg. It has 24 degrees of freedom (DOF). The hopper dog is actuated by a control system that tries to maintain a desired forward velocity and hopping height, as well as desired positions for joints in some of the appendages. The equations of motion for the system are generated using a commercially available package[16]; dynamic simulation is used to produce the animations.

We started with a basic hopping motion, and then used a DG approach to explore seven input quantities of interest to achieve stylistic, physically attainable gaits. The seven quantities are: the forward velocity, the hopping height, and the positions of 2-DOF ear joints, a 2-DOF tail joint, and a 1-DOF neck joint. For each of these, a time-varying sinusoid is chosen as the desired trajectory, with the minimum value, maximum value, and frequency specified in the input vector, which therefore contains 21 values.

In this particular case, the elements of the output vector correspond closely to those of the input vector. The 14-element output vector contains the averages and variances of the same seven quantities, and is obtained by dynamically simulating 30 seconds of the hopper dog's motion. (Output vectors from simulations in which hopper dog falls are automatically discarded.) As for the previous two applications, the output-space distance metric is Euclidean, and the arrangement method and interface from §3.3 are used. The hopper-dog DG is illustrated in the video proceedings.

4.3 Particle Systems

Particle systems are useful for modeling a variety of phenomena such as fire, clouds, water, and explosions [15]. A useful particle-system editor might have 40 or more parameters that the animator can set, so achieving desired effects can be tedious. As in the previous subsection, we use a DG interface to refine an animator's rough approximation to a desired animation.

The subject for our experiment is a hypothetical beam weapon for NASA space shuttles. A first draft was produced by hand using a regular particle-system editor; a still from midway through the animation is shown in Figure 8. The input vector contains the subset of particle-system controls that the animator wishes to be tweaked. The controls govern: the mean and variance of particle velocities, particle acceleration, rate of particle production, particle lifetime, resilience and friction coefficients of collision surfaces, and perturbation vectors for surface normals. Among the parameters that are held fixed in this example are the origin, average direction, and color of the beam.

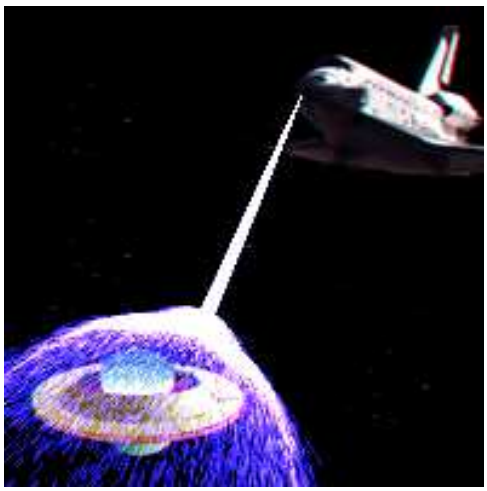

Figure 8: A still from a particle-system animation.

For efficiency reasons, DG output vectors are based on subsampled versions of the final graphic where possible, thereby reducing computational costs and allowing more of the space to be explored. For example, static images can be rendered at low resolution (§2 and §3). The subsampling strategy for the particle animation is to simulate only every 500th particle generated during the dispersion phase, and to examine the state of the particle system at just two distinct points in time: once midway through the simulation, and once at the end. The output vector comprises measures of the number of particles, their average distance from the origin and the individual variation in this distance, their spread from the average beam, the average velocity of the entire system, and the individual variation from this average (we take logs of all of these quantities except for the beam spread). These six measures are included for each of the two distinguished times, resulting in 12 output parameters. Euclidean distance is the metric on the output space.

The dispersion and arrangement methods from §3 are used to generate the DG. Each thumbnail is the midway still from the corresponding animation. (The user can optionally select thumbnails from different stages in the animation.) As with the double-pendulum DG, thumbnails can be dragged to gallery slots and animated therein. Figure 15 shows the DG of variations on the animator's original sketch from Figure 8. Also as before, lines connect animation stills with their associated thumbnails. The dispersion heuristic ran for $t = 100,000$ iterations, at which point it appeared to reach a plateau. This number of trials took approximately six hours on a MIPS R10000 processor. Generating the 256 animations in the DG with their full complement of particles took a little under five hours on the same processor.

5 Discussion

Table 1 summarizes the DGs described in this paper, in terms of the six basic elements of a DG system. Some of the variation in this table is application specific, while the remainder stems from our investigation of alternative dispersion and arrangement methods. All of the galleries described in the paper produce a useful variety of output graphics.

Using a DG for a particular instance of a design problem is fairly straightforward for the end user. Aside from browsing the final DG, the user's only other task is to loosely focus the dispersion process by, for example, selecting suitable light-hook and light-target surfaces (§2), or by specifying a relevant subset of particle-control parameters (§4.3). However, creating a DG system for an entire class of design problem is more difficult. The DG-system creator is responsible for choosing the structure of the input and output vectors, and the distance metric on the output space. Thus, the creator needs a better understanding of the design problem than the end user. Of the creator's tasks, the simplest is choosing the distance metric: very standard metrics sufficed for all applications we tried. Choosing the input vector is also straightforward. Even when there are many possible ways to parameterize the input, our experience is that choosing an acceptable parameterization is not hard.

The most difficult task of the DG-system creator is devising an output vector. The first two DGs in Table 1 work on static images. In these examples, the perceptual similarity between images correlates well with subsampled image or pixel differences, hence the output vectors comprise subsampled image and pixel values. An added advantage is that the ranges of all components of the output vector are bounded and known. Finding measures that capture the perceptual qualities of a complete animation is harder. The DG systems for animation tasks required several experiments to get a suitable output vector, although the process became easier for each successive system. Among the lessons learned in developing output vectors for motion-control problems, the two most important precepts are, with hindsight, fairly obvious:

- Take the log of quantities that have a large dynamic range. For many such quantities, e.g., velocity, human ability to resolve changes in magnitude diminishes as the magnitude increases. To uniformly sample the perceptual space, one must therefore sample the lower end of the dynamic range more thoroughly.

- The relative weights of the output-vector parameters matter. In general, the output-vector parameters should be scaled so that they each have approximately the same dynamic range, otherwise only the parameters with the largest ranges will be dispersed effectively.

What inevitably happened with a poorly chosen output vector was that the dispersion algorithm found a malicious way to get unfortunate and unexpected spread in one of the vector coordinates, usually through a degenerate set of input parameters, e.g., pendulums with extremely short links and very high rpm's, and particle systems with only a few particles, but very high variance in velocity.

In our experiments, we investigated two dispersion methods and two arrangement methods. The dispersion method of Figure 4 is more complex, but performs better. However, an advantage of the simpler method in Figure 1 is that it may be easier to parallelize. Two arrangement methods were also tried, one based on graph partitioning and the other on MDS. Both allowed the user to navigate through the output graphics effectively, and both had their fans among our group of informal testers. Layout and organizational anomalies were evident in both interfaces, but they did not hinder the user's ability to peruse the output graphics.

6 Conclusion

Design Gallery interfaces are a useful tool for many applications in computer graphics that require tuning parameters to

Application	Light selection & placement	Volume rendering	Double pendulum	Particle system	Hopper dog
Input Vector	Light type, location, and direction	Control points for opacity/color transfer functions	Pendulum dimensions, initial conditions, motor torques	Animator-specified subset of particle control parameters	Desired trajectory sinusoids
Output Vector	Luminances of thumbnail pixels	YUV values for eight pixels	Trajectory statistics (mainly logs of time averages and variances)		
Distance Metric	Manhattan	Euclidean			
Mapping	Raytracing	Volume rendering	2D dynamic simulation	3D particle simulation	3D dynamic simulation
Dispersion	Selection from random sample over neighborhood	Evolution from full random sample		Evolution from random sample over neighborhood	
Arrangement	Graph partitioning	Multidimensional scaling			

Table 1: Summary of Design Gallery experiments.

achieve desired effects. The basic DG strategy is to extract from the set of all possible graphics a subset with optimal coverage. A variety of dispersion and arrangement methods can be used to construct galleries. The construction phase is typically computationally intensive and occurs off-line, for example, during an overnight run. After the gallery is built, the user is able to quickly and easily browse through the space of output graphics.

Inverse design is one technique for setting parameters, but it is only feasible when the user can articulate or quantify what is desired. DGs replace this requirement with the much weaker one of quantifying similarity between graphics. Unlike interactive evolution, DGs are feasible even when the graphics-generating process has high computational cost. Finally, DGs are useful even when the user has absolutely no idea what is desired, but wants to know what the possibilities are. This is often the first step in the creative design process.

7 Acknowledgments

The protein data set used in §3 was made available by the Scripps Clinic, La Jolla, California. Volume rendering was performed using the VolVis system from SUNY Stony Brook, as modified by T. He. R. Kaplan, a. shelat, Y. Siegal, and R. Surdulescu provided helpful comments and code. The university participants were supported in part by grants from MERL and from the National Science Foundation (Grant No. IRI-9350192).

References

[1] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: a survey. *Integration: The VLSI Journal*, 19:1–81, 1995.

[2] P. Beardsley, A. Zisserman, and D. Murray. Sequential updating of projective and affine structure from motion. *International Journal of Computer Vision*, 1997. In press.

[3] I. Borg and P. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, 1997.

[4] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[5] J. C. Gower. Some distance properties of latent root and vector methods used in multivariate analysis. *Biometrika*, 53:325–338, 1966.

[6] P. Haeberli. Synthetic lighting for photography. URL http://www.sgi.com/grafica/synth/index.-html, Jan. 1992.

[7] T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of transfer functions with stochastic search techniques. In *Proc. of Visualization 96*, pages 227–234, San Francisco, California, Oct. 1996.

[8] J. Kahrs, S. Calahan, D. Carson, and S. Poster. Pixel cinematography: a lighting approach for computer graphics. Notes for Course #30, SIGGRAPH 96, New Orleans, Louisiana, Aug. 1996.

[9] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical report, Dept. of Computer Science, Univ. of Minnesota, 1995. See also URL http://www.cs.umn.edu/~karypis/-metis/metis.html.

[10] J. K. Kawai, J. S. Painter, and M. F. Cohen. Radioptimization – goal-based rendering. In *SIGGRAPH 93 Conf. Proc.*, pages 147–154, Anaheim, California, Aug. 1993.

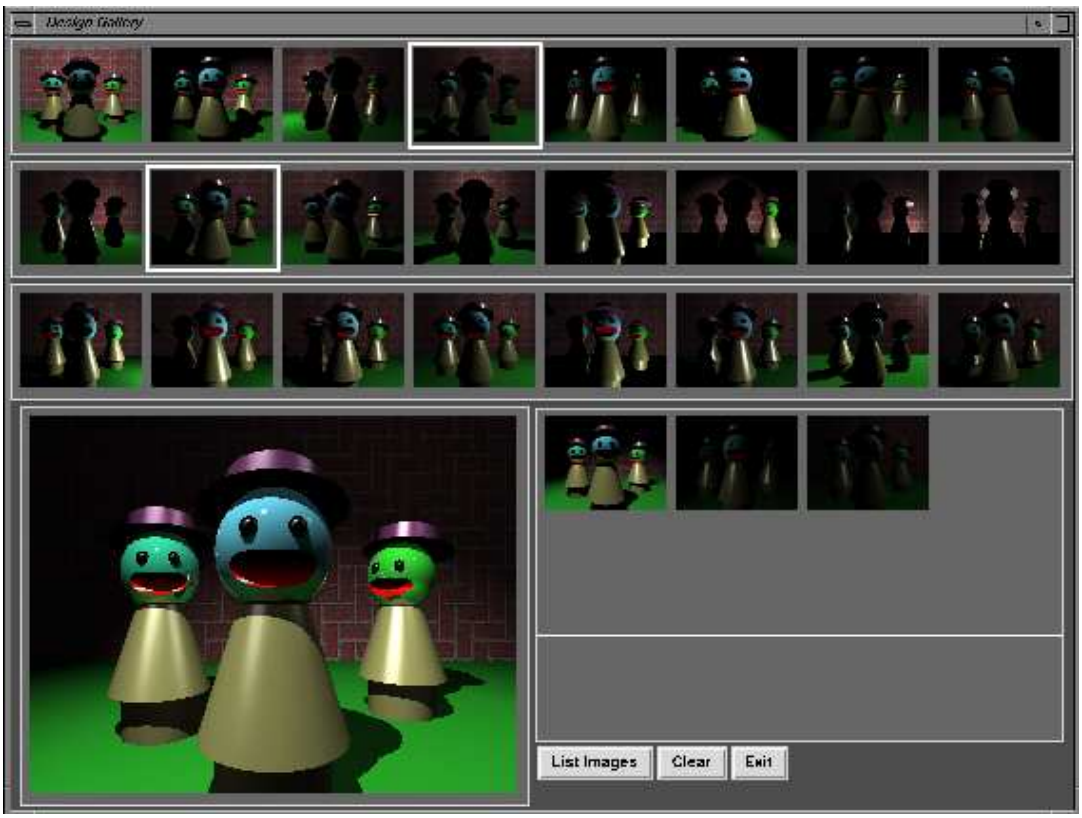

Figure 9: A DG for light selection and placement.

[11] S. Kochhar. A prototype system for design automation via the browsing paradigm. In *Proc. of Graphics Interface 90*, pages 156–166, Halifax, Nova Scotia, May 1990.

[12] Z. Liu, S. J. Gortler, and M. F. Cohen. Hierarchical spacetime control. In *SIGGRAPH 94 Conf. Proc.*, pages 35–42, Orlando, Florida, July 1994.

[13] J. T. Ngo and J. Marks. Spacetime constraints revisited. In *SIGGRAPH 93 Conf. Proc.*, pages 343–350, Anaheim, California, Aug. 1993.

[14] P. Poulin and A. Fournier. Lights from highlights and shadows. In *Proc. of the 1992 Symposium on Interactive Graphics*, pages 31–38, Boston, Massachusetts, Mar. 1992. In *Computer Graphics* 25(2), 1992.

[15] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. on Graphics*, 2:91–108, Apr. 1983.

[16] D. E. Rosenthal and M. A. Sherman. High performance multibody simulations via symbolic equation manipulation and Kane's method. *Journal of Astronautical Sciences*, 34(3):223–239, 1986.

[17] Y. Rubner, L. Guibas, and C. Tomasi. The earth mover's distance, multi-dimensional scaling, and color-based image retrieval. In *Proc. of the ARPA Image Understanding Workshop*, May 1997.

[18] C. Schoeneman, J. Dorsey, B. Smits, J. Arvo, and D. Greenberg. Painting with light. In *SIGGRAPH 93 Conf. Proc.*, pages 143–146, Anaheim, California, Aug. 1993.

[19] T. Shinbrot, C. Grebogi, J. Wisdom, and J. A. Yorke. Chaos in a double pendulum. *American Journal of Physics*, 60(6):491–499, 1992.

[20] K. Sims. Artificial evolution for computer graphics. In *Computer Graphics (SIGGRAPH 91 Conf. Proc.)*, volume 25, pages 319–328, Las Vegas, Nevada, July 1991.

[21] K. Sims. Evolving virtual creatures. In *SIGGRAPH 94 Conf. Proc.*, pages 15–22, Orlando, Florida, July 1994.

[22] S. Todd and W. Latham. *Evolutionary Art and Computers*. Academic Press, London, 1992.

[23] W. S. Torgerson. *Theory and Methods of Scaling*. Wiley, New York, 1958. See especially pages 254–259.

[24] M. van de Panne and E. Fiume. Sensor-actuator networks. In *SIGGRAPH 93 Conf. Proc.*, pages 335–342, Anaheim, California, Aug. 1993.

[25] J. Ventrella. Disney meets Darwin – the evolution of funny animated figures. In *Proc. of Computer Animation 95*, pages 35–43, Apr. 1995.

[26] A. Witkin and M. Kass. Spacetime constraints. In *Computer Graphics (SIGGRAPH 88 Conf. Proc.)*, volume 22, pages 159–168, Atlanta, Georgia, Aug. 1988.

Figure 10: Another DG for light selection and placement.

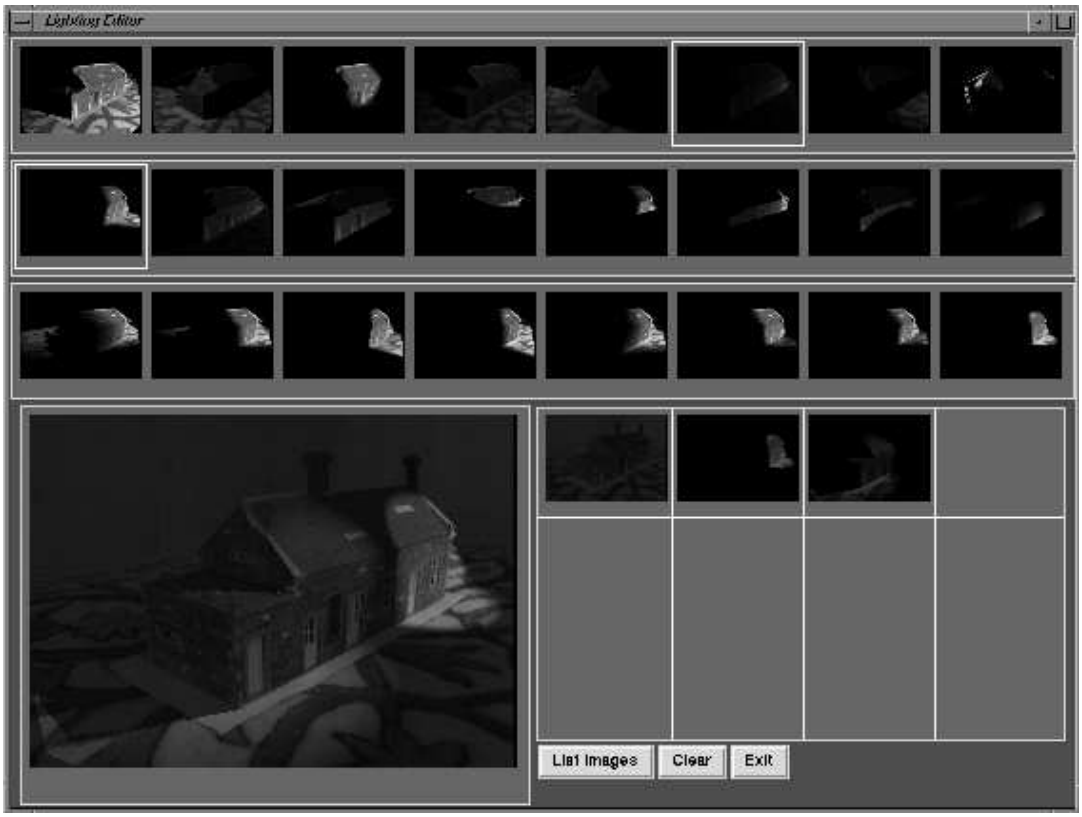

Figure 11: Light selection and placement for synthetic lighting of a photograph.

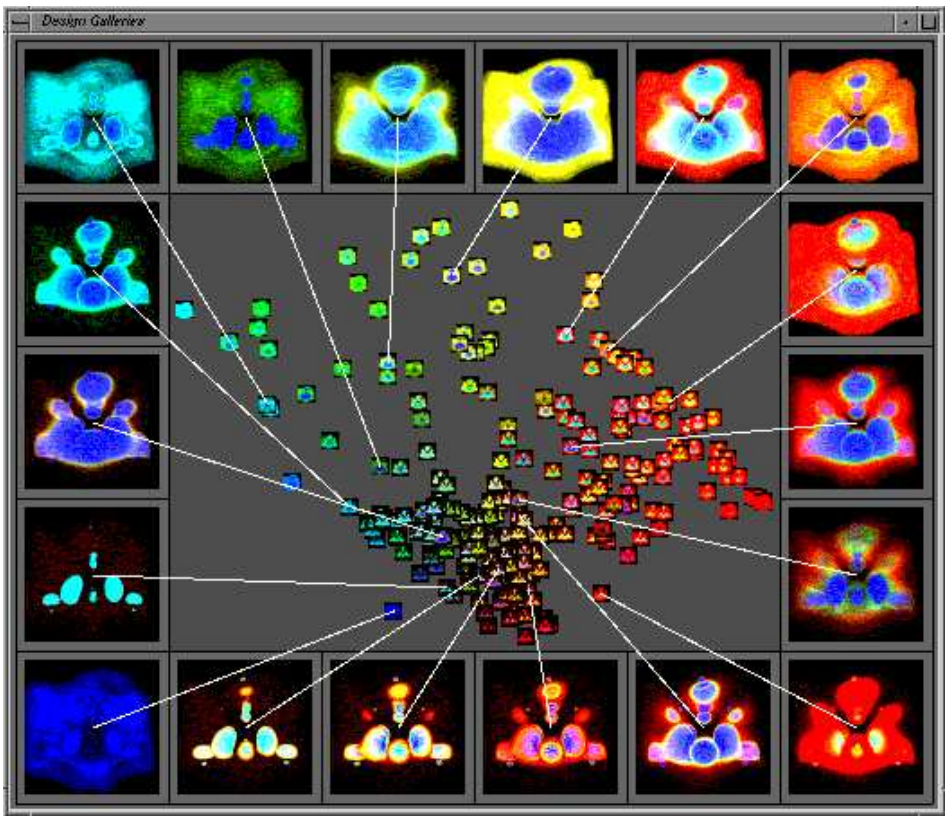

Figure 12: A DG with different opacity and color transfer functions.

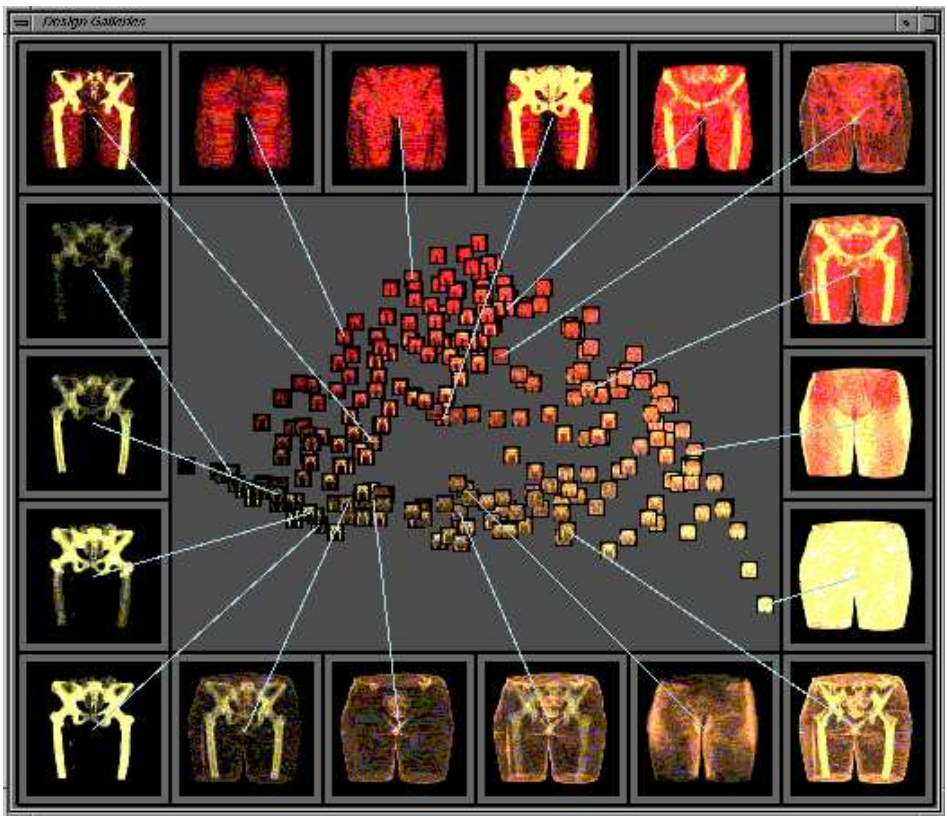

Figure 13: A DG with different opacity transfer functions.

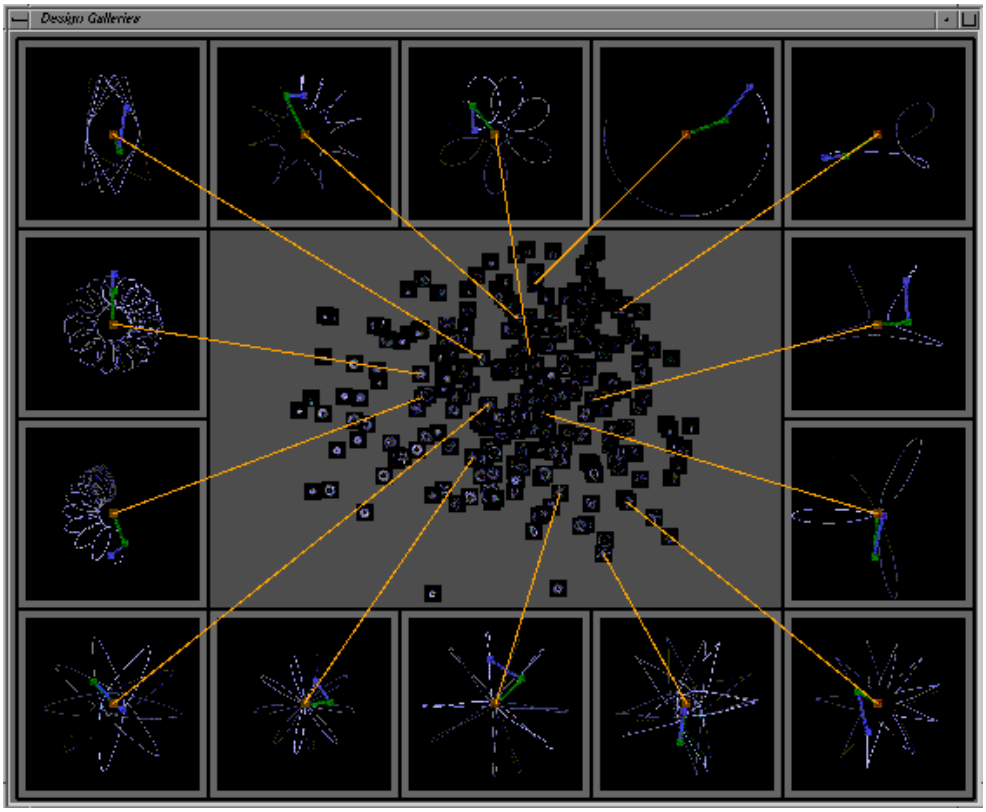

Figure 14: A DG for an actuated 2D double pendulum.

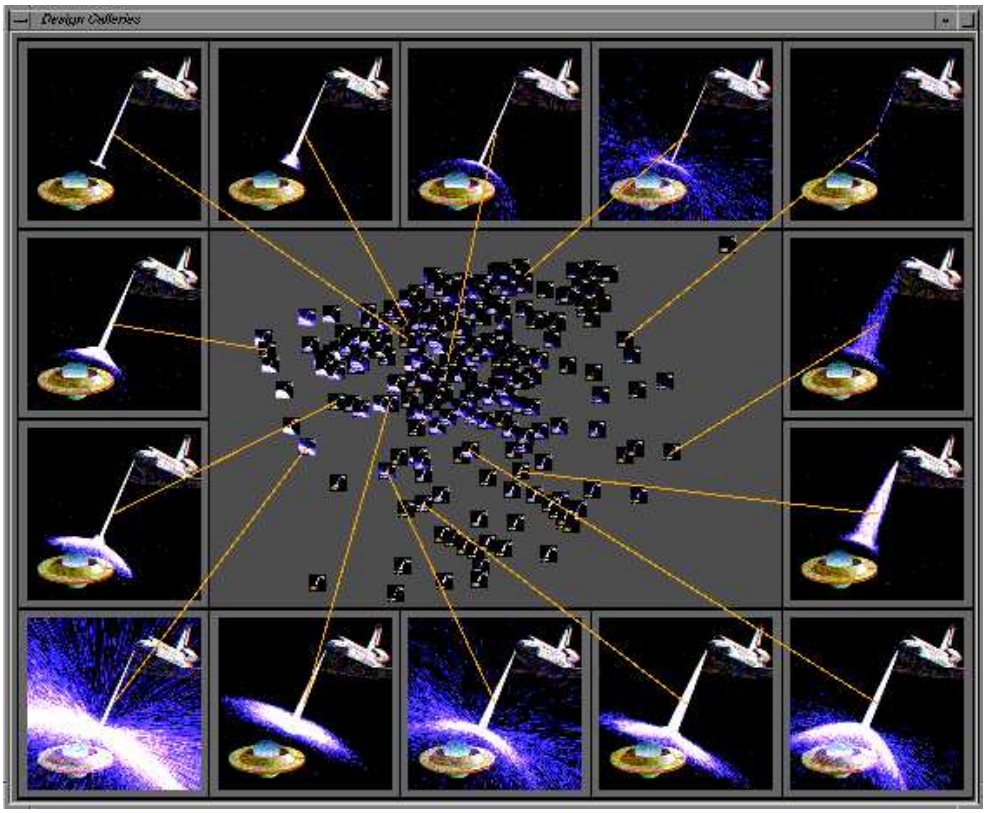

Figure 15: A DG for a particle system.