Chapter 2
A Brush Stroke Synthesis Toolbox

Stephen DiVerdi

2.1 Introduction

Digital painting has progressed by leaps and bounds since its humble beginnings [30]. What some of the original painting applications called "brushes" were really simple line-drawing mechanisms based on the model of a circular tip, swept along a path. Still, within these systems, artists prevailed to make impressive images with the first inklings of a natural media style.

Things have advanced greatly since then. Much of the rest of this book deals with the rise of non-photorealistic rendering, and in some cases specifically painterly rendering, in which 3D models or 2D images and videos are processed automatically to have the appearance of a piece of natural media artwork. On the other hand, traditional media artists who consider moving into the digital media world look for the same level of expressive control and manual interaction that they have mastered with physical brushes. From this perspective, painting engines that aim to replicate or emulate real artistic tools represent the path forward, and there has been a correspondingly large amount of research in the field.

However, it is not the case that brush strokes are only interesting to manual digital painters. For example, in the automatic generation of painterly images from photos or 3D models, many algorithms are based around the idea of stroke-based rendering (SBR) [21], where the composition is broken down into a set of brush strokes that combine to form the target output. The primary distinction between SBR and brush simulation is that SBR assumes realistic brush strokes as an input to the algorithm, whereas brush simulation is concerned with the synthesis of those realistic brush strokes, but clearly the two areas are closely connected. SBR is discussed in detail in Chap. 1.

S. DiVerdi (✉)
Adobe Systems Inc., 601 Townsend St., San Francisco, CA 94107, USA
e-mail: stephen.diverdi@gmail.com

P. Rosin, J. Collomosse (eds.), *Image and Video-Based Artistic Stylisation*,
Computational Imaging and Vision 42, DOI 10.1007/978-1-4471-4519-6_2,

Algorithm 2.1 The swept stroke algorithm

```
1: function BASICSWEEP(𝕊, d)
2:     for i ← [1, n − 1] do
3:         DRAWLINE(d, s_i, s_{i+1})
4:     end for
5: end function
```

Therefore, brush strokes are a sort of atomic unit of non-photorealistic rendering, and brush stroke synthesis is a bread and butter algorithm that underlies the success of many different applications.

The basic and most commonly used algorithm for constructing brush strokes will be described in Sect. 2.2. It is a baseline for creating strokes that are parameterized by some input path and a selected brush tip, and is the core of many commercial applications and research systems. To go beyond this simple model, we will survey the available technologies for more realistic brush stroke synthesis from the research community in Sect. 2.3. In the remaining sections of the chapter, we will show step-by-step how to implement a more complex, physically based model, and discuss the tradeoffs at each stage of the process. By the end of this chapter, we will have collected a set of algorithmic tools that will allow us to tailor a brush stroke synthesis algorithm to satisfy constraints from real-world painting applications.

2.2 The Basic Brush Model

In the simplest case of brush stroke rendering, the user provides an input curve in the form of a 2D trajectory, uniformly sampled in time, from an input device such as a mouse or a tablet. The set $\mathbb{S} = \{\mathbf{s}_1, \ldots, \mathbf{s}_n\}$ is the sequence of these 2D samples. The goal is to make a stroke that follows this trajectory, with a simple circular brush of diameter d. Then, given a function DRAWLINE$(w, \mathbf{p}, \mathbf{q})$ that draws a line of width w from $\mathbf{p}$ to $\mathbf{q}$ (found in any 2D graphics package), the pseudocode in Algorithm 2.1 produces the desired result. See Fig. 2.1 for an illustration.

This model is fast and easy to implement, but also has some significant limitations. For example, it does not properly draw transparent strokes, because of the overlapping line endcaps at each sample point. It cannot create any kind of textured brush stroke, such as the smooth falloff of an airbrush or the scratchy appearance of a dry paint brush. It is unable to support variations along the stroke, such as controlling the size of the stroke by the pressure of a tablet stylus.

A straightforward amendment to this simple algorithm will enable a wider variety of brush strokes with a similar level of complexity. Rather than draw a line between consecutive sample points, we will walk along the input trajectory and place stamps of the brush footprint at uniform arc length. Given δ is the distance between stamps (called the "spacing"), INTERPOLATE$(\mathbb{S}, a)$ is a function that returns the 2D position along trajectory $\mathbb{S}$ at arc length a, Ω is the 2D raster image of the brush footprint, and STAMP$(\mathbf{p}, \Omega)$ is a function that draws the brush footprint at position $\mathbf{p}$.

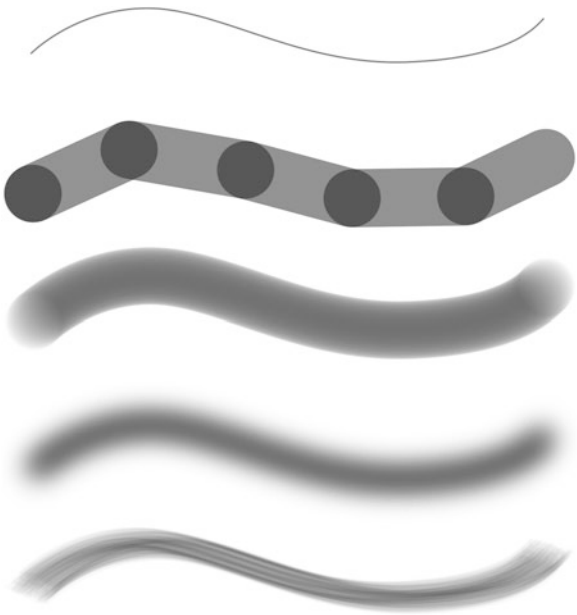


Fig. 2.1 A demonstration of the different output possible with the basic model. *Top* to *bottom:* the input trajectory, the output of BASICSWEEP, the output of BASICSTAMP with a circular stamp, with a gaussian blob stamp, and with a "brush" stamp

Algorithm 2.2 The stamped stroke algorithm

```
1: function BASICSTAMP(𝕊, Ω, δ)
2:     l ← ARCLENGTH(𝕊)
3:     a ← 0
4:     while a ≤ l do
5:         p ← INTERPOLATE(𝕊, a)
6:         STAMP(p, Ω)
7:         a ← a + δ
8:     end while
9: end function
```

Algorithm 2.2 is the core of most digital painting brush engines. Selection of the brush footprint image can create different stroke textures (e.g. a filled circle, a gaussian blob, or a pattern of dots—see Fig. 2.1). Extending the STAMP function to support affine transforms allows the size and distortion of the stamp to be controlled dynamically by tablet parameters such as pressure and tilt. Trajectory smoothing can be handled by modifying INTERPOLATE. Transparency and buildup are correctly handled within STAMP. Finally, dynamic brush footprints are possible by parameterizing Ω by additional parameters, such as the output of a physical simulation.

These two examples, BASICSWEEP and BASICSTAMP, also demonstrate a fundamental difference between stroke synthesis algorithms. "Tweening" is short for "in-betweening", and refers to the generation of intermediate states between sample

points. Sweeping and stamping are two ways to address the problem, and roughly correspond to analytical and numerical solutions. Stamping has the limitation that the necessary sampling rate (i.e. spacing) can be high depending on the frequency content of the brush footprint. Sweeping on the other hand, is limited to strokes that can be analytically computed in an efficient manner, which makes effects like textured appearances difficult. This issue will be discussed in more detail in Sect. 2.6.

2.3 Available Technologies

Particularly in the last decade, realistic brush stroke synthesis has been an active area of research. Major contributions in the field are summarized in Table 2.1. Much of this work has been conducted as part of larger agendas around the simulation of paint media as well, such as watercolor or oil paint, but these results are out of the scope of this text and are not considered here. In addition to research work, we also consider some of the most popular commercially available applications for digital painters [2, 3, 16, 18].

To help understand the space of different technologies available, we have categorized the contributions of each work based on common themes. These categories are the output type, the algorithm strategy, the spline model, the brush head model, the solution method by which new states are computed, and the type of tweening used. Each of these categories is discussed in turn.

Output Type The options for the output of a brush stroke synthesis algorithm are raster or vector. Raster is the most common type, and means that the output brush stroke is a 2D grid of pixels. Conversely, vector output represents the brush stroke as a set of filled, Bézier-bounded paths instead of pixels (n.b. vector representations can be more complex, but for the purposes of this text, this definition is sufficient). Adobe Illustrator [2] is a common example of a program that outputs vector brush strokes. Vector representations are desirable because they are resolution independent and sparse (for compactness of storage), but they are limited in terms of the types of appearance they can represent. Soft airbrush strokes or textured dry brush strokes are generally not supported by vector algorithms.

Algorithm Strategy The different strategies for creating brush strokes can be grouped into a few bins: procedural, simulation, acquisition, and data-driven. Procedural strategies include those of commercial painting applications and are based on simple, ad hoc models that can be controlled programmatically to create brush strokes, without modeling a virtual 3D brush head. Alternately, simulation is the approach of computing the dynamics of a physical system that mimics a real brush. Rather than trying to compute the brush footprint shape, the acquisition strategy uses custom hardware to measure the shape of a real brush in real-time. Finally, data-driven approaches record real brushes offline and reproduce their effects via machine learning of some form.

Table 2.1 Overview of existing implementations and their characteristics. *Output:* raster (R) or vector (V). *Strategy:* acquisition of real brush footprints (A), physical simulation (S), data-driven methods (D), or procedural methods (P). *Spline:* for work that models splines, are they discrete (D) or continuous (C). *Brush:* for work that models a brush head, is it a mesh (M), skeleton (S), interpolated geometry (I), or individual bristles (B). *Solution:* systems can be solved via integration (I), energy minimization (M), or data-driven methods (D). *Tween:* short for in-between, intermediate states between simulation steps can be generated by stamping (T) or sweeping (W)

work	output	strategy	spline	brush	solution	tween
Adobe Illustrator [2]	V	P	–	–	–	W
Adobe Photoshop [3]	R	P	–	–	–	T
Ambient Design ArtRage [18]	R	P	–	–	–	TW
Bai et al., 2007 [5]	R	P	–	–	–	T
Baxter at al., 2001 [9]	R	S	D	M	I	T
Baxter and Govindaraju, 2010 [7]	R	D	C	M	D	T
Baxter and Lin, 2004 [8]	R	S	D	MI	M	T
Chu and Tai, 2004 [14]	R	S	D	S	M	T
Chu, 2007 [13]	R	S	D	S	M	W
Corel Painter [16]	R	P	–	–	–	TW
DiVerdi at al., 2010 [19]	RV	S	C	B	I	TW
Lu and Huang, 2007 [25]	R	S	C	B	M	T
Mi et al., 2002 [26]	R	P	–	–	–	W
Okabe et al., 2005 [27]	R	A	–	–	–	T
Pudet, 1994 [28]	V	P	–	–	–	W
Saito and Nakajima, 1999 [29]	R	S	C	M	M	T
Van Laerhoven and Van Reeth, 2007 [32]	R	S	D	MI	M	T
Vandoren et al., 2009 [34]	R	A	–	–	–	T
Vandoren et al., 2008 [33]	R	A	–	–	–	T
Xie et al., 2010 [35]	R	P	–	–	–	W
Xu et al., 2004 [36]	R	D	–	MB	–	T

Spline Model For algorithms that work by creating a virtual brush head, the spline is a basic building block that can deform in bristle-like ways to control the shape of the brush head and thus footprint. These splines can be modeled discretely, as a piecewise linear approximation consisting of sample positions joined by straight line segments. Discrete models lend themselves to easy computation of physical dynamics and collisions, but may require many segments for acceptable quality. Continuous splines are also possible, using e.g. a Bézier or helix, and manipulating the control points to change the shape. Collision and dynamics are more complicated, but the resulting shape will always be smooth.

Brush Head Model Given a spline or set of splines, the brush head model controlled by the spline(s) can be of varying levels of complexity. In the simplest case, the brush head is represented as a single bulk triangle mesh, which is deformed by

the shape of a control spline running through it. A more structured brush head can use a skeleton of splines, branching off one another, to provide a way to control the (potentially asymmetric) changes in spread of the brush. To represent the brush head as a collection of bristles, each piece of bristle geometry can be controlled by an individual spline, for maximum fidelity, but at a large computational cost. A faster way to model a collection of bristles is by interpolating the bristle geometry from a few control splines.

Solution Method When computing a physical simulation of spline dynamics, the most straightforward way to update the shape is by integration of the internal and external forces. As brush bristles tend to be stiff systems, implicit integration is necessary. Furthermore, bristles tend to achieve their rest shape very quickly in the presence of changed forces, and so a commonly explored alternative is to compute the quasi-static configuration via energy minimization. Potentially fastest of all, data-driven solutions can determine the rest shape without costly math.

Tweening As discussed in Sect. 2.2, tweening refers to the generation of intermediate states between samples. For brush strokes, there are two options. Stamping is the numerical approach that works by computing many samples, whereas sweeping is the analytical approach that attempts to compute the final result in a single step.

2.4 Spline Modeling

To begin our more realistic brush stroke synthesis, in this section we construct the spline model. Our spline is comprised of a chain of rigid links connected by hinge joints. This is a discrete model, which allows for a simpler formulation of the spline dynamics than a continuous model, at the cost of some fidelity. An illustration of the spline model structure can be seen in Fig. 2.2.

A canvas is necessary to provide a surface for the spline to collide with, to create the characteristic deformations. We will use a simple plane to represent the canvas surface. The canvas will also provide friction forces to cause the bristles to drag appropriately. To control the brush, we attach the base link of the spline to a brush handle cylinder. This handle has a position and orientation that is controlled by the user's input. Ideally, a tablet and stylus with six degrees of freedom (2D position, pressure, 2D tilt, and barrel rotation) is available, which can provide a direct mapping between all the input DOFs and the handles'. Otherwise, some sort of indirect control must be defined.

As bristles are stiff, the spring constants for the spline joints must be high. This creates a stiff system that means we will need to use an implicit integration scheme.

Physical simulation is a complex topic and an active area of research, which has been covered many times in other texts [6]. Getting into the details of how to

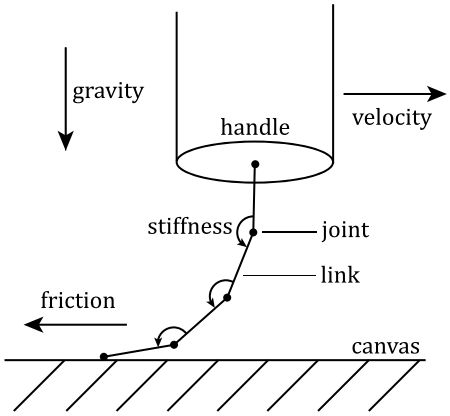


Fig. 2.2 The basic model of a brush spline as a chain of rigid links connected by stiff joints. The user controls the handle's position and velocity, while gravity and the canvas deform the spline

construct and solve a system of equations based on our system design would be quite involved and would distract us from our goal of brush stroke synthesis. However, the purpose of this chapter is to provide the reader with practical tools to achieve the effects we have discussed. Therefore, to implement our brush simulation, we will employ a physics engine.

Free, open source physics engines are available today largely thanks to their rising popularity in the game industry and the maturation of the mathematical techniques to implement fast and robust simulations. Some of the more popular examples include ODE [31], Bullet [17], and Tokamak [23]. There are even efforts to define a common API across all the engines to allow for decoupling of a program's system definition from the underlying engine [11, 12]. Each library emphasizes different aspects of the problem, such as focusing on robustness or on real-time performance. Support for advanced features like soft bodies and fracturing varies. However, the formulation of our spline model is relatively simple and can be supported by any of these engines. Therefore, we will use ODE for the remainder of this section, because of its prioritization of absolute robustness above all else.

A comment on notation: in the algorithm listings in this section, function names that begin with a 'd' and are typeset like `dWorldCreate()` refer to functions in the ODE API. Consult the ODE documentation for the particulars of input arguments, return values, and types.

2.4.1 Creating a Spline

Creating the basic brush and spline geometry in ODE is straightforward, and is addressed in Algorithm 2.3. The CREATEHANDLE function initializes the ODE world and creates the brush handle object. Multiple brush handles could exist in the same world, but we will assume there is one. Then CREATESPLINE can be used multiple times to add individual splines to the brush. ODE's model is such that rigid bodies

Algorithm 2.3 Creating the brush geometry

```
1: function SetMass(b)
2:     M ← dMassSetCapsule(radius, length)
3:     dMassSetMass(M, mass)
4:     dBodySetMass(b, M)
5: end function
6:
7: function CreateHandle()
8:     W ← dWorldCreate()
9:     dWorldSetGravity(0, 0, g)
10:    H ← dBodyCreate(W)                          ▷ brush handle body
11:    SetMass(H)
12:    I_body ← dGetInertialTensor(H)              ▷ store for later
13: end function
14:
15: function CreateSpline()
16:    for i ← [1, n] do                           ▷ create each chain link
17:        b_i ← dBodyCreate(W)
18:        dBodySetPosition(b_i, px_i, py_i, pz_i)     ▷ initialize link pose
19:        dBodySetQuaternion(b_i, qx_i, qy_i, qz_i, qw_i)
20:        SetMass(b_i)
21:    end for
22:    for i ← [1, n] do                           ▷ create joints between links
23:        j_i ← dJointCreateUniversal(W)
24:        if i = 1 then
25:            dJointAttach(j_i, H, b_i)           ▷ attach first link to handle
26:        else
27:            dJointAttach(j_i, b_{i−1}, b_i)
28:        end if
29:        dJointSetUniversalAnchor(j_i, px_i, py_i, pz_i)
30:        dJointSetUniversalAxis1(j_i, 1, 0, 0)       ▷ x-axis hinge
31:        dJointSetUniversalAxis2(j_i, 0, 1, 0)       ▷ y-axis hinge
32:    end for
33: end function
```

are represented as objects that store a position, orientation, and a mass and inertia tensor. Joints are separate objects that define a connection between two bodies. Collision detection is done on separate geometry objects that are associated with bodies, to decouple the dynamics from the collision detection, and is discussed in Sect. 2.4.4.

CreateSpline specifically mentions creating "universal" joints, which are a standard concept in mechanical systems that allows two degrees of freedom—rotation about the two axes perpendicular to the major axis. A third DOF is twist about the major axis, which a universal joint restricts. An alternative joint type is

Algorithm 2.4 Animating the brush geometry

```
1: function StepSimulation()
2:     m ← number of substeps
3:     for i ← [1, m] do
4:         AddBrushForceAndTorque()                ▷ see Sect. 2.4.2
5:         dSpaceCollide2(G, S_s, C)               ▷ see Sect. 2.4.4
6:         dWorldStep(Δt)
7:         dJointGroupEmpty(C)
8:     end for
9: end function
```

"ball and socket", which allows all three DOFs and is supported by ODE. Full physical hair simulation, particularly for long hair such as on a virtual character's head, includes twist in its formulation because it has a significant impact on the overall dynamic behavior, but for short, stiff brush bristles, it is generally unnecessary.

Once the handle and spline bodies and joints have been created, animating them is handled by the StepSimulation function in Algorithm 2.4, which should be called once per frame in the application's main loop. Simulation in its most basic form consists of calling `dWorldStep`, but with gravity as the only force on the spline, its behavior will not be particularly interesting. Next we will see how to add user control for a more interactive simulation.

2.4.2 User Control

Once the virtual brush handle and spline have been constructed, the user input needs to be integrated to allow for control of the brush. We assume that the user has some mechanism of providing the six degrees of freedom (DOF) necessary to position and orient the brush handle. Ideally this would come from directly manipulating the six DOF of a tablet stylus, but other approaches are fine as well.

The instantaneous state of the brush handle is $\Gamma = \langle \mathbf{p}, \mathbf{v}, \mathbf{q}, \omega \rangle$, where $\mathbf{p} = \langle p_x, p_y, p_z \rangle$ is the 3D position, $\mathbf{v} = \langle v_x, v_y, v_z \rangle$ is the 3D velocity, $\mathbf{q} = \langle q_x, q_y, q_z, q_w \rangle$ is the orientation quaternion, and $\omega = \langle \omega_x, \omega_y, \omega_z \rangle$ is the 3D angular velocity.

At time t_0, the state is $\Gamma_0 = \langle \mathbf{p}_0, \mathbf{v}_0, \mathbf{q}_0, \omega_0 \rangle$. User input specifies the position and orientation $\langle \mathbf{p}_1, \mathbf{q}_1 \rangle$ at the end of the timestep, $t_1 = t_0 + \Delta t$. Therefore, we must compute the force and torque to apply to the brush handle to achieve the change in state over the timestep. The relevant equations are

$$\mathbf{f} = \frac{m}{\Delta t^2}(\mathbf{p}_1 - \mathbf{p}_0 - \mathbf{v}_0 \Delta t) \tag{2.1}$$

$$\omega_1 = \frac{2}{\Delta t}(\mathbf{q}_1 - \mathbf{q}_0)\mathbf{q}_0^{-1} \tag{2.2}$$

$$\mathrm{I}_0 = \mathrm{R}(\mathbf{q}_0)\, \mathrm{I}_{\mathrm{body}}\, \mathrm{R}(\mathbf{q}_0)^{\mathsf{T}} \tag{2.3}$$

$$\tau = \frac{1}{\Delta t} \mathrm{I}_0(\omega_1 - \omega_0) \tag{2.4}$$

where m is the brush handle's mass, $\mathrm{R}(\mathbf{q})$ is the 3×3 rotation matrix corresponding to $\mathbf{q}$, and $\mathrm{I}_{\mathrm{body}}$ is the brush handle's 3×3 inertia tensor in local coordinates. In Eq. (2.2), the 3D ω_1 is constructed from the 4D quaternion product by omitting the quaternion's w-component. The results of these equations are $\mathbf{f}$, the 3D force vector and τ, the 3D torque vector, which are applied to the brush handle in ODE using `dBodyAddForce` and `dBodyAddTorque`.

It is helpful to break this process up into substeps, to improve ODE's stability and convergence (see Algorithm 2.4). If the desired simulation step is $\Delta t = 0.016$ s (corresponding to 60 Hz), then with three substeps, the timestep becomes $\Delta t = 0.005$ s. Furthermore, the user input specifies $\langle \mathbf{p}_3, \mathbf{q}_3 \rangle$, and the force and torque computations must be done three times to compute the three substeps from Γ_0.

With user input-based control of the brush handle, the simulation should be more interesting—changing the position and orientation will result in forces being applied to the spline, so it should swing around like a limp rope. Two problems remain however. First, it does not collide with the canvas, and second, the spline has no stiffness. We will add stiff springs next.

2.4.3 Adding Springs

Because brush bristles are stiff, our spline needs stiff springs at each of the joints to maintain its rest shape. For example, a real brush's bristles do not droop due to gravity, because the strength of the bending stiffness is greater than the effect of gravity on the bristle's mass. Due to the stiff behavior, we must construct the springs in such a way that they are implicitly integrated by ODE.

External body forces in ODE are integrated explicitly, but constraint forces are integrated implicitly. Constraint forces are represented with joints in ODE. As we have universal joints constructed between each consecutive pair of links in the spline, we can modify them to add springs. Specifically, each joint has a notion of "joint limits", which restrict the range of motion the joint allows about its degrees of freedom (think of an elbow and how the forearm cannot bend back past a certain angle). As joint limits are hard constraints, they are implicitly integrated, and so are ideal for our uses.

Algorithm 2.5 contains the SETJOINTSPRINGS function which sets the parameters of a universal joint to have a straight rest shape of some springiness. The low and high stops are both set to $\theta = 0$, which is the rest angle. $f_{\max}$ is the maximum restorative force that can be applied. The CFM and ERP parameters determine the spring and damping coefficients of the constraint. ERP stands for the Error Reduction Parameter, in the range of $[0, 1]$, and represents how much of a constraint violation is corrected in each timestep (for hard contacts, e.g., it would seem a value of 1 would be desired, but this can lead to instability). The CFM is the

Algorithm 2.5 Adding bristle stiffness

```
1: function SETJOINTSPRINGS()
2:     dJointSetUniversalParam(j, dParamLoStop, θ)
3:     dJointSetUniversalParam(j, dParamHiStop, θ)
4:     dJointSetUniversalParam(j, dParamFMax, f_max)
5:     dJointSetUniversalParam(j, dParamStopERP, ERP)
6:     dJointSetUniversalParam(j, dParamStopCFM, CFM)
7: end function
```

Constraint Force Mixing, in $(0, \infty)$, a measure of how much a constraint is allowed to be violated. These two parameters are fundamentally related to the spring and damping coefficients, k_p and k_d respectively, of the joint, and can be computed by

$$\text{ERP} = \frac{\Delta t k_p}{\Delta t k_p + k_d} \tag{2.5}$$

$$\text{CFM} = \frac{1}{\Delta t k_p + k_d} \tag{2.6}$$

After the joint springs have been set, our spline model should be much less rope-like. Without something to collide with, the only way to see the spline bend is by moving the brush quickly, or by increasing gravity (or decreasing the spring coefficient). Finally we will add collisions, to complete the spline model.

2.4.4 Adding Collisions

Collisions in ODE are orthogonal to the dynamics by design, so applications that need collisions but not dynamics (e.g. collision detection to keep players from navigating through walls in virtual environments) can be supported, as well as simulations that have specific collision detection requirements (such as a-priori knowledge about the types of objects that will be colliding with one another).

Creation of the collision objects is shown in Algorithm 2.6's CREATECOLLIDERS function. Geometry objects are contained in spaces for organization, and associated with body objects for bookkeeping. Geometry objects and body objects need not be the same shape or mass (a common optimization is to represent an entire virtual avatar with a single capsule), but for now we will make them the same. Also a ground plane is necessary for the spline to collide with.

For collisions to be detected, `dSpaceCollide2` must be called in STEPSIMULATION. This function checks the geometry (in our case the ground plane) against the geometry in the space (containing the spline) for potential collisions by using an approximation such as overlapping bounding boxes. When it detects a potential collision, it calls a callback function DETECTCOLLISION in Algorithm 2.7 that

Algorithm 2.6 Creating brush collision objects

```
1: function CREATECOLLIDERS()
2:     C ← dJointGroupCreate()                          ▷ holds contacts
3:     S_w ← dSimpleSpaceCreate()                       ▷ collision space for world
4:     S_s ← dSimpleSpaceCreate(S_w)                    ▷ collision space for spline
5:     G ← dCreatePlane(S_w, 0, 0, 1, 0)                ▷ x, y ground plane
6:     for i ← [1, n] do                                ▷ create each link collision object
7:         g_i ← dCreateCapsule(S_s, r, l)
8:         dGeomSetBody(b_i, g_i)                       ▷ attach collision object to link body
9:     end for
10: end function
```

Algorithm 2.7 Handling collisions

```
1: function DETECTCOLLISION(g_1, g_2)
2:     if dCollide(g_1, g_2) then
3:         c ← dJointCreateContact(W, C)                ▷ stored in contact group
4:         b_1 ← dGeomGetBody(g_1)
5:         b_2 ← dGeomGetBody(g_2)
6:         dJointAttach(c, b_1, b_2)
7:     end if
8: end function
```

uses `dCollide` to find a specific contact point, and if it is successful, creates a temporary joint object to represent the contact.

In this manner, collisions are detected and responded to for our spline, and once this functionality is added, we should have a springy spline that we can press into the canvas to cause characteristic bristle-like deformations.

2.5 Brush Head Modeling

A deformable spline is an important building block towards creating a brush stroke, but by itself it is incomplete. Brush heads normally contain hundreds of bristles, but it is not currently feasible to simulate hundreds of splines with high enough fidelity. Therefore, different approaches are used to "bulk up" a simulated brush head, by using a small number of splines to control a larger amount of geometry. We will consider two such approaches. The first creates additional un-simulated bristles by interpolating among splines, which is very fast but still creates a large amount of geometry. The second represents many bristles as a single triangle mesh which is deformed by splines. The mesh is even faster, but is more limited in what types of effect it can reproduce.

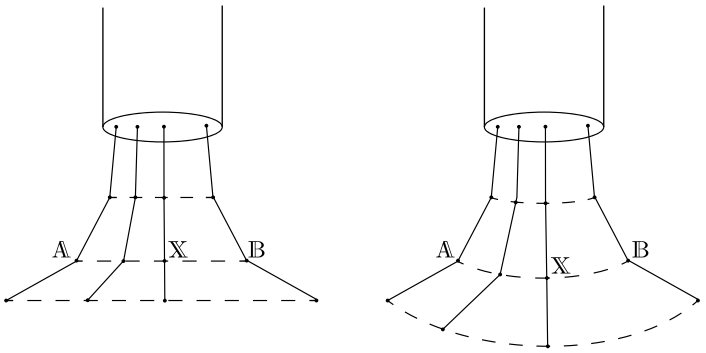


Fig. 2.3 Bristle $\mathbb{X}$ can be linearly interpolated between splines $\mathbb{A}$ and $\mathbb{B}$ either based on the vertex positions (*left*) or on the joint angles (*right*)

2.5.1 Interpolation-Based

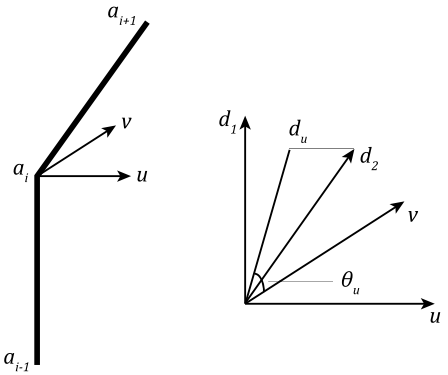
Regardless of simulation strategy, the output of a spline-based model is a sequence of 3D positions of points along the spline in its new shape. Call this sequence, $\mathbb{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$. Then if there are three splines, $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$, arbitrary linear interpolations can be created from them as

$$\mathbb{X}(\alpha, \beta, \gamma) = \alpha\mathbb{A} + \beta\mathbb{B} + \gamma\mathbb{C} \tag{2.7}$$

$$\mathbf{x}_i(\alpha, \beta, \gamma) = \alpha\mathbf{a}_i + \beta\mathbf{b}_i + \gamma\mathbf{c}_i, \quad \forall i \in [1, n] \tag{2.8}$$

This linear interpolation of positions is fast and simple, but has the downside that it can result in interpolated splines that are shorter in length than the simulated splines. See Fig. 2.3 for an illustration. The magnitude of this effect depends on the difference among the splines being interpolated, and so for larger numbers of simulated splines, or for less severe brush head deformations, it may be sufficient.

Rather than interpolating splines' 3D positions $\mathbb{A}$, another option is to interpolate their 2D joint angles instead, $\overline{\mathbb{A}}$. Then after computing $\overline{\mathbb{X}}$ as a linear combination of spline joint angles, the spline positions $\mathbb{X}$ can be constructed with knowledge of the interpolated spline's link lengths. This will ensure interpolated splines have the same length as simulated splines, but has the downside that it could result in splines that penetrate the canvas plane. To compute $\overline{\mathbb{A}}$ from $\mathbb{A}$, each joint is defined by three consecutive points, $\mathbf{a}_{i-1}$, $\mathbf{a}_i$, and $\mathbf{a}_{i+1}$, where $\mathbf{a}_i$ is the joint location, and $\mathbf{u}$ and $\mathbf{v}$ are the joint's local right and up vectors. Then, we define $\mathbf{d}_1$ as the default direction for the joint and $\mathbf{d}_2$ as the deformed direction, such that $\mathbf{d}_1 \cdot \mathbf{u} = \mathbf{d}_1 \cdot \mathbf{v} = 0$. To find the angles about the $\mathbf{u}$ and $\mathbf{v}$ vectors, we can use



$$\mathbf{d}_1 = \frac{\mathbf{a}_i - \mathbf{a}_{i-1}}{\|\mathbf{a}_i - \mathbf{a}_{i-1}\|} \tag{2.9}$$

$$\mathbf{d}_2 = \frac{\mathbf{a}_{i+1} - \mathbf{a}_i}{\|\mathbf{a}_{i+1} - \mathbf{a}_i\|} \tag{2.10}$$

$$\mathbf{d}_u = \mathbf{d}_2 - (\mathbf{d}_2 \cdot \mathbf{u})\mathbf{u} \tag{2.11}$$

$$\theta_u = \cos^{-1}\left(\frac{\mathbf{d}_u \cdot \mathbf{d}_1}{\|\mathbf{d}_u\|}\right) \tag{2.12}$$

(with corresponding equations for θ_v). Then θ_u and θ_v are the joint angles, which allows us to construct $\overline{\mathbf{a}_i} = \langle \theta_u, \theta_v \rangle$.

To do the reverse transform and compute $\mathbb{A}$ from $\overline{\mathbb{A}}$, starting at the base of the spline with $\mathbf{d}_1$, the up and right vectors, $\mathbf{u}$ and $\mathbf{v}$ and the joint angles θ_u and θ_v, we can use the equation

$$\mathbf{d}_2 = \mathbf{d}_1 + \mathbf{u}\tan\theta_v + \mathbf{v}\tan\theta_u \tag{2.13}$$

where $\mathbf{d}_2$ is the direction of the subsequent spline link.

Regardless of the scheme used, interpolation can be used to generate bristles inside the triangle defined by any three splines. Therefore, to produce the best bristles, create as many splines as possible to simulate distribute around the head of the brush, and then triangulate their positions to determine which splines should be interpolated for new bristles at arbitrary locations.

2.5.2 Mesh-Based

Creating interpolated splines can make a nice bristly appearance for scratchy strokes, but for smoother paint application, modeling the brush head as a continuous mesh may be more desirable. In that case, one or more splines can be used to control the deformation of one or more triangle meshes that represent a bulk collection of bristles and wet paint. We will discuss here how to control a single triangle mesh, but a brush head can be modeled by multiple meshes to support a wider range of behaviors. For example, a wide flat brush could benefit from multiple meshes to allow it to support splitting deformation, which a single mesh would not allow.

Deforming a mesh by a collection of splines is a straightforward application of vertex skinning, also known as vertex blending or skeleton-subspace deformation [24], which is a commonly used algorithm in the realtime 3D video games field for character mesh animation based on an animated skeleton of rigid bones. The approach is well-documented in many places [10], which we will summarize here. See Fig. 2.4 for an overview.

The core concept is that each bone's new position defines a transformation matrix that can be applied to any vertex in the mesh. Thus, by assigning a weight to the

Algorithm 2.8 Computing vertex weights

```
1: function FindNearestNeighbors(v, n, 𝕊)
2:     return indices of n closest points to v in 𝕊, sorted by distance
3: end function
4: function AssignWeights(n)
5:     for all v ∈ 𝕍 do
6:         {i_1, ..., i_{n+1}} ← FindNearestNeighbors(v, n + 1, 𝕊)
7:         max ← ‖v − s_{i_{n+1}}‖
8:         min ← ‖v − s_{i_1}‖
9:         for j ← [1, n] do
10:            d ← ‖v − s_{i_j}‖
11:            w_j ← 1 − (d − min)/(max − min)
12:        end for
13:        for j ← [1, n] do
14:            w_j ← w_j / Σ_k w_k                ▷ normalize weights
15:        end for
16:    end for
17: end function
```

influence of each bone per-vertex, the new position of the vertex is generated by a linear combination of the bones' transformations. Generally, a vertex has only a small number of non-zero weights, between two and four. In our case, the bones are each link of the spline. Therefore, a mesh vertex position can be computed as

$$\mathbf{v}' = \sum_i w_i \mathrm{M}_i \mathbf{v} \tag{2.14}$$

where w_i are the bone weights, M_i are the bone transformation matrices, from model coordinates to world coordinates, and $\mathbf{v}'$ is the output vertex position. This can be efficiently computed inside a vertex shader on modern graphics cards.

Once a triangle mesh and set of splines have been specified, the vertices' bone weights can be specified by hand, though this may be a tedious process. Alternately, they can be computed automatically based on the distance of each vertex to the bones. Consider the pseudocode in Algorithm 2.8, which assigns n weights per-vertex for a mesh of vertices $\mathbb{V}$ and the spline $\mathbb{S}$.

Mesh skinning is still an active area of research, and so as new techniques are developed, they may be applied here, if they are found to have more desirable properties [22].

While mesh-based brush heads have an advantage over bristle-based, in that they can have significantly less geometry and can produce smoother strokes, they have difficulty reproducing some brush head behaviors. Specifically, tip-spreading is a common effect where, for example, the point of a round brush will spread out when it is pressed into the canvas. Using multiple splines to control a single mesh can mimic this behavior, or other workarounds may be employed depending on the quality of result needed.

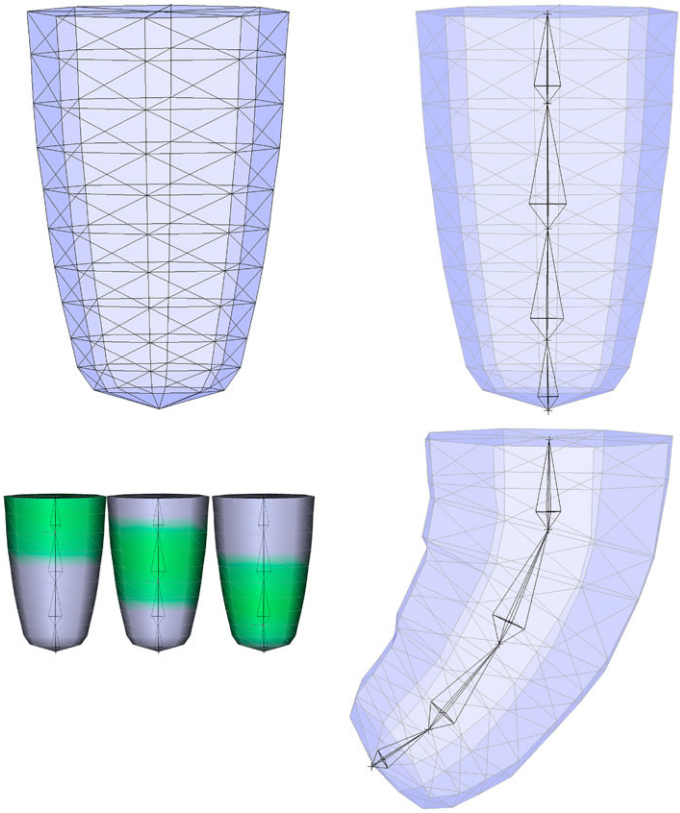


Fig. 2.4 A triangle mesh based brush head modeled in a common 3D modeling application. *Left* to *right*, *top* to *bottom:* The triangle mesh. The bones corresponding to spline links, placed in the mesh. The mesh vertex weights for the first three bones, computed by Art of Illusion [20]. The mesh is deformed based on the position of the bones by Eq. (2.14)

2.6 Stroke Rendering

At this point, our virtual brush has a physically simulated deformable head that is represented either by a set of bristles or as a triangle mesh. Using it to create the brush stroke output is the final step of this chapter. This process is non-trivial for a number of reasons, and the tradeoffs made here have a significant impact on the final output of our algorithm.

The first issue is that physical simulation is expensive to compute. It might be appealing to perform a simulation step for each new position of the brush (perhaps after some number of pixels have been traversed), but realistically, users can move the brush at arbitrary speeds across large canvases, creating a prohibitive computation burden. Furthermore, physics engines in general and ODE in particular, are significantly more stable when they are stepped with uniform time increments. Therefore, our strategy is to compute a constant number of physics steps per update of the

application (generally per frame render). As fast strokes mean there will be a significant pixel distance between consecutive steps, the question becomes how to fill in the pixels in between.

As discussed in Sect. 2.2, tweening is the term for generating in-between samples when filling in a sparsely computed system (the source of the term is 2D animation, where smooth transitions are drawn between artists' keyframes). We will consider both the numerical approach of stamping, and the analytical approach of sweeping, from the perspectives of bristle-based and mesh-based brush heads.

It is interesting to note that brush strokes are inherently inexact—that is, it is difficult to quantify a specific desired output for a set of parameters. There is an important question of how much fidelity is necessary to produce a usable painting system, which has been partially answered by the availability of digital painting programs to date. Therefore, the process of generating the output brush stroke from a physical simulation is as much a matter of taste as it is of science, and hence there are many opportunities to tune the particulars of an implementation.

2.6.1 Stamping

Stamping is the numerical approach to generating the continuous output of a brush stroke. Rather than trying to directly compute the value of each pixel covered by a stroke, the instantaneous pose of the brush is computed many times at small increments, and their effect is combined to generate the final output. This has a number of important implications to our algorithm design. Since this is the method of choice for commercial digital painting applications, the design space has been extensively explored already.

First, the instantaneous stamp needs to be computed. For a virtual brush head, we call this the brush's "footprint"—the contact area between the bristles and the canvas, where paint is deposited. The output of our brush head model is some geometry that we could test for actual canvas contact, but that would likely be a very small area and would not reflect the size of the mark we would expect such a brush to make. Realistically, a brush head has wet paint on it which creates a larger contact area with the canvas than just the bristle geometry. Therefore, we use what we call a "deposition threshold" which is a plane slightly offset from the canvas, and state that any geometry in the volume between the deposition threshold and the canvas is considered to be in contact (see Fig. 2.5).

For a triangle mesh brush head, computing the footprint is therefore very easy. The mesh is simply rasterized using a standard rendering library such as OpenGL, with an orthographic projection setup looking down at the canvas and the deposition threshold set as the depth clipping distance (the camera's near or far plane).

With a bristle brush, from interpolation or individual bristles, each bristle can be rendered in the same OpenGL setup as a bent cylinder. This geometry is straightforward to compute on the CPU, or a geometry shader can be used in OpenGL to generate it automatically on the GPU, but either way it results in a significant

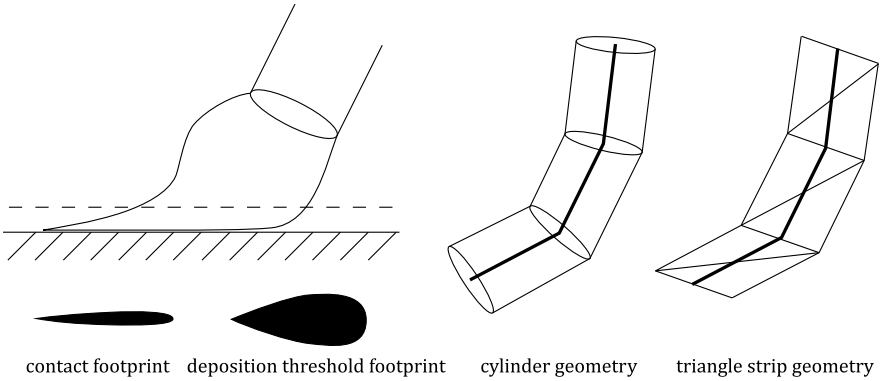


Fig. 2.5 *Left:* The deposition threshold is offset slightly from the canvas surface and results in a significantly larger brush footprint than the exact contact area. *Right:* An individual bristle can be represented as a bent cylinder or as a triangle strip

amount of geometry to be processed. An alternative is to render each bristle as a strip of triangles, oriented perpendicular to the camera view direction. This way the width of the bristle can still be explicitly controlled, but with a significantly smaller geometry burden. These options are illustrated in Fig. 2.5.

The second important consideration for stamping is how frequently to place the stamps, called the brush "spacing". Because each stamp has an associated cost in terms of memory bandwidth and computation, the best performance will come from placing as few stamps as possible. However, the frequency content of the stamp images dictates how closely spaced stamps must be placed to create a stroke without sampling artifacts (see Fig. 2.6). For example, the commonly available gaussian blob brushes in programs like Photoshop [3] have only very low frequencies, and so can generally be used at high spacings of around 25 % of the size of the brush without artifact. On the other hand, hard-edged circular brushes at 25 % create obvious stepping and un-smooth silhouette problems, and require spacings closer to 1–5 %. For our brush heads, thin individual bristles can be as small as a single pixel, and worse, since they are dynamic within the stamp image, even a spacing of 1 pixel may be insufficient. Tricks like using wider bristles and applying a small gaussian blur or a motion blur can help reduce the problem.

The other difficulty of spacing is that it can make proper coverage and cumulative effects difficult to calculate. For example, while it is easy to apply an alpha value to each stamp, the number of overlapping stamps at a single pixel is determined by a combination of the footprint and the spacing rate, and can mean that changing either one alters the final stroke's transparency. This issue is further compounded when bidirectional (brush $\leftrightarrow$ canvas) pigment transfer is considered, as discussed by Chu et al. [15].

Finally, the last concern with stamping is how to interpolate all the data to create the intermediate stamps, such as the x, y stamp position and the intermediate brush head configuration. Linear interpolation is straightforward, but creates piecewise linear brush strokes that are immediately apparent and undesirable. Smooth strokes

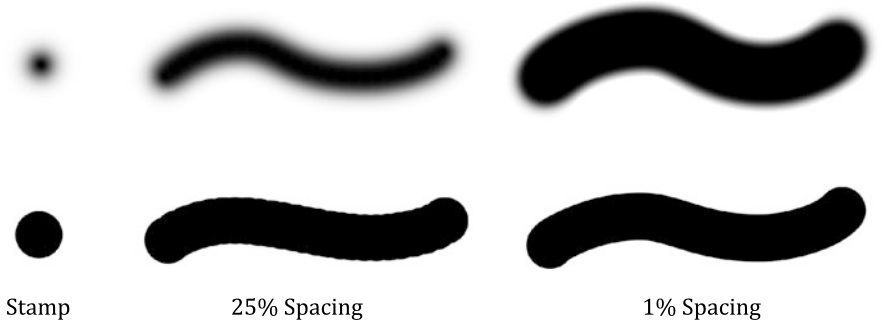


Fig. 2.6 Stamping artifacts depend on the stamp used and the spacing. For a low frequency stamp (*top*), a high spacing produces a smooth stroke while a low spacing causes more alpha accumulation than desired. For a high frequency stamp (*bottom*), a low spacing results in silhouette artifacts

are the result of interpolating the x, y positions with a higher order technique such as cubic interpolation or Bézier fitting [4]. As higher order interpolation can be used for the stamp position, it can also be used for each brush vertex's position as well, but the additional fidelity is not generally noticeable or worth the computational cost.

2.6.2 Sweeping

The alternative to the numerical approach of stamping is the analytical approach of sweeping. Computing swept volumes of 3D models is a much-researched problem for applications like robot motion planning [1]. The same problems arise in the case of 2D swept areas. Many approaches for computing sweeps rely on simplifying assumptions about the nature of the problem, such as translation-only motion, or convex polyhedra (polygon) models only.

In the case of a bristle-based brush head model, we can take advantage of such a simplifying assumption. Specifically, computing a sweep is trivial if the object being swept has fewer dimensions than the space in which it is being swept. Consider a 2D square being swept along a path in 3D—the swept volume can be generated by connecting the corners of the square in consecutive positions. Similarly, for a 1D line segment swept along a 2D path, the swept area can be generated by connecting the end points of the line segment at consecutive positions. See Fig. 2.7 for an illustration. We can use this simplification to create sweeps trivially out of the motion of bristles across the canvas. For each link in each bristle, given its position at the previous simulation step and the current one, use a rasterizer such as OpenGL to render a quad defined by the end points of link at the two timesteps. With the same camera setup as described in Sect. 2.6.1, clipping to the deposition threshold will provide the same approximation of canvas contact.

The main limitation of computing the stroke as the swept area of line primitives is that when the bristle is straight (when looking down from above) and the motion

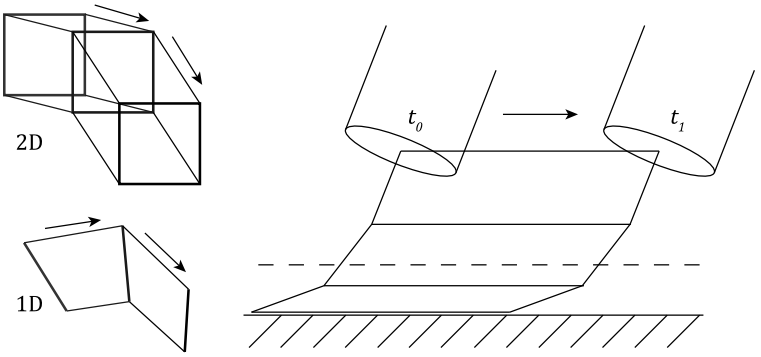


Fig. 2.7 *Left:* Trivial sweeps of a 2D square in 3D, and a 1D line in 2D. *Right:* Since a bristle consists of a set of 1D lines, bristle sweeps can be trivially computed by connecting the vertices of the same bristle at consecutive timesteps

of the bristle is parallel to its contact with the canvas, then the swept area will be zero because a 1D line has zero width. This can be worked around in a number of ways. For example, the bristles could always be made to have a slight curve, or many bristles could be put on the brush at different angles to avoid the case that all the bristles will ever become parallel. Each of these workarounds is an approximation of varying quality, with its own pitfalls that may be significant depending on the specific use case.

Another approach that also supports triangle mesh brush heads is to compute the 2D swept area of a 2D polygon using an algorithm such as that proposed by Pudet [28]. Pudet describes a way to efficiently compute brush strokes generated by sweeping a 2D brush stamp along an input path. An algorithm like this is the basis of the calligraphic brush tool in Adobe Illustrator [2], so it has been demonstrated to be robust. To use Pudet's algorithm, we need the brush footprint represented as a 2D polygon. Given the deformed triangle mesh of the brush head, it is possible to find the intersection between the mesh and the deposition threshold, and to extract the polygon contour of that intersection for use in sweeping. To reduce the number of triangles that need to be considered, only the triangles where two vertices are on opposite sides of the deposition threshold need to be considered.

The downside of creating a swept area from the brush head triangle mesh is that the output of the sweep algorithm is a single contour to which a constant fill gets applied. Therefore, if 50 % gray ink is deposited, the output brush stroke will be a solid 50 % gray, which loses some of the natural media textural quality that is desirable in real brush strokes. To create more variation within the stroke, rather than computing a single sweep from a triangle mesh footprint, re-consider the individual bristle model. Each bristle's contact with the canvas can be considered to be a polygon (e.g. an ellipse for simplicity), dynamically sized based on the bristle thickness and the length of the bristle-canvas contact. Then Pudet's algorithm can be used to compute the swept area of each bristle's contact independently, outputting one flat-filled region per-bristle. The combination of all these swept bristle regions, when given transparent fills, can create the desired variation in shading across the

Fig. 2.8 *Left*: A brush stroke with a single, flat fill. *Right*: One flat fill per bristle adds texture

brush stroke (see Fig. 2.8). This is the algorithm used in Adobe Illustrator's bristle brushes [19].

One of the primary advantages of sweeping is that it can be used, as described above, to generate vector output. The nature of Pudet's algorithm is that it computes the polygonal outline of the swept stroke, and the end-point connecting approach generates triangle vertices as its output. In either case, the final primitives are resolution-independent and can be stored efficiently in vector image formats such as SVG.

2.7 Summary

At this point, we have described the different components necessary to create a physically based virtual brush and to use it to render brush strokes. For each component, we have provided an assessment of the advantages and disadvantages of the available options. Finally, we have summarized the research literature on brush stroke synthesis, so interested readers can pursue a deeper understanding of the material, while implementers interested in a practical understanding of brush stroke synthesis can use this chapter as a guide.

References

1. Abdel-Malek, K., Blackmore, D., Joy, K.: Swept volumes: foundations, perspectives, and applications. Int. J. Shape Model. **12**(1), 87–127 (2006)
2. Adobe: Illustrator (2012). http://www.adobe.com/illustrator/
3. Adobe: Photoshop (2012). http://www.adobe.com/photoshop/
4. Armstrong, J.: Composite Bezier curves (2006). http://www.algorithmist.net/composite.html
5. Bai, B., Wong, K.W., Zhang, Y.: An efficient physically-based model for Chinese brush. In: Proceedings of the International Conference on Frontiers in Algorithmics, pp. 261–270 (2007)
6. Baraff, D., Witkin, A.: Physically based modeling: Principles and practice. In: ACM SIGGRAPH Courses (1997). http://www.cs.cmu.edu/~baraff/sigcourse/
7. Baxter, W., Govindaraju, N.: Simple data-driven modeling of brushes. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 135–142 (2010)
8. Baxter, W., Lin, M.: A versatile interactive 3D brush model. In: Proceedings of the Pacific Conference on Computer Graphics and Applications, pp. 319–328 (2004)
9. Baxter, B., Scheib, V., Lin, M., Manocha, D.D.: Interactive haptic painting with 3D virtual brushes. In: Proceedings of ACM SIGGRAPH, pp. 461–468 (2001)
10. Beeson, C.: Animation in the "Dawn" demo. In: Fernando, R. (ed.) GPU Gems, pp. 223–233. Addison-Wesley, Reading (2004)
11. Boeing, A.: Physics Abstraction Layer (2009). http://pal.sourceforge.net/

12. Boeing, A., Bräunl, T.: Evaluation of real-time physics simulation systems. In: Proceedings of Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pp. 281–288 (2007)
13. Chu, S.H.: Making digital painting organic. Ph.D. thesis, Hong Kong University of Science and Technology (2007)
14. Chu, N., Tai, C.L.: Real-time painting with an expressive virtual Chinese brush. IEEE Comput. Graph. Appl. **24**(5), 76–85 (2004)
15. Chu, N., Baxter, W., Wei, L.Y., Govindaraju, N.: Detail-preserving paint modeling for 3D brushes. In: Proceedings of the International Symposium on Non-photorealistic Animation and Rendering, pp. 27–34 (2010)
16. Corel: Painter (2012). http://www.corel.com/painter/
17. Coumans, E.: Bullet physics library (2010). http://www.bulletphysics.org/
18. Design, A.: ArtRage (2012). http://www.artrage.com/
19. DiVerdi, S., Krishnaswamy, A., Hadap, S.: Industrial-strength painting with a virtual bristle brush. In: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pp. 119–126 (2010)
20. Eastman, P.: Art of illusion (2012). http://www.artofillusion.org/
21. Hertzmann, A.: A survey of stroke-based rendering. IEEE Comput. Graph. Appl. **23**, 70–81 (2003)
22. Kavan, L., Sloan, P.P., O'Sullivan, C.: Fast and efficient skinning of animated meshes. Comput. Graph. Forum **29**(2), 327–336 (2010)
23. Lam, D.: Tokamak physics engine (2010). http://www.tokamakphysics.com/
24. Lewis, J.P., Cordner, M., Fong, N.: Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In: Proceedings of ACM SIGGRAPH, pp. 165–172 (2000)
25. Lu, T.K., Huang, Z.: A GPU-based method for real-time simulation of Eastern painting. In: Proceedings of the International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pp. 111–118 (2007)
26. Mi, X., Xu, J., Tang, M., Dong, J.: The droplet virtual brush for Chinese calligraphic character modeling. In: Proceedings of the IEEE Workshop on Applications of Computer Vision, pp. 330–334 (2002)
27. Okabe, Y., Saito, S., Nakajima, M.: Paintbrush rendering of lines using HMMs. In: Proceedings of the International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia, pp. 91–98 (2005)
28. Pudet, T.: Real time fitting of hand-sketched pressure brushstrokes. Comput. Graph. Forum **13**(3), 205–220 (1994)
29. Saito, S., Nakajima, M.: 3D physics-based brush model for painting. In: Proceedings of ACM SIGGRAPH Conference Abstracts and Applications, p. 226 (1999)
30. Smith, A.R.: Digital paint systems: an anecdotal and historical overview. IEEE Ann. Hist. Comput. **23**, 4–30 (2001)
31. Smith, R.: Open Dynamics Engine (2007). http://www.ode.org/
32. Van Laerhoven, T., Van Reeth, F.: Brush up your painting skills: realistic brush design for interactive painting applications. Vis. Comput. **23**(9), 763–771 (2007)
33. Vandoren, P., Van Laerhoven, T., Claesen, L., Taelman, J., Raymaekers, C., Van Reeth, F.: IntuPaint: bridging the gap between physical and digital painting. In: IEEE International Workshop on Horizontal Interactive Human Computer Systems, pp. 65–72 (2008)
34. Vandoren, P., Claesen, L., Van Laerhoven, T., Taelman, J., Van Reeth, F.: FluidPaint: an interactive digital painting system using real wet brushes. In: Proceedings of the IEEE International Workshop on Tabletops and Interactive Surfaces (2009)
35. Xie, N., Laga, H., Saito, S., Nakajima, M.: IR2s: interactive real photo to Sumi-e. In: Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering, pp. 63–71 (2010)
36. Xu, S., Tang, M., Lau, F., Pan, Y.: Virtual hairy brush for painterly rendering. Graph. Models **66**(5), 263–302 (2004)

http://www.springer.com/978-1-4471-4518-9

Image and Video-Based Artistic Stylisation
Rosin, P.; Collomosse, J. (Eds.)
2013, XII, 397 p., Hardcover
ISBN: 978-1-4471-4518-9