Andrew Glassner's Notebook

http://www.glassner.com

Interactive Pop-up Card Design, Part 2

Andrew Glassner

Pop-up cards are fun to create and receive. They're also a great output medium for computer graphics, offering an economical and compact way to show 3D scenes without the need for special glasses, shutters, or any other electronic hardware.

In my last column, I talked about the geometry behind two basic pop-up mechanisms: the single-slit and the V-fold. These are the heart of my interactive pop-up design assistant, which I use to design cards on the computer that I then print out and assemble.

Happily, the single-slit and V-fold mechanisms are also among the most general of all techniques used in pop-ups, since many of the other constructions are combinations of these mechanisms or variations on their geometry.

Of course, a variety of pop-up mechanisms exist that aren't captured by those ideas. Happily, most of those are straightforward to design and create with special-purpose code and don't present the sort of design challenges that the slit and V-fold designs do. We'll see some of them later.

Building an assistant

In this section, I'll describe how I put together my pop-up design assistant. Let's begin with a quick review of the relevant geometry from my January/February 2002 column.

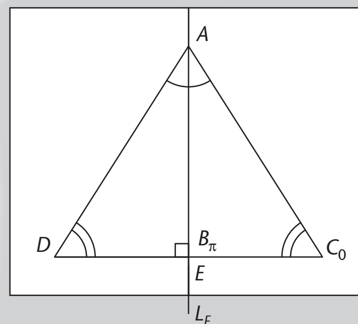
Figure 1 shows the basic geometry of a single slit from that column. To briefly recap, recall that plane π_1 doesn't move, while plane π_2 rotates around the central fold L_F by an angle ω. This tells us how to find point C_ω. The only missing point is B_ω, which we find by intersecting three spheres. Writing (A, r) for a sphere with center A and radius r, the spheres are $(A, |AE|)$, $(D, |DE|)$, and $(C_\omega, |DE|)$.

To create a V-fold, we only need to generalize this a little bit. Figure 2 shows the new geometry, where B_ω no longer lies next to E. Remembering that B_0 is the location of point B_ω when $\omega = 0$, we can find B_ω by intersecting three spheres with the same locations as the single slit but with slightly different radii. The spheres are $(A, |AB_0|)$, $(D, |DB_0|)$, and $(C_\omega, |DB_0|)$.

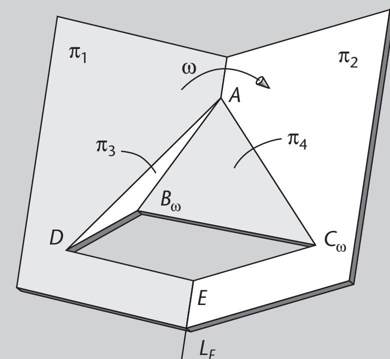
That's the essence of the geometry. What remains are the data structures, algorithms, and routines to evaluate the geometry. Let's take these in turn.

The heart of any program lies in its data structures. For my assistant, the most important data structure is the *riser*. A riser contains the information we need to position all the points of a single-slit or V-fold element.

The riser's first job is to help us identify points A, D, and C_ω. I represent each point with three pieces of information: a pair of coordinates, a pointer to another riser, and a flag. To position a point, I retrieve the pointed-to riser and select either the left or right side as specified by the flag. I then use the central fold and the appropriate bottom edge as two vectors that span a plane, scale those vec-

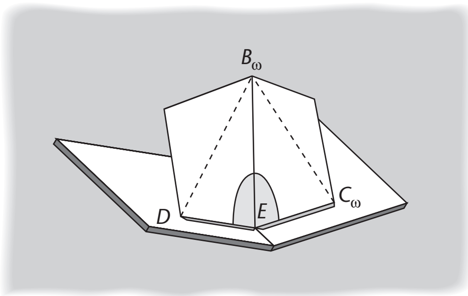


(a)

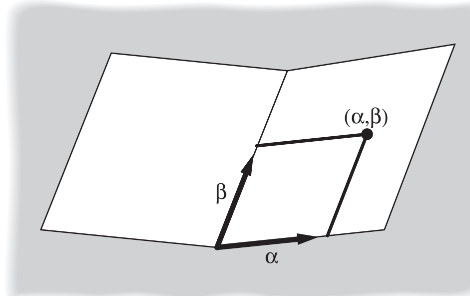


(b)

1 Geometry for the single-slit mechanism.



2 The basic V-fold is a generalization of the single slit.



3 To find a point on a riser, I use the coordinates (α, β) associated with that riser to scale the central fold and one of the lower edges, respectively. The point is the sum of these two scaled vectors.

tors by the coordinate values, and add those results together to find the sought-after point's position. Figure 3 shows this idea. This recursion of risers pointing to other risers ends with a special riser marked as the *card*.

I maintain all of the risers in a list. I add new risers to the end of this list as they're created (and of course remove them when the designer eliminates them from the model).

To process the risers, I start at the beginning of the list and look for a riser that can be positioned; that is, both risers it depends on are already positioned, and it isn't already positioned itself. If I find such a riser, I compute its points and mark it as positioned. When I reach the end of the list, if I positioned any riser on that pass, I go back to the start and go through the list again.

When I pass through the list but nothing gets positioned, then every riser should be accounted for. I scan the list once more as a check and look for unpositioned risers. If I find any, I report an error.

On the first pass through the list I can only position the card itself. Special-purpose code handles the card, because it doesn't depend on any other risers. I mark it as positioned and then continue running through the list. Because the risers are strictly hierarchical, this algorithm should always produce a completely positioned card.

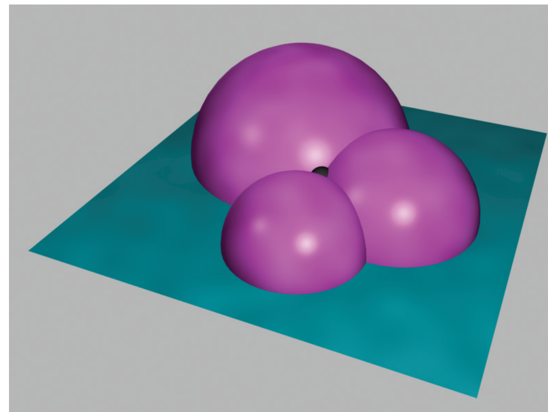
The two coordinates associated with each point describe the scale factors on the edges of the riser on which the point depends. One of those edges is always the riser's central edge. The other is the left or right bottom edge, as selected by the left or right flag.

The system must reposition the entire card every time the designer changes the opening angle, which occurs frequently. If efficiency is an issue, you can preprocess the list and build a tree structure that you can later traverse in a single pass. I found that repeatedly running through the list of a dozen or so risers was no problem for my 800-MHz Pentium III PC to handle in real time.

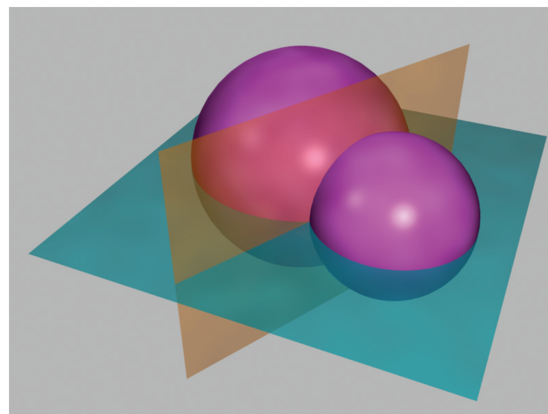
Carrying the geometric ball

Last time I described my algorithm for finding B_ω as the intersection of three spheres. Here are the details behind the heart of the routine.

Suppose we have three spheres, S_1, S_2, and S_3, with radii r_1, r_2, and r_3. The radii needn't be different, but they generally will be. Figure 4 shows three such spheres, and the plane that contains their centers. I've also marked in black one of their intersection points; there's another point symmetrically placed on the other side of the plane.



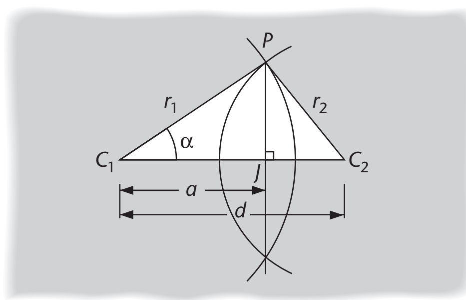
4 Three spheres that meet in a single point.



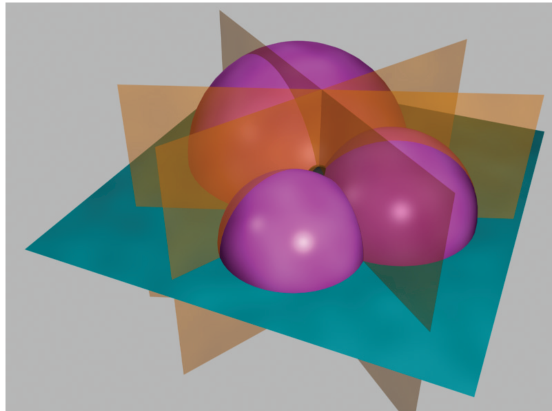
5 The points common to two intersecting spheres lie in a plane that's perpendicular to the line that joins their centers.

We'll work with these spheres in pairs. It doesn't matter where we start, so let's pick S_1 and S_2. When two spheres intersect, the points in common form a circle. Our first goal is to find the plane that contains that circle (see Figure 5).

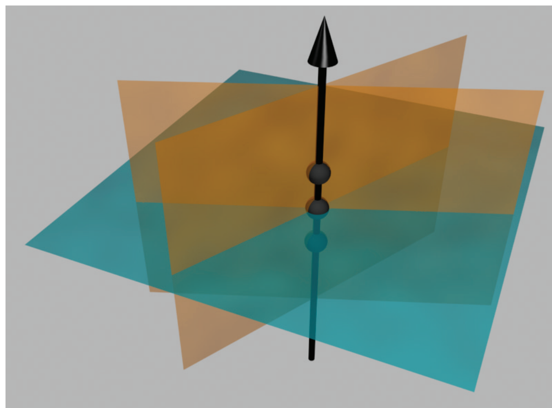
We saw last time how to find the point J on the line that contains the two sphere centers, as in Figure 6 (next page). To recap, the sphere centers are C_1 and C_2, their radii are r_1 and r_2, respectively, and they're distance d apart. We find J by finding the distance a. We see from triangle PJC_1 that $a = r_1 \cos \alpha$. We find $\cos \alpha$ from the Law of Cosines:



6 The geometry of Figure 5. We are given the centers C_1 and C_2, the radii r_1 and r_2, and the distance d and use this information to find point J that's contained in the plane.



7 The three intersection planes created by the three spheres.



8 Intersecting two planes gives us a line.

$$\cos\alpha = (d^2 + r_1^2 - r_2^2)/(2r_1 d)$$

so $a = (d^2 + r_1^2 - r_2^2)/(2d)$. Using these values for a and d, we can find

$$\begin{aligned} J &= C_1 + (C_2 - C_1)(a/d) \\ &= C_1 + (C_2 - C_1)\frac{d^2 + r_1^2 - r_2^2}{2d^2} \end{aligned}$$

Our plane passes through J with a normal parallel to $C_2 - C_1$. Let's call this plane π_{12}. I package this all up in a routine that takes as input two spheres and returns their plane of intersection.

We can repeat this for the other two pairs of spheres, creating planes π_{23} and π_{13}. Figure 7 shows all three planes.

Now I intersect any two of these planes to find the line they have in common (as shown in Figure 8). For numerical stability, I use a robust technique published by Jim Blinn (see the "Further Reading" section for a citation). This algorithm takes as input two planes represented with homogeneous coordinates and returns their line of intersection, if there is one (if the planes are parallel, they don't intersect and there's no common line).

Since Blinn's paper presents the theory, I'll just summarize the necessary equations here. The input to the algorithm is two planes, which I'll call P and Q. Plane P has a normal given by (P_x, P_y, P_z) and an offset P_d; plane Q is similar. The output is a line defined by a point B and a direction vector $\mathbf{V}$. The first step is to compute six handy terms p through u:

$$\begin{aligned} p &= P_z Q_d - P_d Q_z \\ q &= P_y Q_d - P_z Q_y \\ s &= P_x Q_d - P_d Q_x \\ t &= P_x Q_z - P_z Q_x \\ u &= P_x Q_y - P_y Q_x \end{aligned}$$

These tell us all we need to find the direction vector $\mathbf{V}$:

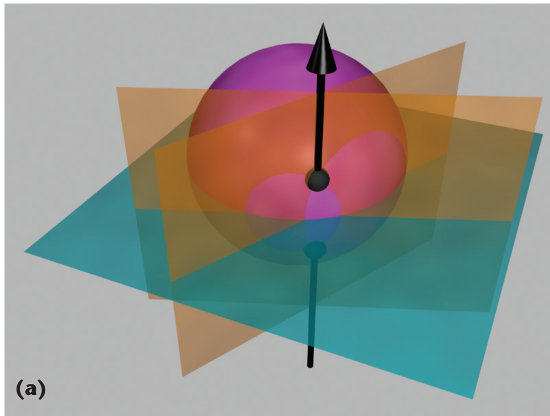
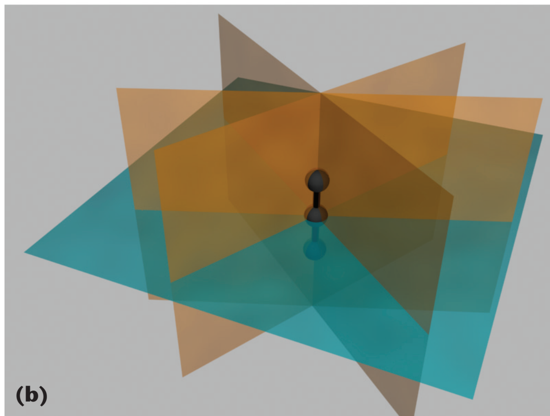
Further Reading

In my January/February 2002 column, I suggested a broad range of pop-up books to study, how-to books on their construction, and a few previous technical papers. All those references contributed to my understanding of the field. I encourage you to return to that column for a complete list. As a reminder, the three books that I've found most useful for discussions of technique are *The Elements of Pop-up* by David A. Carter and James Diaz (Simon & Schuster, 1999), *Paper Engineering* by Mark Hiner (Tarquin Publications, 1985), and *The Pop-up Book* by Paul Jackson (Henry Holt, 1993).

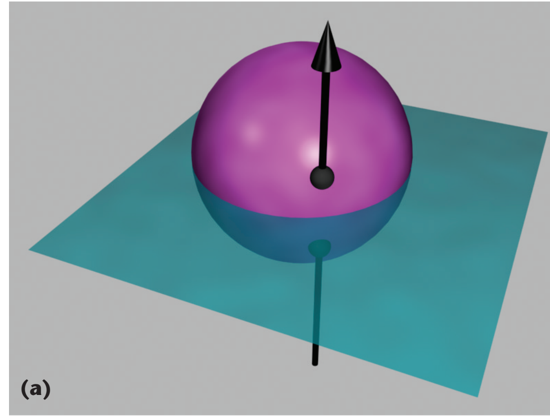
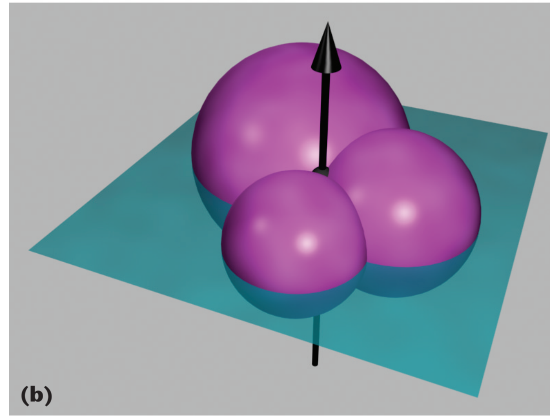
The solution I presented here for finding the line of intersection formed by two planes was published by Jim Blinn in his classic Siggraph 77 paper "A Homogeneous Formulation for Lines in 3-Space," in *Computer Graphics* (vol. 11, no. 2, 1977, pp. 237–241).

A good place to start to learn about packing algorithms in the textiles industry is "Placement and Compaction of Nonconvex Polygons for Clothing Manufacture," by V. Milenkovic, K. Daniels, and Z. Li, in the *Proc. 4th Canadian Conf. Computational Geometry* (1992, pp. 236-243).

For information on ray tracing, and how to find the intersection of lines with a variety of geometric objects including spheres, you can look at *An Introduction to Ray Tracing* by Glassner et al. (Academic Press, 1989) or the more recent *Practical Ray Tracing* by Peter Shirley (AK Peters, 2000).

9 **(a) Where *L* intersects the largest sphere. (b) In this application, each pair of planes results in the same line.**

10 **(a) The points of intersection of *L* with the largest sphere. (b) That intersection point is common to all three spheres.**

$$\mathbf{V} = (r, -t, u)$$

I then normalize **V**—that is, I scale it so that it has a length of 1.0. Now we can find the base point B. We have three cases that handle any degeneracies and special cases:

	if $r \neq 0$ then $B = (0, p/r, -q/r)$
else	if $t \neq 0$ then $B = (p/t, 0, -s/t)$
else	if $u \neq 0$ then $B = (q/u, -s/u, 0)$
else	error: planes are parallel

Now we have the line in a convenient parametric form: $L = B + t\mathbf{V}$, where the real number t sweeps us along all the points on the line L with direction **V** and passing through point B. In Figure 8, point B appears on the line where it pierces the plane.

It doesn't matter which pair of planes we choose to find L, since that line is common to all three planes, as shown in Figure 9. The line L is shared by all three pairs of planes. Here I've also shown where L intersects one of the spheres.

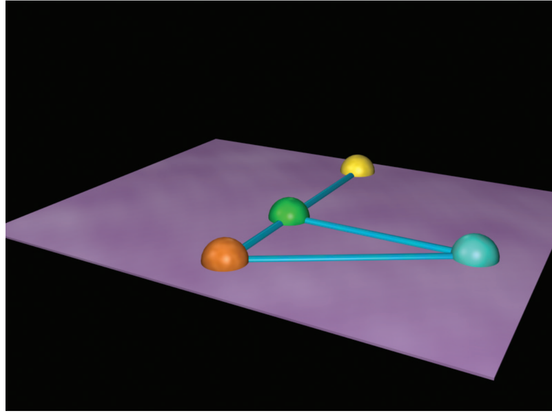
Now that we have a sphere and a parametric line, we can use standard ray-tracing techniques to intersect that line with any of the three spheres. I use a library function that takes a line and a sphere and returns the two points of intersection. I won't give the details of that calculation here, because they're available in every book on ray-tracing (see the "Further Reading" section). For numerical stability, I use the sphere with the largest radius. (If more than one sphere has the largest radius, I use one of them at random.)

The result is a pair of points, as Figure 10a shows. Which one do we want? That depends on whether the card designer wants the pop-up to rise out of the card's center or fall back behind it. We make this choice at design time, of course, and store it with the riser. I use the normal of the riser that's referenced by point D to categorize the two points. Each point will be on either the riser's positive or negative side. The designer's choice is stored in a flag associated with the riser. The most common case is where the pop-up rises from the card.

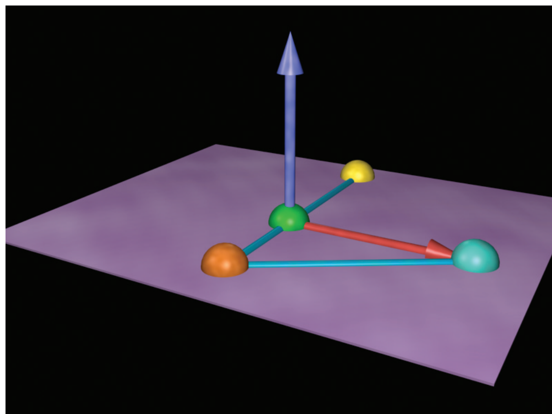
This line–sphere intersection point is of course shared by all three spheres, because it's their point of common contact; Figure 10b shows the line and spheres.

Another useful library routine is one that rotates a point around a given line by a given angle. This is useful when we want to position points on risers as they're moving around a fold axis. Suppose that we have a point P and a line given by the two points A and B, and we want to find point Q, the result of rotating P around line (A, B) by an angle θ.

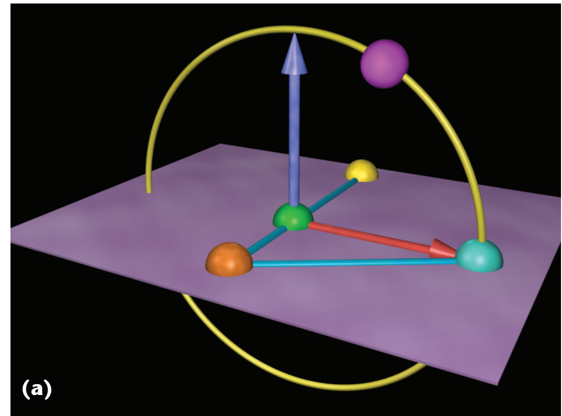
You can write some efficient code to do this if you need it. Here's a solution that uses only common vector

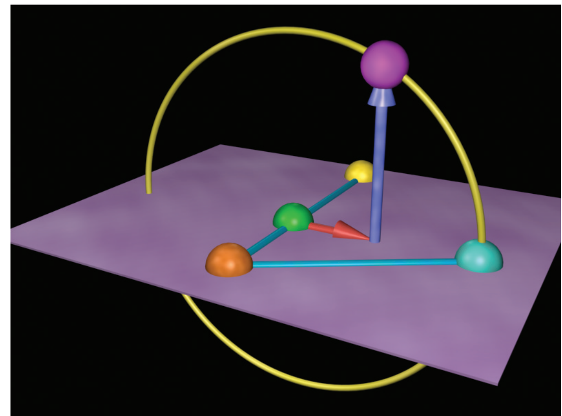


11 Rotating a point around a line. The line *AB* is shown by point *A* in orange and point *B* in yellow. The point to be rotated, point *P*, is in cyan off to one side. The point *M* in green is the point on *AB* that's closest to *P*.



12 The H vector is in red, and the V vector is in purple.





13 (a) The plane of rotation is formed by the H and V vectors. The yellow ring shows the circle in this plane with center *M* and radius $|MP|$. The new point *Q* (in red) lies on this circle. (b) We find *Q* as a sum of scaled versions of the H and V vectors.

operations that should be available in almost any 3D library. Figure 11 shows the basic setup.

We first find the point M which is the point on line AB, which is closest to P. I first define the vectors $\mathbf{C} = B - A$ and $\mathbf{D} = P - A$. Writing $\hat{\mathbf{C}}$ for the normalized version of $\mathbf{C}$, we find M by simply scaling $\mathbf{C}$ by the length of the projection of $\mathbf{D}$ onto $\mathbf{C}$:

$$M = A + \hat{\mathbf{C}}(\hat{\mathbf{C}} \cdot \mathbf{D})$$

where I'm using the dot product between $\hat{\mathbf{C}}$ and $\mathbf{D}$. Figure 11 shows point M in green.

Now I make two vectors $\mathbf{V}$ and $\mathbf{H}$ that span the plane that includes this line and point:

$$\begin{aligned} \mathbf{H} &= P - M \\ \mathbf{V} &= \mathbf{H} \times \mathbf{C} \end{aligned}$$

where $\times$ is the cross product. I also find the distance r from P to M:

$$r = |P - M|$$

Figure 12 shows the vector $\mathbf{H}$ in red and $\mathbf{V}$ in purple.

The new point Q is now easy to find. Essentially, we rotate the point P in a circle spanned by the normalized $\mathbf{H}$ and $\mathbf{V}$ vectors and then recenter it to point M:

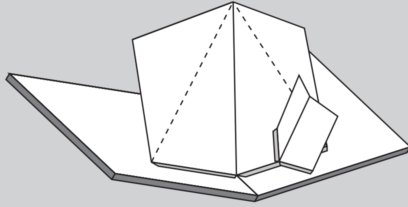
$$Q = M + r\hat{\mathbf{H}} \cos \theta + r\hat{\mathbf{V}} \sin \theta$$

Figure 13 shows this circle and the position of the new point Q.

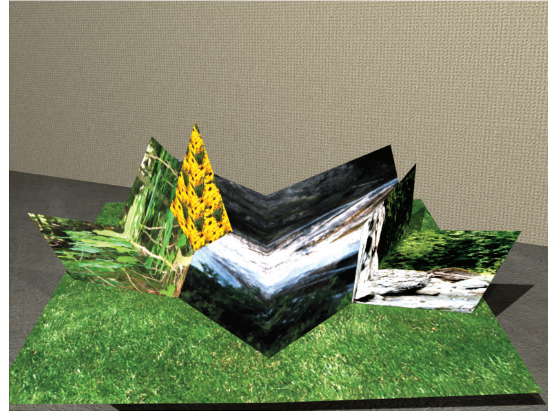
Generations

The method presented in the last section for determining a pop-up's points with respect to its base risers makes it easy to provide designers with a technique known as *generations*. Basically that just means cascading a series of mechanisms one after the other. In terms of single-slit and V-fold mechanisms, it usually means placing a new mechanism on the card so that it isn't powered by the central card fold but rather by the induced creases of an earlier riser. So as the riser pops up, it's like a little card that is opening, and the fold of that little card drives another pop-up. Figure 14 shows the basic idea schematically.

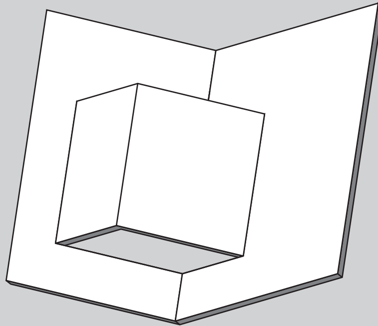
The design assistant I described in the last section handles this mechanism naturally, because it just involves placing the points of one riser with respect to an earlier one. Figure 15 shows an example of a card with



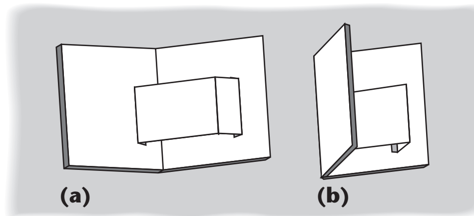
14 **The basic idea of creating generations by using one V-fold as part of the base for another.**



15 **A pop-up card based on generations of V-folds. The yellow flowers are on a third-generation riser.**



16 **A schematic for a double-slit mechanism.**



17 **A schematic for the strap mechanism.**

Original card © 2002 Andrew Glassner

18 **A greeting card that uses V-folds and a strap. The icebergs are V-folds. The ship is also a V-fold, based on one edge of a strap. (You can see the strap to the left of the ship since the card is still slightly open.)**

multiple-generation V-folds. Note that it's hard to make more than a few generations of V-folds on top of each other. This is because each V-fold sits at an angle a little closer to the viewer than the one upon which it's based. This means that eventually we'll run out of nice angles for viewing. More importantly, each generation reduces the angle of the crease upon which the V-fold sits. When a V-fold sits on an open card, the two sides are flat. As the sides come together at a smaller angle, as they do on higher generation V-folds, it gets more difficult to both design and construct the card so that it both moves properly and looks good.

Other mechanisms

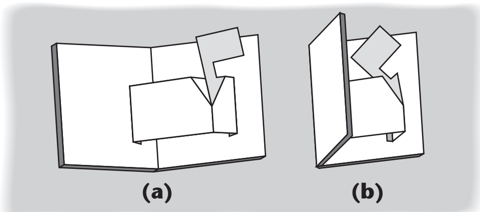
Although the V-fold and the single slit provide the basic geometry for many pop-up designs, I support several other mechanisms in my design assistant. (The books in the "Further Reading" sidebar provide more details on these mechanisms.) I won't describe the detailed geometry of these mechanisms because they're all either variations on what we've already seen, or they're pretty straightforward on their own.

For simplicity, I'll discuss these mechanisms with respect to the main card. Of course, they can all be built with respect to any two risers and built up in a generations style, one upon the other.

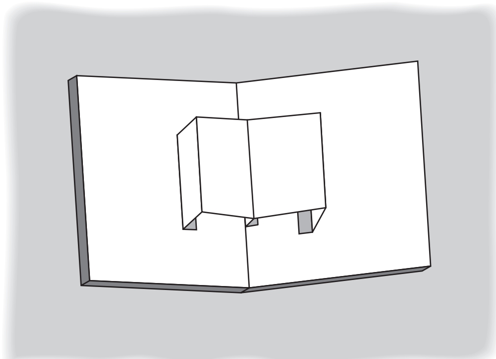
The *double slit*, as Figure 16 shows, is a minor variation on the single-slit design. Although I've shown it with cuts that are straight, perpendicular to the fold, and symmetrical about the fold, none of these need to hold true—the cuts can be of any shape.

Let's next look at the *strap* in Figure 17. The strap lets us do two things at once: displace the central fold to the left or right, and reverse the fold's direction. In Figure 17, I've used the strap to move the fold to the right, and although the card's sides form a valley with respect to the card's central fold, the strap's sides form a mountain. The strap's geometry is easy to implement, because it always forms a parallelogram with the card.

Figure 18 shows a card built on a strap and two V-



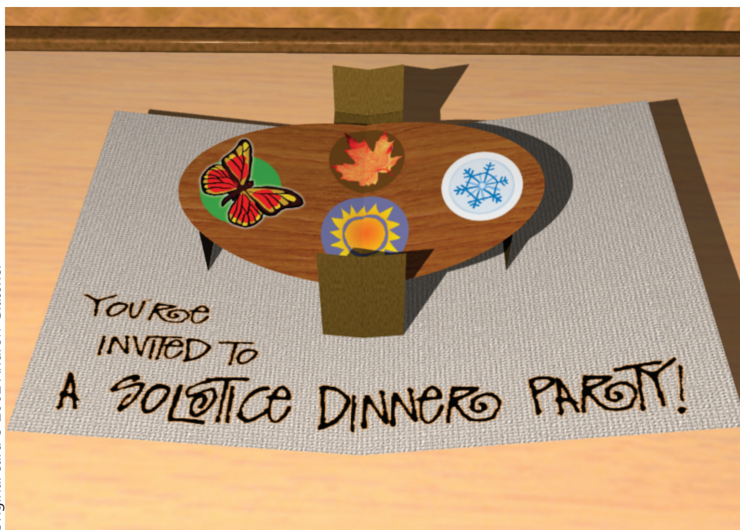
19 **A schematic for a moving-arm mechanism, created by merging a strap with a single slit.**



21 **A schematic for a floating layer mechanism.**

Original card © 2002 Andrew Glassner

20 **A greeting card that uses the moving arm.**



Original card © 2002 Andrew Glassner

22 **An invitation card that uses a floating layer for the table. The chairs are V-folds.**

folds. The two icebergs are standard V-folds that rise off the card's sides. The ship is a V-fold that sits on one end of a strap that straddles the central fold. Thus when the card opens, the ends of the strap flatten out and the ship pops up behind the iceberg. Because the card isn't fully open in this image, you can see the strap to the left of the ship where it hasn't quite flattened out.

You can combine the strap with the single slit to create the *moving arm* or *pivot* mechanism, as Figure 19 shows. The single slit may be a little hard to see; think of the strap's top as the slit. The moving arm is the technique I used in Figure 1 of last issue's column to make the flag pop out of the mailbox. Combining such basic mechanisms is how we achieve some of the most surprising pop-up effects.

Figure 20 shows a New Year's card with a moving arm. The dragon pivots out of the card counterclockwise.

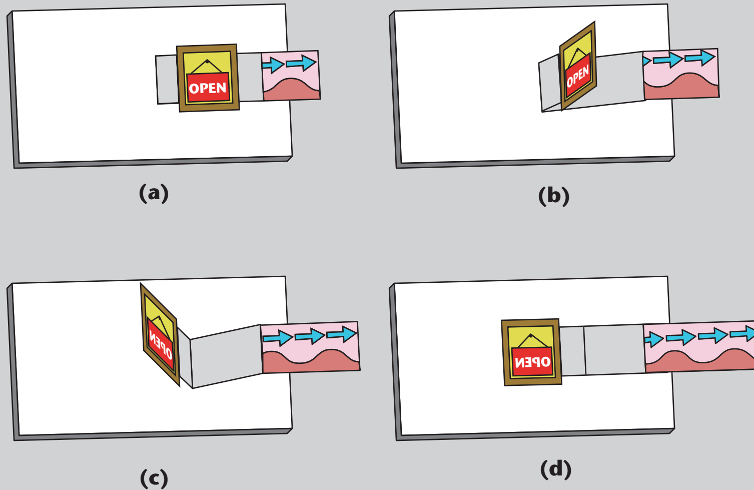
You can create a layer that's parallel to the card but sits above it using the technique called the *floating layer*, as Figure 21 shows. Supports at the two sides hold up the floating layer. The supports are generally the same height and placed an equal distance from the center fold, but they needn't be. For stability, one often includes a support piece in the middle, as Figure 21 shows. By changing the heights of the three supports, you can change the slope of the floating layer's two sides from flat to inclined in either direction. At the extremes, you can shrink the side supports to create a mountain or tent, or shrink the support in the middle to make a valley. The floating layer idea is versatile, and you can use it to make complex figures like boxes and cylinders.

Figure 22 shows a party invitation that uses the floating-layer technique for the dinner table. The two chairs are built out of V-folds.

Pulling tabs

So far we've looked at pop-up mechanisms that work themselves. That is, all the reader has to do is open the book or card and the paper does its thing. There are also a few popular mechanisms that let the reader take a more active role.

One such device, known as the *pull tab* or *pull-up plane*, has various implementations. Figure 23 shows one of the basic versions. In this mechanism, two sheets



23 A schematic for the pull-tab mechanism. Gray parts are sandwiched between the top and bottom layers of the card.

make up the page, sandwiching the mechanism between them. I've drawn these obscured parts in gray.

It's convenient to think of three parts to the pull tab, even though they're all made out of one piece of paper. There's the tab itself, which sticks out from the page through a slot cut in the upper sheet. Then there's the visible flap, which also sticks up through a slot. Finally, there's the bit underneath, which is a single rectangle of paper with a fold in it.

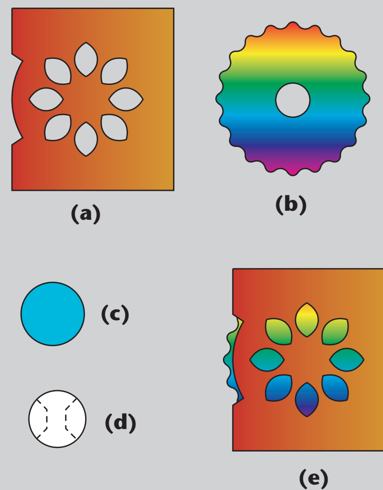
As you pull the tab, the segment underneath tries to straighten out and bulges downward. It tries to pull the visible flap into the space between the paper, but this flap is taller than the slot. Thus, the part underneath causes the flap to pivot around the slot. When we pull out the tab all the way, the flap flips over and lies flush against the page in the other direction, revealing the flap's back and whatever was on the page underneath it.

You can get this effect for free if you use a complete collision-detection or constraint system, as I discussed last time. But the mechanism is so simple that a bit of special-purpose code to flip the flap around the slot can do the job quickly and accurately.

The pull-tab is a nice way to create a card within a card. Really ambitious designers can include V-folds and other mechanisms inside this little minicard, but you must be careful not to get carried away. The risk is that people (both adults and children) often quite understandably think that if pulling the tab opens the mechanism, then pushing the tab should close it. This sometimes works in simple cases, but often the flap under the card buckles and then the mechanism never again works as it should. Sometimes designers reinforce the hidden flap with additional layers of paper or thicker board to prevent this problem.

Spinning the wheel

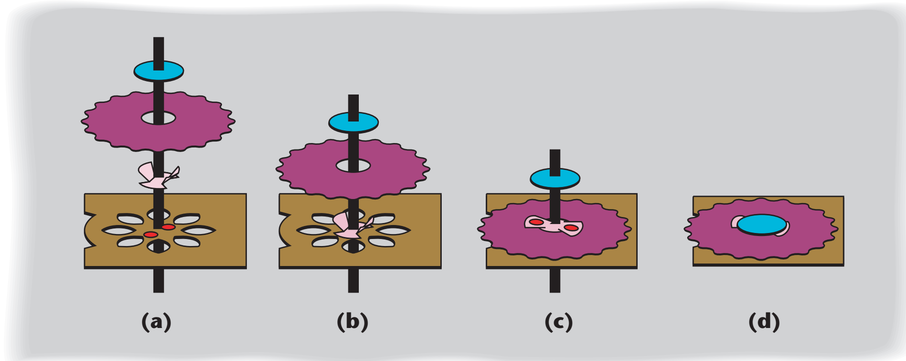
Another popular device is based on the idea of a *wheel*. The wheel has four parts, as Figure 24 shows. In Figure 24a, I show the front of the card. The notch on the left gives the reader access to the wheel's edge, and the holes in the card let what's printed on the wheel itself show through. Obviously, these holes can be of any shape and number as long as the card still holds together. A second card of the same size and shape (except without the holes) is usually made made for the back.



24 The parts for a wheel mechanism. (a) The front card with a notch that lets readers access the wheel's edge, and holes through which they see the wheel itself. (b) The wheel. (c) The nut. (d) The butterfly hub. (e) The completed card as seen from the front.

Figure 24b shows the wheel itself, which has a rippled edge so that the reader can easily spin it. The two small parts in Figures 24c and 24d form the hub and nut, which I'll show in more detail in a moment. The final card (as seen from the front) is in Figure 24e. The wheel shows through the holes, and the reader can spin the wheel from the side.

The hub is easy to make (once you know the secret). To make the wheel's center we need one piece I call the



25 Constructing a wheel. (a) First glue two wings of the butterfly to the back of the front card. (b) Thread the other wings through the wheel. (c) Glue the top of the wings to the nut. (d) The completed sandwich seen from the back. The black line isn't part of the card, but is just to show how the pieces are vertically stacked.

Original card © 2002 Andrew Glassner

26 A birthday card based on a wheel mechanism. The rider sits on an extension affixed to the wheel. (a) The card at one position of the wheel. (b) When the wheel is turned, the spokes turn (they are visible through the holes in the front of the card), and the rider rocks back and forth.

butterfly (Figure 24c) and one nut (Figure 24d). You can see the assembly process in Figure 25. Flip the top sheet over and glue two wings of the butterfly to the back of the front card. The red dots in Figure 25a show where the glue goes to affix the butterfly to the card's back. Then, we fold up the butterfly's wings and pass them through the hole in the wheel's center. Next, straighten out the butterfly flaps and glue the top of each one to the nut that goes on the top of the stack. The red dots in Figure 25c show where to apply glue. Figure 25d shows the final assembly. Notice that nothing gets glued to the wheel itself! That's why it rotates freely. Cut the wings so that they'll fit through the center of the wheel. You'll need to curl up the wings when you thread them, but they'll hold the wheel securely once the nut is in place. When the whole sandwich is complete, you can glue the card's front and back together around their rim. Make sure to leave enough room for the wheel to spin freely.

Figure 26 shows a birthday card based on the wheel. The spokes on the unicycle turn when the wheel spins, and as the wheel turns, the rider rocks back and forth. To make the rider's rock happen I perched the unicyclist on an arm that passes through a slot in the card's front, as Figure 27 schematically shows. Inside the card, I attached a larger disk behind the wheel with the spokes printed on it and then placed another hub near the rim of that disk. Thus as the main wheel goes around, the smaller hub on its edge goes with it. The rider is pulled up and down with the wheel's motion and pivots when the wheel extends to the slot's left or right.

Staying in bounds

When the card is folded flat, we usually don't want any of the pieces to stick out beyond the cover. I check for this condition by setting the folding angle ω to 0 and then calculating the position of every point in the card. I check to make sure that each point lies within the region defined by the outermost card. If any points are outside this region, they'll stick out when the card is closed. I mark them with a bright color so that the designer can fix the problem.

Note that the shape of the outside card doesn't have to be rectangular. That's by far the most common shape for self-contained cards, but there's no reason not to use a card cut into an ellipse or any other shape.

Collisions

One of the hardest problems to solve when manually designing a card is detecting and resolving collisions. When you have several mechanisms moving at the same time, in different places and at different speeds, it's all too easy to end up with the pieces banging into each other. This is bad enough when the card is opening, but it makes it almost impossible to close the card again without damaging it.

I think the best way to handle collisions is to detect them and then let the designer figure out how to resolve them. One can certainly cook up all kinds of automatic schemes that move the points around algorithmically, and that would indeed solve the technical side of the problem, but it may also change the aesthetics. The point here is to make a designer's tool, not find a result that makes the computer happy! I prefer to let the designer know that there's a problem and use a human touch to make things right.

A careful, algorithmically complete job of collision detection would certainly work well. But it looks to me like it would be a complicated program—I'd have to figure out the motion paths for every point, edge, and plane and then check them all against each other. It seems easier to take a rough-and-ready approach inspired by point sampling rendering algorithms.

To search for collisions, I simulate the card's opening by stepping the fold angle ω from 0 to π. At each step, I completely position the card and then look for any edges that pass through any planes. If I find any, I flag them and continue. When I'm done, I highlight any offending edges with a bright color. If the designer clicks on any of those edges, a display shows all the angles where collisions occurred. The designer can then click to set the card to one of those angles, which will show the collision. The designer can then manually adjust the points to repair the collision and can then go on to fix other problems, or run the collision-detection routine again.

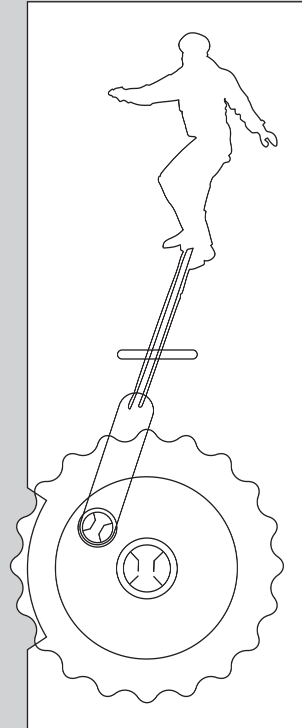
This scheme is fast and has always worked for me. It has the potential to fail when there's an intersection for a brief period of time between two of the checked angles. The easiest way to reduce the risk of a collision falling between the cracks is to simply crank up the number of steps taken by ω. I use 250 steps by default, and I've never been surprised when building any cards. I bet you could get away with 100 steps or even fewer if speed is an issue.

Any collisions that sneak through this test are probably not worth worrying about, because they come and go so quickly the paper's inherent flexibility will probably let it bend enough to avoid actually intersecting. It's possible that some big collisions could sneak through (for example, two corners could just touch and lock up against each other), but I've never seen any failures of this approach so far.

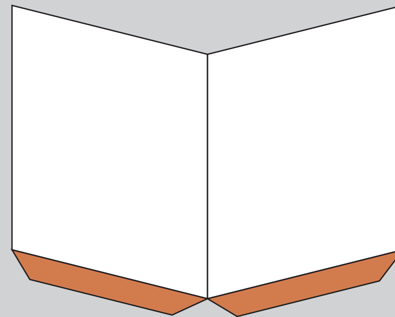
Printing it

As I said in my last column, my reason for making my pop-up design assistant wasn't to create cards for viewing on the computer, but rather to create cards to print, cut out, build, and share. So printing out the designed card is important.

The first issue is making sure that you can easily and properly assemble the card. Gluing down the risers in the right position has always been tricky for me. It's important that each piece is just where it should be so that the card will be flat when folded shut, the pieces will rise and stand up in the correct place and angle when the card is open, and we won't have any collisions along the way.



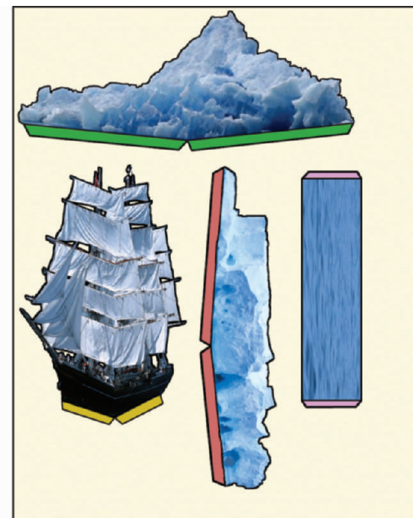
27 A schematic for the cam mechanism of Figure 26. The rider sits on a piece that rotates around a point on the wheel, through the use of another butterfly hub and nut at that point.



28 Creating little glue-down tabs for a V-fold.

Before we figure out how to glue the card together, we have to make sure that there's something to be glued down! So far all my discussions have assumed that risers are infinitely thin and sit on the card, perched there as if by magic. The stiff paper I use for construction is thin enough that I don't have to explicitly model its thickness for most things. But somehow we do have to provide tabs for gluing pieces into place.

Figure 28 shows a typical V-fold riser and the two



29 The final cutout pages for Figure 18. I've turned on the optional heavy black outline for these pieces to make them easier to cut. (a) The card's base. (b) The pieces to be cut. Notice that the strap and one of the icebergs are rotated 90 degrees.

tabs that I automatically generate for gluing it down into place. Note that these glue tabs are little trapezoids, not rectangles; particularly at the center of a V-fold, rectangular tabs would overlap. This would create an ugly bump and also make the card a little harder to close.

To help me assemble cards, I have a switch to turn on the printing of *guidelines* on all risers that have other risers built upon them. This is just a colored line where the riser should be glued down. I then print the tabs with the same color. So anywhere I see a red line, for example, I know that a riser with a red-colored tab should be glued down on that line.

If you build a card, you'll quickly discover that the part of the tab that shows is the back of the page, which can look pretty bad if the color of the page doesn't blend in with the artwork. One solution is to cut a slit where the riser meets the card, and then slip the tab through the slit, gluing it to the underside of the card. This requires a sharp crease to make sure the mechanisms can rise and fall properly. Another approach is to print the base riser's texture on the tab of the riser that sits on it, so that you can glue down the tab and the art is seamless. This requires a color printer that can print on both sides of the page with good registration.

Now that we know how to assemble the card, we need to get the pieces onto paper. Of course, we could be lazy and print one piece per page, but that would be wasteful. What we really want to do is to pack the pieces together into the smallest number of pages.

The best references I could find for this process were in the clothing manufacturing industry, where it's important to conserve materials. Every bit of waste is expensive, so they work hard to lay out the pieces as efficiently as possible, including rotating them and then otherwise shuffling them around to get the densest packing. (The "Further Reading" section identifies a good paper for getting into that literature.)

Those algorithms can be complex. As always, I prefer simple solutions that get me 90 percent of the way there over complex solutions that get me to 100 percent. Achieving that last 10 percent often requires 10 times more work! My simple technique is a greedy algorithm that packs the pieces in one at a time from largest to smallest.

I begin by creating a data structure called a *page*, which contains a list of *rectangles*, representing regions of the page that haven't yet been printed upon. This list begins with a single rectangle that covers the page. I run through the list of pieces as a one-time preprocess and sort them by the size of their bounding rectangles. I start with the largest rectangle and work my way down to the smallest, placing them as I go and creating new pages when necessary.

To place a piece, I look at the first page and its list of available space—that is, the list of rectangles representing blank space on the page. I try to position the piece's bounding rectangle on this page, trying it in both horizontal and vertical orientations. If there's a way to get the current piece's bounding rectangle onto the page, I place it as snugly as I can. I then use simple geometry to remove that rectangle from the page's available rectangle list. If the rectangle won't fit on this page, I repeat the process on the next page. If it doesn't fit on any existing pages, I create a new page. In this way, I tend to make a bunch of pages that start out with just one or two big pieces and a lot of empty space, but then that space gets nibbled away by the smaller pieces as they arrive and are placed.

Figure 29 shows the result of this algorithm for the card design in Figure 18. Note that the algorithm rotated one of the icebergs and the strap 90 degrees to fit the page. Without this rotation, the pieces would have taken up two pages.

This algorithm isn't perfect by any means. There's some waste, and the pieces could surely be packed more tightly. But I usually get pretty good density on the pages, and the algorithm has the benefits of being easy to program and fast to run.

Moving up

There are lots of ways to extend my pop-up card design assistant. One thing I'd really like to try is applying computer-vision segmentation techniques to pho-

tographs, automatically dividing them into several planes based on distance. For example, we could have a foreground, several middle grounds, and a background. The system could then take these segmented regions of the photo, place them on a series of V-folds spaced roughly like the objects in the scene, and create a 3D version of a photograph (sort of like Figure 18). If a couple of versions of the photo are available, you could use them to fill in the holes in the background layers where the foreground information was cut. Alternatively, you could fill in the holes using texture synthesis methods.

Another fun project would be to create pop-ups automatically from 3D scene descriptions. Pop-up designers have many special-purpose tools up their sleeves, from self-assembling tables and cubes to lattices and stacks. It would be very cool to take a 3D model—say of a car, teapot, or flamingo—and automatically generate a pop-up version for 3D, interactive, offline viewing. Even scientific visualizations could be done this way. The advantage is that we can make the pieces available on any image medium, from a printed journal to the Web. Then someone just prints them out, assembles them, and enjoys the view.

It would be interesting to automatically incorporate forced-perspective illusions onto the planes to give the card an even richer illusion of depth.

I'd also like to add support for nonrigid constructions, such as curved folds and pressure forces. For example, in Figure 1b of last issue's column, I showed a moving card I designed where I made three V-folds with holes cut through them. As the card opened and the V-folds rose, they pulled up a floating layer that passed through them. There's no direct mechanism here that causes this to happen; it's just that there's nowhere else for the floating layer to go, so it must rise up with the V-folds themselves. There are many such possibilities, and it would be nice to provide them to the designer.

I'd like to flesh out my program with some of the other special-purpose pop-up mechanisms, such as pulleys, Venetian blinds, cylinders and cubes, and other uncommon but useful constructions.

Finally, I'd like to bundle up my design system into a plug-in for a commercial modeling package. That way people could use systems they're already familiar with and use all the tools they already know for modeling, texturing, lighting, rendering, and so on.

I think that these are all rich research topics full of juicy pieces of geometry, optimization, segmentation, and interface esign. I encourage you to play around with this exciting new form of graphics output technology. ■

Acknowledgments

Thanks to Sally Rosenthal for pop-up advice and encouragement, and Bryan Guenter for pointing me to several useful references. I modeled all the computer-rendered pop-ups in this column with my assistant and lit and rendered them with Discreet's 3DS Max 4.

Readers may contact Andrew Glassner by email at andrew_glassner@yahoo.com.

IEEE Computer Graphics AND APPLICATIONS

Call for Papers

January/February 2003 issue on

Web Graphics

Guest Editors:
Rynson W.H. Lau and Tosiyasu L. Kunii

Submissions due: 1 May 2002

With the popularity of the Internet, we're seeing a migration from traditional applications to those that run in the Web environment and a growing demand for more powerful Web-based applications. Fused by the increasing availability and dramatic reduction in the cost of 3D graphics accelerators, a new direction of research, called Web Graphics, is emerging. It includes developing graphics applications and tools that support these applications in Web environments. This special issue will explore recent research results in this area.

***IEEE CG&A* covers a wide range of topics in computer graphics, bridging the gap between theory and practice.**

For submission instructions visit

http://computer.org/cga