Globs: A Primitive Shape for Graceful Blends Between Circles

Andrew Glassner
The Imaginary Institute

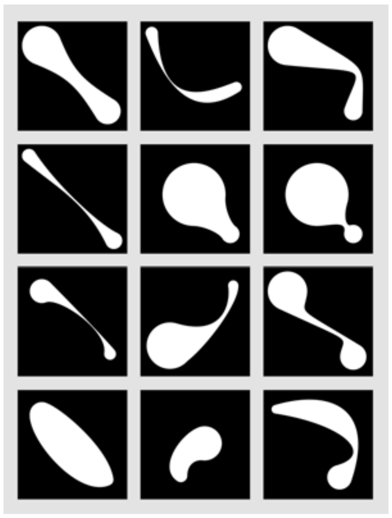

Figure 1. A gallery of globs.

Abstract

Geometric primitives are a staple of illustration systems. They provide artists with shapes that are easy to understand and create, yet they can be easily combined and modified into more complex structures. Here we introduce a new 2D geometric primitive called a *glob*. It is a smooth curved shape that blends two circles of any radii. Globs are controlled with a few parameters which have immediate geometric interpretations.

1. Introduction

Almost every illustration system offers a mix of free-form and predefined shapes that can be used as starting points for more complicated shapes. These shapes can be broken down into three categories: geometrically primitive shapes (e.g., circles and polygons), compound shapes (e.g., rectangles with circles on the ends), and procedural shapes (e.g., swept shapes and algorithmically-generated blends).

Part of the design of an illustration system involves choosing the right collection of these primitives. A list that's too small can be constraining, while a list that's too big can be unwieldy and bewildering.

In this paper, we take our inspiration for a new 2D shape from a popular class of 3D primitives which combine two spheres with a connector.

One such sphere-and-connector shape is the *cone-sphere* introduced by Max [1990]. This primitive smoothly joins two non-overlapping spheres with a right circular cone. Another sphere-and-connector shape emerges from an implicit surface technique popularized by Blinn [1982]. This approach models each sphere as a spherically symmetric Gaussian density field called a *blob* and then extracts an isosurface from the combined field of all the blobs. This naturally produces a smooth join between each pair of spheres that are close enough together.

Both of these primitives have a lot going for them: The cone-spheres require no parameters or controls, and blobs produce lovely stretching and clumping effects as the blob centers move with respect to one another. They are both easily adapted to a 2D drawing system.

These primitives each have drawbacks, though. Because cone-spheres have no shape parameters, they offer no variation or control, and the silhouette is always a straight line joining two circles. Blobs have more parameters, but these offer little control over the fundamental nature of the shapes they produce. For example, as two blobs pull apart, the neck between them gets thinner and thinner until the two shapes disconnect, but one cannot control the length or shape of this neck. To compensate, some modeling systems allow the designer to use other shapes to generate density fields [Wyvill et al. 1999]. This approach is very flexible and offers an exciting breadth of control, but that very flexibility introduces a significant increase in complexity and effort from the designer.

We have designed a 2D family of shapes that generalizes some of the best features of cone-spheres and blobby spheres, offering a wide range of shapes from a relatively small set of parameters. Some examples are shown in Figure 2. Our shapes provide an explicit curved neck between any two circles. The circles may overlap, and the neck can bulge inward or outward and move flexibly in the space between the circles. We can recreate classic cone-sphere and blobby shapes, but we can also easily create new kinds of shapes that would otherwise be difficult to produce, such as necks that start abruptly at one end but grow gracefully at the other, and asymmetric, bulging necks.

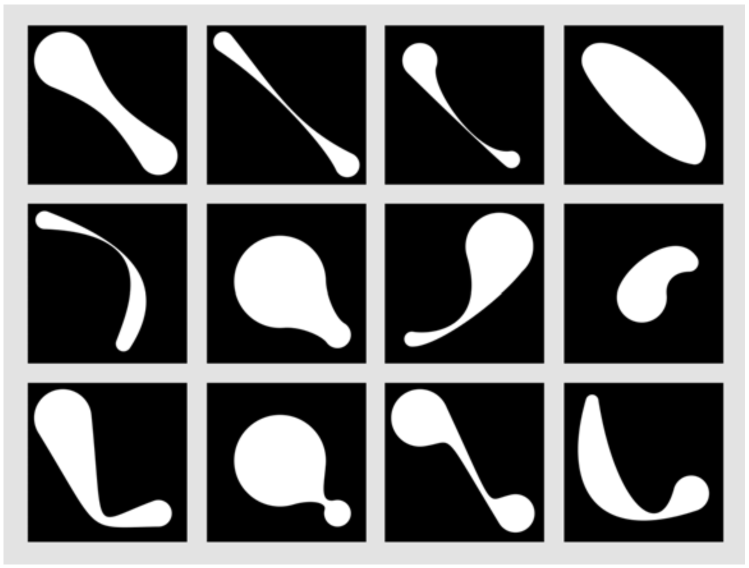

Figure 2. A gallery of globs.

2. The Geometry

The heart of the idea is that we build a pair of Bézier curves between the two circles. We construct the curves so that they have at least first-derivative continuity with both circles. The designer can adjust many visual aspects of this connective neck.

Figure 3 shows the basic idea. We call the two circles $C0$ and $C1$, with respective centers C_0 and C_1, and radii r_0 and r_1. It is tempting to simplify the geometry by placing C_0 at the origin and C_1 directly to its right along the positive X-axis, and indeed one can do this with no loss of generality. As a practical matter, though, we have found that it's more convenient to solve the general case using coordinate-free vector techniques. This lets us produce results that can be immediately translated into code, without the need to write additional steps for scaling, rotation, and translation after the construction is done.

The first step is to introduce a local coordinate system. We create the normalized vector $\widehat{\mathbf{U}}$ on the axis from C_0 to C_1:

$$\mathbf{U} = C_1 - C_0,$$
$$\widehat{\mathbf{U}} = \frac{\mathbf{U}}{|\mathbf{U}|}.$$

We can then find the perpendicular $\widehat{\mathbf{V}}$ by requiring that it satisfies

$$\widehat{\mathbf{V}} \cdot \widehat{\mathbf{U}} = 0.$$

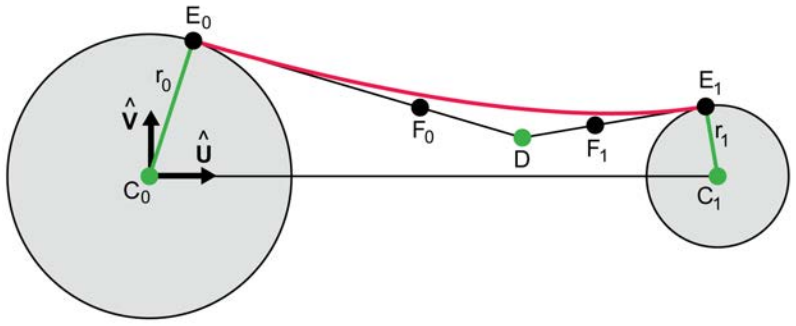

Figure 3. The geometry of a glob. C_0 and C_1 are the centers of circles with radii r_0 and r_1. To build a coordinate system, we find vector $\widehat{\mathbf{U}}$ by normalizing the vector from C_0 to C_1, and then form its perpendicular $\widehat{\mathbf{V}}$. The designer specifies the circle centers and the radii, along with the point D. The system computes points E_0 and E_1 which are the points of contact of a line through D and tangent to each circle. The points F_0 and F_1 are placed along the lines (E_0, D) and (E_1, D). From these points, we draw the Bézier curve (E_0, F_0, F_1, E_1).

This is particularly easy in 2D if we allow ourselves to momentarily refer to coordinate values:

$$\widehat{\mathbf{V}} = (\widehat{U}_y, -\widehat{U}_x).$$

The narrowest (or widest) part of the neck is given by a point D that the designer can place anywhere outside of the circles. To create our Bézier curve, we draw lines from D to each circle so that they are tangent to the circle. In the figure, these points are labeled E_0 and E_1. They serve as the endpoints for the Bézier curve.

To build a cubic Bézier curve we need two control points. We locate these on the lines (E_0, D) and (E_1, D); they are marked F_0 and F_1 in the figure. The locations of F_0 and F_1 are found using two designer-specified scalars that tell us where they should be placed along their respective lines.

The curve we want is then given by (E_0, F_0, F_1, E_1). By construction, this has first-derivative continuity with the circles at both ends.

Note that D can be literally anywhere that is not within either of the two circles. It can be below the line (C_0, C_1) or far outside the convex hull of the circles.

It is fun (and instructive) to find the E and F points using plane geometry, creating a web of right triangles and solving for their lengths and angles. However, a coordinate-free vector approach is much simpler and robust.

We start with finding point E_0. Figure 4 shows the geometry for this step. We can think of Figure 4 as a subroutine: given a circle with center A and radius r, and a second point B that is outside the circle, find a point P_0 on the circle such that the line (P_0, B) is tangent to the circle. As the figure shows, there are two points that

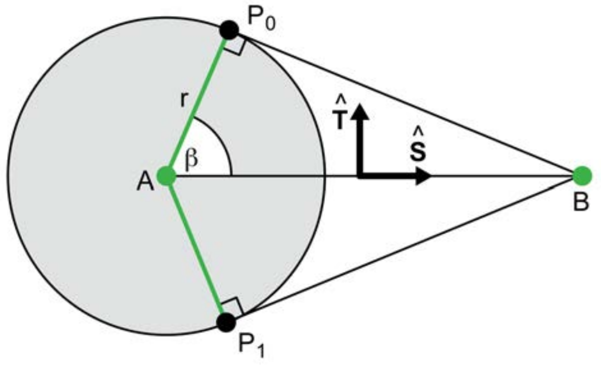

Figure 4. We are given the circle defined by center A and radius r, along with point B. We seek the points P_0 and P_1 that are on lines tangent to the circle and through point B. We find β and use that to add scaled versions of the coordinate system $(\widehat{\mathbf{S}}, \widehat{\mathbf{T}})$ to point A.

fulfill that condition. We indicate which one we want using one more argument to our routine (discussed in a moment). When we use this diagram to find point P_0, then points P_0, A, and B in Figure 4 correspond to E_0, C_0, and D in Figure 3.

We begin by constructing a frame of perpendicular unit vectors $\widehat{\mathbf{S}}$ and $\widehat{\mathbf{T}}$, where $\widehat{\mathbf{S}}$ points from A to B. To find P_0, we only need to find angle β. We know one leg and the hypotenuse of right triangle AP_0B. The length of the remaining leg is provided by Pythagoras:

$$|P_0B| = \sqrt{|AB|^2 - r^2}.$$

Now we can find β using arc-tangent:

$$\beta = \tan^{-1}(|P_0B| \,/\, r).$$

From this, we can create the two points P_0 and P_1, one on each side of $\widehat{\mathbf{S}}$. The expressions are identical except that we add a scaled version of $\widehat{\mathbf{T}}$ when forming P_0 and subtract it for P_1:

$$P_0 = A + (r\cos(\beta)\,\widehat{\mathbf{S}}) + (r\sin(\beta)\,\widehat{\mathbf{T}}),$$
$$P_1 = A + (r\cos(\beta)\,\widehat{\mathbf{S}}) - (r\sin(\beta)\,\widehat{\mathbf{T}}).$$

Essentially we're placing the radius vector at A, pointing at B, then rotating it by β, and finding the components of the new vector in the $\widehat{\mathbf{S}}$ and $\widehat{\mathbf{T}}$ directions.

Both P_0 and P_1 satisfy our initial criteria. We mentioned above that we have one additional argument that lets us pick the one we want. This argument specifies whether we want the one that is on our left or right as we stand at A and look towards B. To find which point lies on which side, we find two cross products and take the sign of their z-component:

$$\begin{aligned}
\mathbf{K_0} &= (P_0 - A) \times \widehat{\mathbf{S}}, \\
\mathbf{K_1} &= (P_1 - A) \times \widehat{\mathbf{S}}, \\
s_0 &= \text{sgn}(\mathbf{K_0}_z), \\
s_1 &= \text{sgn}(\mathbf{K_1}_z).
\end{aligned}$$

By construction, one of these values will always be positive (on our left) and the other will always be negative (on our right). So, if we want the left-hand point, we return P_0 if $s_0 > 0$; otherwise, we return P_1. For the right-hand point, we return P_0 if $s_0 < 0$, else P_1.

When applied to both circles, this method provides us with points E_0 and E_1 that are the endpoints of our Bézier curve. The next step is to find the internal control points F_0 and F_1.

To find F_0, the designer specifies a scalar value a. This gives us a point on the line from E_0 (when $a = 0$) to D (when $a = 1$). This is something of a *tightness* or *tension* parameter, since it pulls the curve more or less forcefully towards the point D. We find F_1 in the same way, using tension parameter b:

$$\begin{aligned}
F_0 &= E_0 + a(D - E_0), \\
F_1 &= E_1 + b(D - E_1).
\end{aligned}$$

We can now draw the cubic Bézier curve (E_0, F_0, F_1, E_1). By construction, this has first-derivative continuity with the circles at both ends.

Since point D can be anywhere outside of the two circles, we can produce a wide range of squeezes and bulges. Because the tightness parameters a and b are independent, we can individually adjust how quickly the neck forms at each end, as shown in Figure 5.

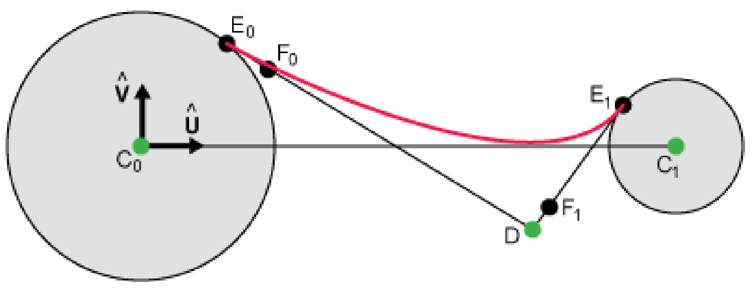

Figure 5. Parameter a controls the location of point F_0 on the line (E_0, D), and simliarly parameter b controls the location of point F_1 on the line (E_1, D). These parameters are independent, allowing us to create neck shapes that blend asymmetrically into the two circles.

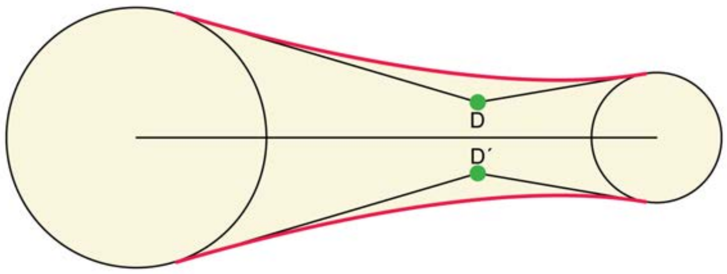

Figure 6. To create a symmetric glob, find the point D' that is the mirror reflection of D around the line (C_0, C_1) and create a new curve as before.

We can make a symmetric glob by drawing another Bézier curve that mirrors the first about the line (C_0, C_1). This curve has the same basic form as the one we just looked at, so we'll use the same labels for the points but attach a prime mark to distinguish the lower points from the upper. We can find a new point D' that is the mirror image of D by breaking D into its projections along $\widehat{\mathbf{U}}$ and $\widehat{\mathbf{V}}$ and then recombining them, negating the component along $\widehat{\mathbf{V}}$:

$$\begin{aligned} D_u &= (D - C_0) \cdot \widehat{\mathbf{U}}, \\ D_v &= (D - C_0) \cdot \widehat{\mathbf{V}}, \\ D' &= C_0 + D_u\widehat{\mathbf{U}} - D_v\widehat{\mathbf{V}}. \end{aligned}$$

A typical result is shown in Figure 6.

We can construct much more interesting globs by creating the lower Bézier curve independently of the upper one, as shown in Figure 7. Thus, the user can place the

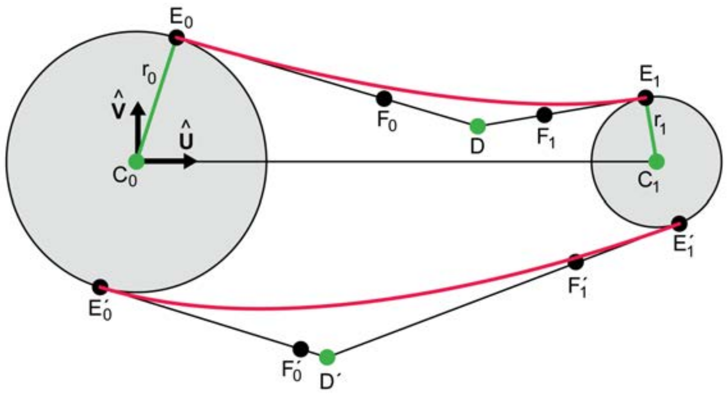

Figure 7. Adding a second curve. The designer only needs to supply a new point D' and new tension parameters, a' and b'.

point D' anywhere outside the two circles, and from that we will find points E'_0 and E'_1. Using tension parameters a' and b', we find control points F'_0 and F'_1, giving us a new Bézier curve. These two curves can interact in many ways, producing a generous variety of shapes, as shown in Figure 2.

3. Discussion

3.1. Cusps and Loops

The construction variables discussed above may be chosen so that the resulting Bézier curve forms a cusp, or even a loop. It is possible to detect both of these cases [Stone and DeRose 1989], and one could then place limits on some or all of the design parameters to prevent such shapes from forming. We prefer to leave the values unconstrained, so designers are free to explore and use the entire design space.

3.2. Unexpected Corners

Our geometry for finding the tangent points is robust and fast, but it can produce a sharp corner where one might expect a smooth blend. This is because, although the curve will be tangent to the circle at the point of type E, depending on the tension parameters and the location of D (or D'), the curve may be shaped so that it does not wrap around the circle, but instead passes through it. The result is what appears to be a corner where the neck emerges from the circle, as shown in Figure 8.

One could detect this condition (perhaps by moving very slightly along the curve and testing to see if that point is inside the circle), and then enforce constraints to avoid it. But just as with cusps and loops, we don't try to prevent designers from creating such globs, because they can be useful.

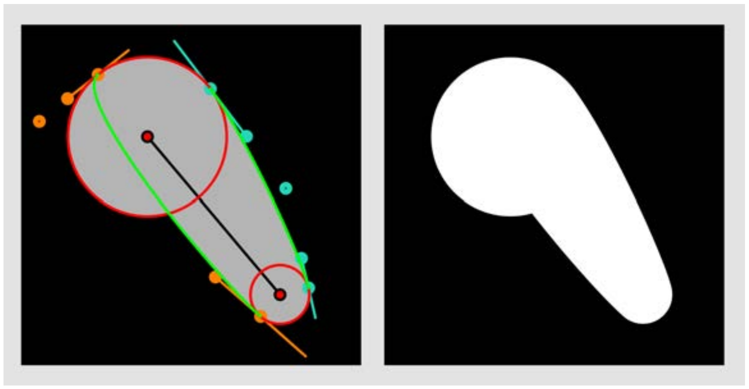

Figure 8. A sharp corner produced in our interactive tool. Left: Here we've drawn the two Bézier curves in green. Right: The glob without the interactive controls.

3.3. Tangents, Normals, Offset Curves, and Motion Paths

A nice feature of globs is that they have easy-to-compute tangent and normal vectors. For points on the circular ends, finding tangents and normals is trivial: the normal is merely the unit vector from the center of the circle to the point on the perimeter, and the tangent is its perpendicular. It's also easy to find these vectors along the Bézier curves. The tangent is given by the standard equation for the tangent to a Bézier curve at a given point, and the normal is its perpendicular [Shirley et al.].

We can use these normals to approximate an offset curve. It is well known that one cannot explicitly construct a Bézier curve that is the offset of another [Kamermans 2015], but there are many approximate algorithms. One of the simplest is to evaluate the curve at many points and find the normal at each point. An approximate offset curve can be producing simply by scaling the normals and joining them together as a spline.

The inner and outer glob offset curves in Figure 9 were constructed in this way. We sampled many points, found the normals, scaled them, and connected the endpoints together in a single closed Catmull-Rom curve.

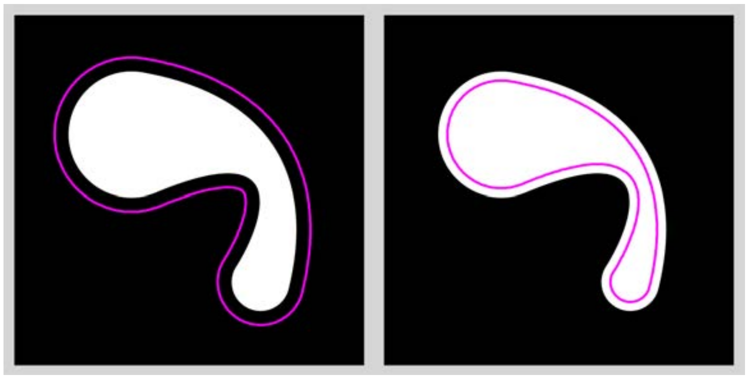

Figure 9. An outer and inner offset curve.

Figure 10 shows the same glob but with the normal vectors included. In this image, we have reduced the number of points evaluated along the curve from the number used in Figure 9 so the normal vectors are easier to see. The inner curve is constructed in the same way, only the normals are scaled by a negative amount.

One note of caution when working with this approach, though, is that one must remember that Bézier curves are not uniformly parameterized. That is, taking equal steps of the input parameter (say t going from 0 to 1 in steps of $.1$) will usually *not* result in equally-spaced points along the curve. This phenomenon is visible in the more highly curved portion of the shape in Figure 10.

When the sampling density of points along the curve is low, this non-uniform spacing has the risk of creating artifacts in the resulting offset curve. Probably the

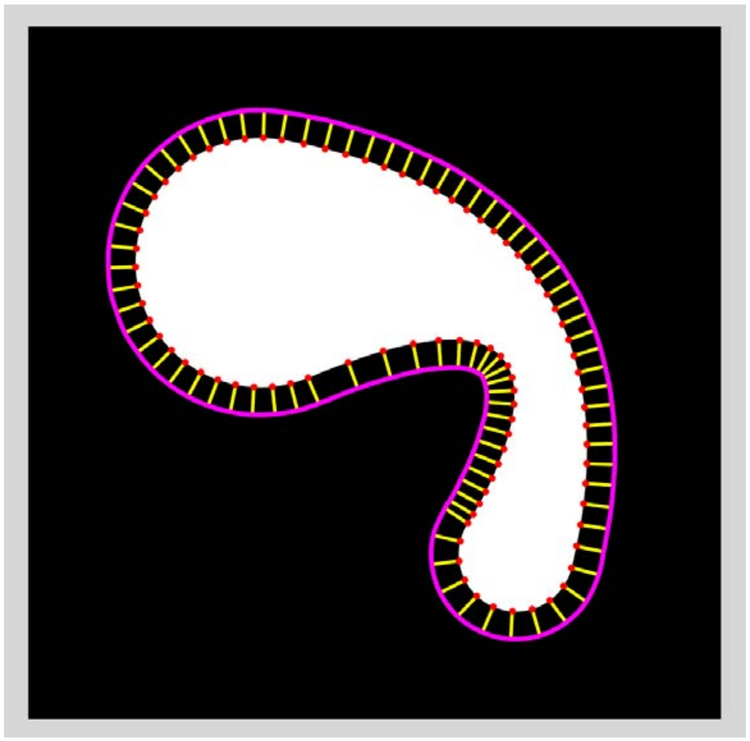

Figure 10. Creating the outer offset curve in Figure 9. The glob is evaluated at many points (red), the normals are found (yellow) and scaled, then joined together as a Catmull-Rom curve to form the offset curve.

easiest way to resolve the problem is to simply use more points. Another option is to explicitly adjust the evaluation parameter for the Bézier curve so that it corresponds to arclength along the curve. This can be done efficiently by pre-computing the approximate arclength at many points along the curve and then looking up each input value in that table, returning a replacement evaluation parameter that produces a point that is at the desired distance along the curve. That is, we hand the table a value of t in the range $[0, 1]$, and we get back a value t' in the same range such that when we evaluate the curve using t', the point we get back is t percent of the way along the curve.

Depending on the particular curve shape and offset amount, offset Bézier curves can exhibit loops, swallowtails, and other non-smooth features. In particular, tight turns and large offsets reliably lead to cusps. The upper-left of Figure 11 shows one such example. Notice that the inner offset for this same glob, shown in the upper-right, does not share the swallowtail in the outer offset. These phenomena are a natural property of approximated offset Bézier curves and are not specific to globs. In general, as with the cusps and loops mentioned above, it's a judgment call by the designer to determine whether to accept these features, or to eliminate them by

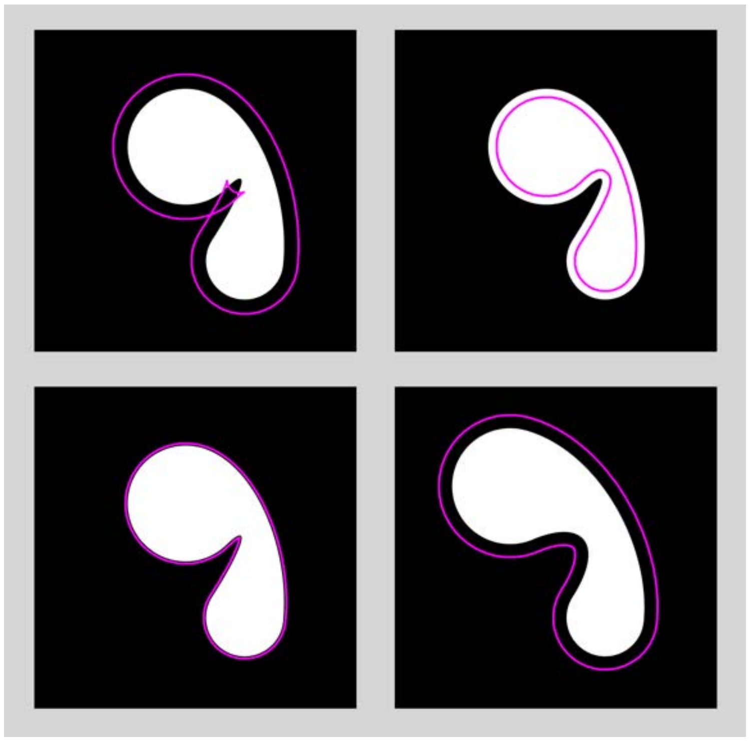

Figure 11. Upper-left: A glob's outer offset with a swallowtail cusp. Upper-right: The inner offset has no cusp. Lower-left: Reducing the distance of the offset removes the cusp. Lower-right: Modifying the shape to remove the cusp.

changing the distance of the offset (lower-left of Figure 11) or changing the shape (lower-right of Figure 11).

Because it's so straightforward to find points along the perimeter of the glob, the shape may also be conveniently used as a motion path. Using the arclength-based point spacing discussed above will give uniform speed along the glob's perimeter.

3.4. Smoothing Multiple Globs

An interesting issue arises when two globs share a common circle at one end. In some cases, they can produce a sharp corner where they overlap, as shown in the left of Figure 12. This is no different than when any two primitives overlap: there is no natural or automatic blending. And in many situations this contrast between smooth curves and sharp corners can be attractive. But because globs normally have a very smooth nature, there are times when we would like them to blend together smoothly as well.

One can design variations on the glob geometry to explicitly produce smooth surfaces when two globs overlap at a common endpoint. That would handle the case of two globs, but then one would need to design new special case geometry for three

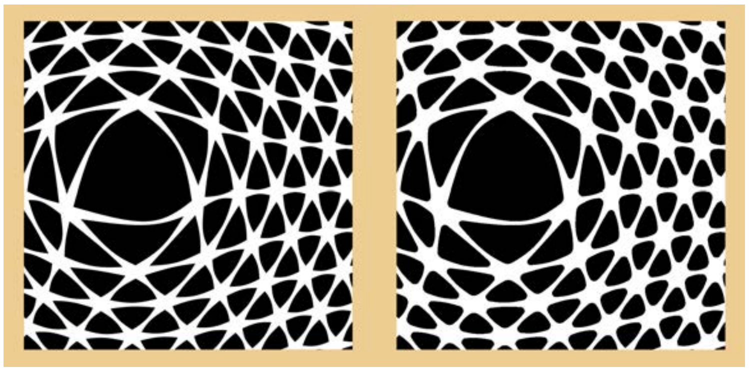

Figure 12. Left: A deformed hexagonal net of globs, showing corners where they happen to overlap. Right: Smoothing the result by thresholding an implicit surface derived from the globs. All the sharp corners are now nicely rounded.

globs, another special case for four, and so on. We haven't found an explicit geometric construction that will always provide smooth joins for any number of shared globs.

Happily, we can solve this problem by using implicit surfaces, like those used by traditional Gaussian blobs [Gourmel et al. 2013]. In 2D, treat each white pixel on the left of Figure 12 as the center of a small radially symmetric blob. For each pixel outside of a glob, sum the field contributions from all in-glob pixels (in practice, only consider pixels that are within the blob's radius, since the contribution from all other pixels will be 0). Normalize the field and then threshold it, creating a smooth isosurface as shown on the right in Figure 12. In this figure, there is some thickening of the shapes due to the spread of the blob fields. To compensate, one can draw slightly thinner shapes in the first place, or use a higher density threshold when extracting the isosurface. This technique can be thought of as using globs as the primitives for building an implicit surface [Bloomenthal 1997]. Indeed, we think the match is a natural one. Globs offer us the ability to control the shape of the blend between spherical functions when we want that control, and an isosurface naturally creates other blends when we don't need to explicitly shape them.

3.5. Specifying Globs

We have built a small stand-alone design program for making globs. Using this program, the designer can specify all the glob parameters interactively.

As shown in Figure 13, the interface closely matches the design structure in Figure 7. We draw small disks to represent each construction point. The points associated with the unprimed points are in teal, while the primed points are in orange. Circle centers are in red.

The points D and D' are highlighted with a black dot in their center. These may be dragged anywhere in the window (though as we'll discuss below, dragging

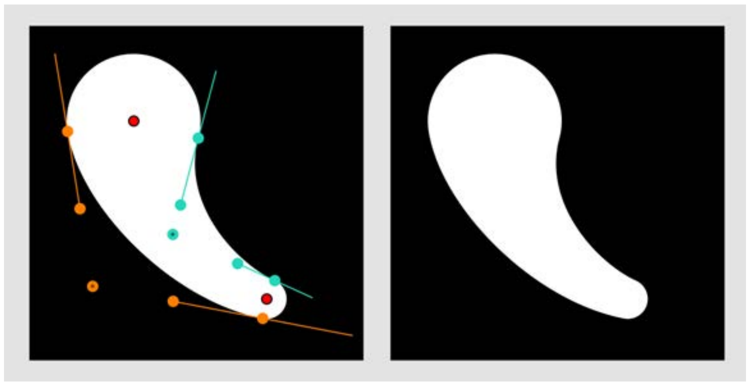

Figure 13. Left: Our design interface in action. The red dots are the centers of the circles. One set of curve points are in teal, the other points are in orange. Points D and D' have a darker center. Right: This glob without the interface elements.

them into the circles will produce non-smooth results). The E family of points are constrained to move on the circles themselves, but pulling them towards or away from their corresponding circle center allows one to adjust the radius of that circle. Finally, the F family of points may be dragged anywhere on the line joining their corresponding D and E points, including regions outside the interval defined by those points. The glob is recalculated and redrawn every time a point is moved. Because the calculation is so fast, the design is fluid and interactive in real time, even when running inside a browser [Glassner 2015b].

If the user adjusts a circle's radius by moving one of its corresponding E points towards or away from the circle center, that point will also move slightly around the circumference of the circle, in order to maintain tangency. In practice, we have found that this effect causes no confusion or surprise, since the point follows the designer's intended location as best it can given the constraints.

To construct the glob, the system infers the parameters a, b, a' and b'. For example, a is given by $a = |F_0 - E_0|/|D - E_0|$, where we use signed lengths, since point F_0 may be dragged outside the interval (E_0, D).

Pressing a key allows the designer to turn these interfaces elements on and off, so the glob can be viewed without distraction.

3.6. API

We have implemented globs in our open-source library [Glassner 2015a] for the Processing system. We provide two slightly different constructors. The first is based directly on Figure 7: the user provides the circle centers and radii, the locations of points D and D', and the four control parameters a, b, a' and b'.

Requiring the explicit locations of D and D' can be inconvenient when the circle centers are moving over time, but we want to preserve the overall shape of the glob. To

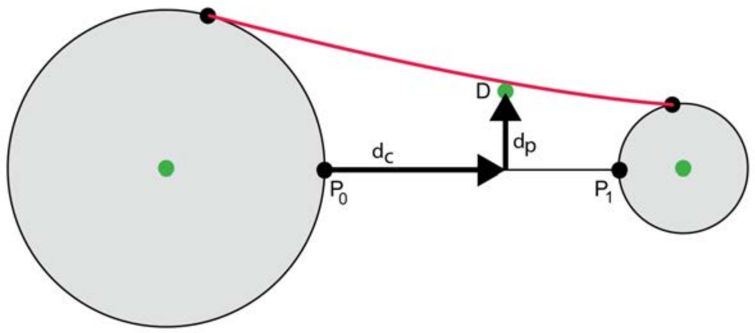

Figure 14. We locate the point D by providing d_c and d_p. These are both percentages of the length of the line between P_0 and P_1.

make this easy, a second API replaces the explicit location of D with a pair of scalars that describe the location of D in terms related to its components in the $(\widehat{\mathbf{U}}, \widehat{\mathbf{V}})$ system in Figure 7.

Let's focus on point D. Because D may not be placed inside either of the circles, the $\widehat{\mathbf{U}}$-coordinate is expressed as a percentage of the length of the line segment formed by joining the circle centers, and then selecting only the region between them. This line is marked (P_0, P_1) in Figure 14. As d_c runs from 0 to 1, we locate a point along the line segment from P_0 to P_1. The perpendicular displacement of the point is given

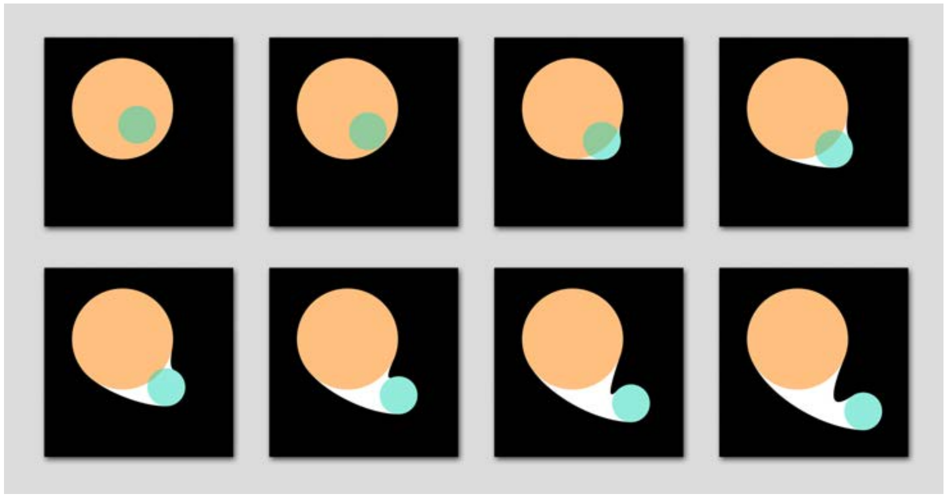

Figure 15. A few frames from a glob animation, where we start with one end circle completely within the other. We can start drawing the neck as soon as at least some of each circle's perimeter is visible.

by parameter d_c, which is again a percentage of the distance $|P_0, P_1|$. Positive values of d_p move the point D away from the line (P_0, P_1) towards the curve's control points, as shown in the figure. Negative values push it in the opposite direction. The point D' is specified in the same way, with its own two scalars, d'_c and d'_p.

As with all our other design parameters, we place no restrictions on these controls, so designers are free to use any values they like in order to fully explore the space of shapes the algorithm can create.

Globs are easily animated. A particularly interesting case is when the two circles overlap. The construction method works fine in this situation, only requiring (as usual) that we keep the points D and D' outside of both circles. Thus, we can start with one circle completely within the other, and let it emerge, as shown in Figure 15. We can draw the neck as soon as at least some of both circles becomes visible.

4. Examples

Globs can create curved, organic shapes. They can also be chained, with one glob sharing an endpoint with another, leading to articulated multi-joint models that can exhibit subtly changing shapes over time without requiring external mechanisms like spatial deformations.

For example, Figure 16 shows a bug with multiple legs. Each leg is defined by a simple data structure that holds each joint radius and position. The legs can be positioned with direct or inverse kinematics, just like any chain structure. And the shapes of the globs can change in response to their motion over time, simulating a dynamics effect.

Figure 17 shows frames from three different animations that use globs to create moving, abstract designs. Over time the globs move and change shape in complex and subtle ways.

5. Related Work

Globs have roots in common with other shapes beyond cone-spheres [Max 1990] and blobs [Blinn 1982]. The approach of Ma et al. [Ma et al. 2012] offers an analytic blending function for cone-spheres, with special-case geometry for creating smooth joins where two cone-spheres share a common end. Similarly, Bastl et al. [Bastl et al. 2015] offers a system for combining many cone-spheres together, but their blends are limited to cones that produce strictly convex shapes. Finally, one can consider globs to be special cases of a general swept sphere surface, as described by van Wijk [van Wijk 1985]. A swept sphere requires a path and a radius profile. We have found that globs offer a simpler alternative to the sometimes challenging design problem of crafting these two curves to achieve a desired shape, particularly when it comes to getting a smooth transition from the end spheres to a curved neck. We feel that a large

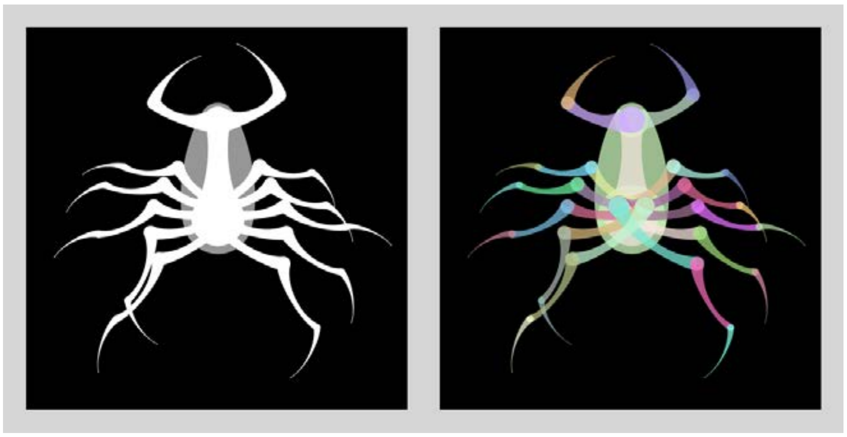

Figure 16. Globs can form organic shapes, particularly when they're chained together by sharing common endpoints. Left: A bug made completely out of globs. Right: The individual globs.

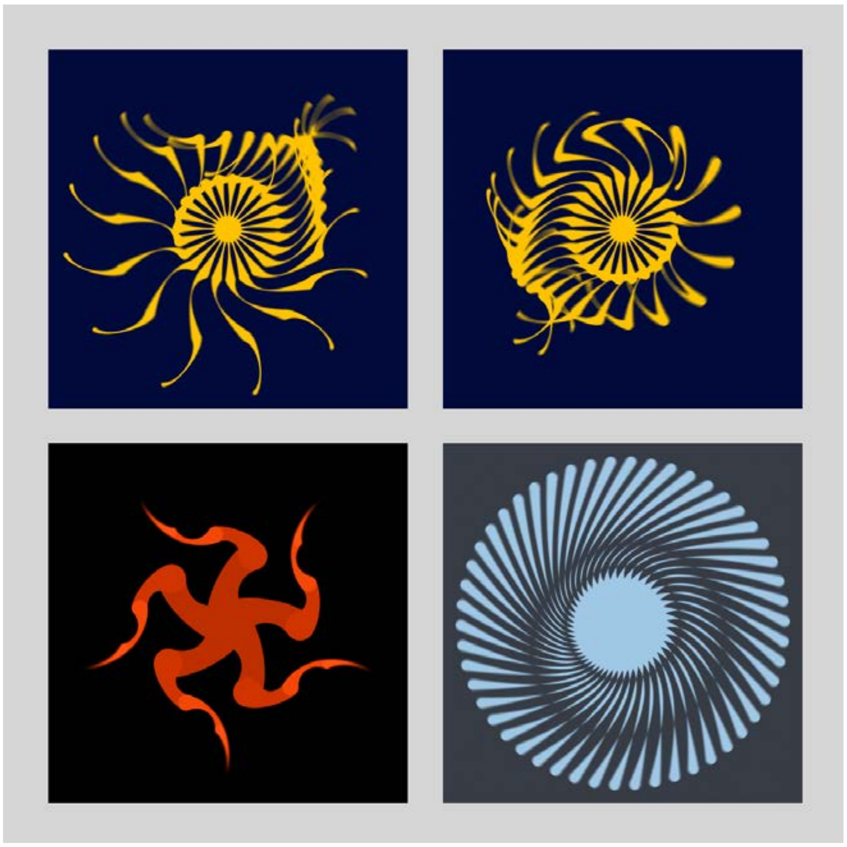

Figure 17. Frames from animations made with globs. The top two frames are from different moments in the same animation; the bottom two are from different animations.

part of the appeal of the glob primitive comes from the variety of shapes that can be produced with just a few controls, each of which has a specific geometric action.

6. Future Work

It would be appealing to construct globs with continuity in the second derivative or higher. To get C^2-continuity with this geometry, we'd have to give up the freedom of placing the F points where we like, which we feel would sacrifice too much of the variety and control that makes globs attractive in the first place. We are currently investigating other geometries and spline types, seeking the same simple and direct interface controls but providing C^2-continuity (or, better yet from a visual standpoint, G^2 geometric continuity). In the meantime, we note that being limited to C^1 parametric continuity isn't always such a bad thing: Max's cone-spheres are only C^1-continuous and they have proven to be a popular and useful primitive. We also observe that even without a constructive guarantee of C^2-continuity, most of the globs we have made are visually smooth. The examples in all the figures here are typical in this respect.

It's natural to think about promoting globs into 3D surfaces. It would be easy enough to create a surface of revolution by rotating any glob around a line, such as the line joining the circle centers. But this is a very limited technique, as it would produce rotationally symmetric shapes that lose much of the asymmetry that makes globs interesting. We are investigating a variety of 3D constructions for promoting a given glob into 3D while maintaining C^1-continuity, and for creating 3D globs directly with a minimal and direct set of design controls.

7. Conclusion

We've shown that with just two points and four scalars, a designer can create a graceful and highly controllable neck between two circles of different radii. The neck can curve asymmetrically, and the designer can control both the location of the thinnest (or thickest) part of the neck, and how sharply the neck rises to join the circle at each end.

Acknowledgements

Thank you to Steven Drucker and Eric Haines for their help in clarifying the presentation, and to John Hughes for enlightening conversations about patches.

Implementation

A basic implementation of this technique is offered by the `Glob` class in Listings 1, 2, and 3. The code is in Processing (a Java-like language). The full implementation in our open-source library, with both constructors discussed above, is part of the library's complete source code included in the library download [Glassner 2015a].

```
class Glob {

  // The input geometry of a Glob
  PVector C0, C1;       // circle centers
  float r0, r1;         // circle radii
  PVector D, Dp;        // target points D and Dprime
  float a, b, ap, bp;   // control scalars a, b, aprime, bprime

  // the points we derive from the input geometry
  PVector U, V;
  PVector E0, E0p, E1, E1p, F0, F0p, F1, F1p;

  // these are for picking which tangent point we want
  final int SIDE_LEFT = -1;
  final int SIDE_RIGHT = 1;

  // make a new Glob with these parameters.  Save them,
  // then build the geometry.
  Glob(PVector _C0, float _r0, PVector _C1, float _r1,
       PVector _D,  float _a,  float _b,
       PVector _Dp, float _ap, float _bp) {
    C0 = _C0.copy();
    C1 = _C1.copy();
    r0 = _r0;
    r1 = _r1;
    D = _D.copy();
    a = _a;
    b = _b;
    Dp = _Dp.copy();
    ap = _ap;
    bp = _bp;
    buildGeometry();
  }
```

Listing 1. An implemention of a Glob class in Processing, Part 1.

```
// This is the heart of the class.  Using the definition of Glob
// geometry, // find the coordinate system (U, V), then the
// tangent points E, and from them the Bezier control points F.
void buildGeometry() {
  U = PVector.sub(C1, C0);
  U.normalize();
  V = new PVector(U.y, -U.x);

  E0 = getTangentPoint(C0, D, r0, SIDE_RIGHT);
  E1 = getTangentPoint(C1, D, r1, SIDE_LEFT);
  E0p = getTangentPoint(C0, Dp, r0, SIDE_LEFT);
  E1p = getTangentPoint(C1, Dp, r1, SIDE_RIGHT);

  F0 = PVector.lerp(E0, D, a);
  F1 = PVector.lerp(E1, D, b);
  F0p = PVector.lerp(E0p, Dp, ap);
  F1p = PVector.lerp(E1p, Dp, bp);
}

// The P-finding routine.  Given a circle of center A
// and radius r, a point B outside the circle, and a
// choice of "side", return a point on the circle that
// forms a tangent line to the circle with point B.
PVector getTangentPoint(PVector A, PVector B, float r, int side) {
  PVector S = PVector.sub(B, A);
  S.normalize();
  PVector T = new PVector(S.y, -S.x);

  float ab = A.dist(B);
  float pb = sqrt(sq(ab)-sq(r));
  float beta = atan2(pb, r);
  float uscl = r * cos(beta);
  float vscl = r * sin(beta);
  PVector P0 = new PVector(A.x + (uscl*S.x) + (vscl*T.x),
                           A.y + (uscl*S.y) + (vscl*T.y));
  PVector P1 = new PVector(A.x + (uscl*S.x) - (vscl*T.x),
                           A.y + (uscl*S.y) - (vscl*T.y));
```

Listing 2. An implemention of a Glob class in Processing, Part 2.

```
    // now pick one or the other, using the sign of
    // cross products from each P to A with S
    PVector dP0 = PVector.sub(P0, A);
    PVector dP1 = PVector.sub(P1, A);
    float p0sgn = S.cross(dP0).z;
    float p1sgn = S.cross(dP1).z;
    // this next test should never succeed if r>0
    if (p0sgn * p1sgn > 0) { // both are negative or positive
      println("getTangentPoint:␣both␣points␣on␣same␣side␣of␣line!");
      return P0;
    }
    if (side == SIDE_RIGHT) { // the positive side is on the right
      if (p0sgn > 0) return P0;
      return P1;
    }
    if (p0sgn < 0) return P0;
    return P1;
  }

  // a utility method to draw the Glob in a straightforward way
  void render(boolean drawNeck, boolean drawCaps) {
    if (drawCaps) {
      ellipse(C0.x, C0.y, 2*r0, 2*r0);
      ellipse(C1.x, C1.y, 2*r1, 2*r1);
    }
    if (drawNeck) { // this shape is what it's is all about!
      beginShape();
        vertex(E0.x, E0.y);
        bezierVertex(F0.x, F0.y, F1.x, F1.y, E1.x, E1.y);
        vertex(E1p.x, E1p.y);
        bezierVertex(F1p.x, F1p.y, F0p.x, F0p.y, E0p.x, E0p.y);
      endShape(CLOSE);
    }
  }
}
```

Listing 3. An implemention of a Glob class in Processing, Part 3.

References

Bastl, B., Kosinka, J., and Lávička, M. 2015. Simple and branched skins of systems of circles and convex shape. *Graphical Models 78* (March), 1–9. 15

Blinn, J. F. 1982. A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1*, 3 (July), 235–256. 2, 15

Bloomenthal, J., Ed. 1997. *Introduction to Implicit Surfaces*. Morgan Kaufmann, Los Alamitos, CA. 12

Glassner, A. 2015. The AU library. Open-source library for the Processing graphics system. URL: `http://www.imaginary-institute.com/resources/AULibrary/AULibrary.php`. 13, 17

Glassner, A. 2015. Globs. Imaginary Institute Technical Note #11, April. URL: `http://imaginary-institute.com/resources/TechNote11/TechNote11.html`. 13, 21

Gourmel, O., Barthe, L., Cani, M.-P., Wyvill, B., Bernhardt, A., Paulin, M., and Grasberger, H. 2013. A gradient-based implicit blend. *ACM Transactions on Graphics 32*, 2. 12

Kamermans, M. P. 2015. A Primer on Bézier Curves. Online tutorial. URL: `http://pomax.github.io/bezierinfo`. 9

Ma, Y., Tu, C., and Wang, W. 2012. Distance computation for canal surfaces using cone-sphere bounding volumes. *Computer Aided Geometric Design 29*, 5 (June), 255–264. 15

Max, N. 1990. Cone-spheres. *Proceedings of SIGGRAPH '90*, 59–62. 2, 15

Shirley, P., Ashikhmin, M., and Marschner, S. *Fundamentals of Computer Graphics*. A K Peters, Natick, MA. 9

Stone, M., and DeRose, T. 1989. A geometric characterization of parametric cubic curves. *ACM Transactions on Graphics 8*, 3 (July), 157–163. 8

van Wijk, J. 1985. *Ray Tracing Objects Defined By Sweeping A Sphere*. PhD thesis, Delft University of Technology. 15

Wyvill, B., Galin, E., and Guy, A. 1999. Extending The CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Computer Graphics Forum 18*, 2 (June), 149–158. 2

Index of Supplemental Materials

An online, in-browser version of our interactive designer is available at [Glassner 2015b].

Author Contact Information

Andrew Glassner
The Imaginary Institute
726 North 47th St..
Seattle, WA 98103
andrew@imaginary-institute.com

Andrew Glassner, Globs: A Primitive Shape for Graceful Blends Between Circles, *Journal of Computer Graphics Techniques (JCGT)*, vol. 4, no. 3, 1–22, 2015
http://jcgt.org/published/0004/02/05/

Received:	2014-02-07		
Recommended:	2014-02-07	Corresponding Editor:	Alexander Wilkie
Published:	2014-02-07	Editor-in-Chief:	Morgan McGuire