Interactive Modeling of Support-free Shapes for Fabrication

Tim Reiner and Sylvain Lefebvre
INRIA

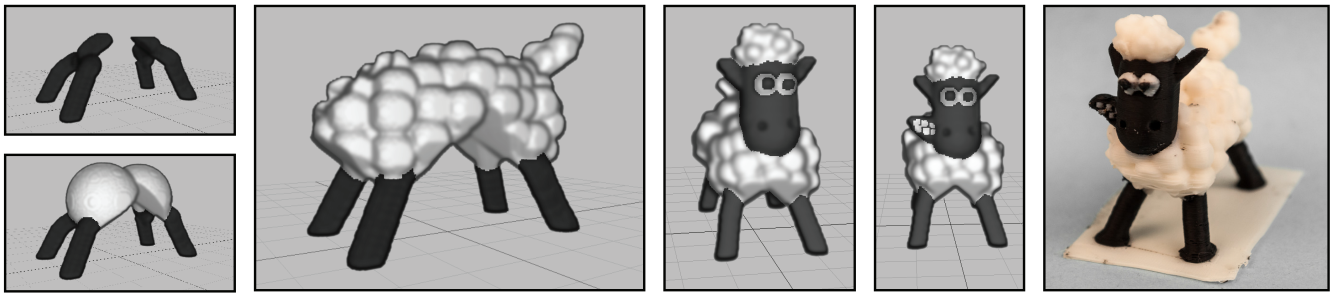


Figure 1: *Interactive modeling session for the* SHAUN *model. The user starts by sculpting four base legs and then attaches two spheres to form the body. Note how the spheres get trimmed to produce a support-free arch. The user then applies little spheres to model the wool and finally sculpts the head.* Right: *3D printed result. During modeling the shape always stays support-free.*

Abstract
We introduce an interactive sculpting approach that enables modeling of support-free objects: objects which do not require any support structures during 3D printing. We propose three operators – trim, preserve, grow – to maintain the support-free property during interactive modeling. These operators let us define brushes that perform either in an unconstrained manner (adapting the shape to the brush effect), or selectively discard changes inside the brush volume. Our technique can be applied to many modeling operations and we demonstrate it on brushes for adding or removing matter. We describe an efficient implementation of a voxel-based modeling tool that produces only support-free shapes, and show example shapes modeled within minutes.

1. Introduction and Previous Work

Interactive modeling and sculpting is an accessible and fun way to create 3D shapes. Many users would like to go one step further and 3D print their objects, but unfortunately this may turn into a frustrating experience as most shapes cannot print without temporary support structures. This is in particular the case on filament printers, which are preferred by home users due to their low hardware complexity and low material cost (PLA and ABS thermoplastics in particular). Unfortunately, shapes having overhangs require support structures to hold the filament that would otherwise fall. Support structures are removed manually after printing, a long and tedious operation that can lead to breaking the object if not done carefully.

Recently there has been a strong effort to improve support techniques [SU14, VGB14, DHL14]. Hu et al. [HJW15] go further by deforming the model to reduce supports. Supports are crucial in a general setting to allow any shape to be fabricated. Yet there exists a class of *support-free* shapes that print much easier and without supports: Hu et al. [HLZCO14] and Herholz et al. [HMA15] propose to locally approximate shapes by pyramids or height fields; both print easily. Hornus et al. [HLDC] give a generic definition of a support-free shape with algorithms to create minimally fitting envelopes and maximum inner carvings. We follow a similar analysis here.

The sculpting metaphor has been thoroughly explored from the seminal work of Galyean and Hughes [GH91] to advanced software used by the movie and video game industries, e.g., ZBrush. It is accepted as one of the most accessible techniques, as witnessed by the emergence of virtual clay software targeted at non-experts, such as Cubify Sculpt or VRClay. We follow this natural way of modeling and propose an interactive tool based on brushes for adding, removing, or locally modifying matter.

These brushes always preserve the support-free property. Starting from a shape that can be printed without support, a modeling session always leads to a new shape which is again support-free. In a standard modeling tool, adding or removing matter can lead to parts which are no longer printable, such as overhangs or islands. To avoid these issues, either the brush has to be constrained to discard some changes, or the shape has to be modified to adapt to the changes made by the brush. We propose both options: using constrained brushes, the user knows that he will not damage the shape around the brush; with unconstrained brushes the user has full sculpting freedom and the shape around adapts to enforce printability.

We introduce four basic brushes: *add-trim, remove-preserve, add-grow, remove-trim*. The first two are constrained, while the remaining two are unconstrained. The methods *trim*, *preserve*, and *grow* observe printability constraints during brushing. These methods are general and can be applied to any modeling operation, which we demonstrate by implementing a *smooth-trim* brush later. We implemented our entire voxel-based system on the GPU, performing all operations at interactive frame rates, and demonstrate our approach with a variety of 3D printed models.

2. Our Approach

We model the shape as a voxel grid V of resolution N^3. This is both a natural representation for sculpting algorithms [FCG99] and for expressing fabrication constraints. A voxel is indexed as $V(i,j,k)$ where k aligns with the vertical z-axis (build direction). Each voxel $V(i,j,k)$ stores a density (1: solid, 0: empty) and a material ID.

Before fabrication the shape is *sliced* by intersecting it with a set of uniformly spaced planes [DGLC15]. Each slice contains a set of contours delimiting inner areas that will be fabricated as a physical layer. Layers are fabricated one after the other, from bottom to top. The layer above will bond to the layer below, progressively forming a solid object. For the sake of simplicity we align the slices with the voxel grid, and obtain slice S_k as the plane of voxels

$$S_k = \{V(i,j,k) \mid V_{\text{density}}^{(i,j,k)} = 1,\ \forall (i,j) \in N^2\}.$$

2.1. Fabrication Constraints

The two major restrictions to consider are overhangs and islands. They appear where the slope of downward faces is too large, preventing material to properly adhere to the layer below. Islands are disconnected regions, typically protrusions with a downward angle. On filament printers they lead to failed prints with dangling material: the filament being deposited is not supported from below and either fails to properly bond or falls by gravity. On resin printers similar defects appear: disconnected regions end up floating in the resin tank and bond at random places on the object, while overhang areas can distort when the print is pulled away from the fabrication surface.

Hornus et al. [HLDC] showed that both overhangs and islands can be detected through morphology operations on the object slices. Indeed, when considering slices, the overhang constraint translates to verifying whether the slice S_{k+1} is entirely included within a dilation of slice S_k by a disk that captures the maximum allowed overhang, a condition we denote as:

$$S_{k+1} \subseteq S_k \oplus \mathcal{B}_r\ , \tag{1}$$

where $\oplus$ denotes the dilation operator ($\ominus$ the erosion), $\mathcal{B}_r$ is a discrete disk of radius $r = h \cdot \tan\theta$, h being the layer thickness, and θ the max overhang angle. We use $h = 0.25$ mm, $\theta = 45^\circ$ and perform all operations in the discretized voxel setting. Intuitively, $\mathcal{B}_r$ indicates how far S_k can grow without producing an excessive overhang, while $S_k \oplus \mathcal{B}_r$ denotes the largest area that can be supported in S_{k+1}. Islands directly contradict Equation (1) since they only exist in S_{k+1}.

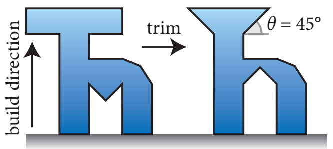
A support-free shape has to enforce Equation (1) everywhere. Areas violating the constraint in slice S_{k+1} are easily detected as $S_{k+1} \setminus (S_k \oplus \mathcal{B}_r)$. (Note that we disregard the ability of filament printers to print straight bridges within a single layer.)

We use Equation (1) to determine whether each voxel is *supported*. A voxel in slice S_{k+1} is supported if and only if $V(i,j,k+1) \in S_k \oplus \mathcal{B}_r$. Ground voxels $V(i,j,0)$ are always supported.

2.2. Brushes (Overview)

The four main brushes *add-trim*, *remove-preserve*, *add-grow*, and *remove-trim* are unrestricted in their shape and size. Two add matter and two remove matter. There are two ways of preserving fabricability: either by constraining changes within the brush (add-trim, remove-preserve), or by modifying the shape around the brush (add-grow, remove-trim). We now introduce three operators that recover from overhangs and islands: *trim*, *preserve*, and *grow*.

Trim (add-trim, remove-trim). Trimming removes any problematic part from the shape, such as floaters, islands, and overhangs. We perform a bottom-top sweep, removing unsupported voxels within all slices. This operation removes all unsupported voxels; it cascades from bottom to top and thus can have a global impact on the shape.



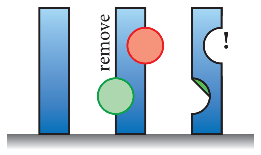
Add-trim: this brush adds matter inside the brush volume, but only the support-free subset. (This is equivalent to filling the entire brush and then calling the trim operation.) It is useful to add details while never creating overhangs or islands. It never affects the surrounding shape and the effect remains localized inside the brush volume.

Remove-trim: this brush deletes all matter inside the brush and then calls a global trim operation. The effect propagates upwards in cascade during the sweep, if previously supported voxels have been turned into unsupported voxels. Eventually only the support-free subset of the shape remains.

Preserve (remove-preserve). The previous *remove-trim* brush can lead to undesirably large deletions. We therefore propose another removal brush that attempts to *preserve* the shape above the brush. Particularly we want to preserve voxels inside the brush volume necessary for support above.

To achieve this, we consider the minimal necessary support for a slice S_k. Let U_{k-1} be the smallest surface required at slice $k-1$ to support S_k, that is $S_k \subseteq U_{k-1} \oplus \mathcal{B}_r$ (Equation (1)).

Let us assume for now that $S_k = (S_k \ominus \mathcal{B}_r) \oplus \mathcal{B}_r$, that is S_k is invariant by morphological opening. Under this assumption it follows from Equation (1) that $U_{k-1} = S_k \ominus \mathcal{B}_r$. It is minimal since removing any voxel would produce a surface that cannot support S_k, as $S_k = U_{k-1} \oplus \mathcal{B}_r$.



This remark leads us to a simple algorithm to preserve voxels necessary for support. First delete all voxels within the brush volume. Then sweep down from the slice just above the brush to the slice just below, replacing in sequence each slice by $S_k = S_k \cup U_k$ with $U_k = S_{k+1} \ominus \mathcal{B}_r$. This guarantees that the minimal set of necessary voxels is reintroduced, *under the assumption that* $S_k = (S_k \ominus \mathcal{B}_r) \oplus \mathcal{B}_r$. Note how in the example image a part of the green brush has been preserved.

Unfortunately the assumption breaks in two notable cases. The first case is due to spurious voxels that are not included in the opening of S_k. These voxels correspond to "corners" that cannot be captured by $\mathcal{B}_r$. The second case is more problematic and is due to small islands: surfaces smaller than $\mathcal{B}_r$ in S_k. While we experimented several possible approaches, we find that for modeling purposes the simplest technique is to trim these voxels.

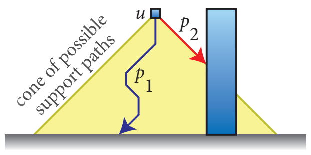
Grow (add-grow). This brush allows to paint in free space while growing the shape to ensure the newly added matter is printable.

For this purpose we define a *grow* operation. It is akin to support techniques, where a temporary structure is built to support the shape during fabrication. However, we seek to define a *modeling* operation that will modify the shape to embed the support inside its geometry. The requirements are very different. Support structures are optimized to use little material, to print fast and to snap easily. Instead we define an operation that behaves in a predictable manner, connects seamlessly to the existing shape and feels natural to the user. In addition, while support generation techniques have access to the final object, our technique has to discover the shape incrementally as it is gradually modeled by the user.

Our key intuition is that the shape should grow in a minimal way, avoiding overhangs and islands while adding matter which naturally connects to the existing shape. Let $u = V(i, j, k)$ be a single unsupported voxel in slice S_k. Consider a path $V(i_0, j_0, k_0), \dots, V(i_p, j_p, k_p), \dots, u$ of *empty* voxels that is monotonically growing, i.e., $k_{p+1} = k_p + 1$. Such a path is *supporting* iff:

1. $k_0 = 0 \quad or \quad V(i_0, j_0, k_0) \in S_{k_0-1} \oplus \mathcal{B}_r$,
2. $V(i_p, j_p, k_p) \notin S_{k_{p-1}} \oplus \mathcal{B}_r$,
3. $V(i_p, j_p, k_p) \in \{V(i_{p-1}, j_{p-1}, k_{p-1})\} \oplus \mathcal{B}_r$,

where 1) states that the first voxel is supported by the slice below, 2) states that other voxels are not supported by the shape, but 3) would be supported by their voxels below in the path if they were solid.



Any such supporting path can be added to the shape to support voxel u. There is a large number of possibilities: all possible support paths lie within a cone of θ slope with its tip on u. The choice is critical as it defines *how* the shape grows during modeling.

We propose to consider *closest supporting paths*: supporting paths that remain as close as possible to the existing surface. The cost of a path is defined as $C(v_0, \dots, v_k) = \sum_i (D(v_i))$ where $D(v)$ is the distance between the empty voxel v and its closest filled voxel within the slice plane. The rationale is to reconnect with the shape while staying as close as possible to existing surfaces. The closest supporting paths achieve this by minimizing the distance between the path and the existing shape. In practice, computing D for every slice is expensive. We reformulate the cost to take only the distance on the ground plane into account: $C(v_0, \dots, v_k) = k + D_{\text{ground}}(v_0)$. With this cost definition the closest supporting paths can be computed very efficiently by a bottom-top sweep through V. This achieves a good compromise during interactive modeling.

The brush generates surfaces by connecting new matter *as close as possible* to the existing shape. It is therefore predictable and quite natural to use after a few minutes. It lets the user freely paint without being stopped mid-air by the constraint enforcement.

Generic Brushes. The trim, preserve, and grow operators can be used in combination with other modeling operations. As an example, we have implemented a *smooth-trim brush* which sets a voxel's density according to an average of its neighborhood. Outliers and isolated voxels are removed by subsequent trimming.

3. Implementation

We implemented our modeling tool using OpenGL 4.5 and make extensive use of compute shaders for all modeling operations. The entire modeling space spans 512^3 voxels, which allows us to sculpt fairly large objects in the context of printing: using a layer thickness of 0.25 mm every dimension can be up to 12.8 cm in size.

We allocate 3D textures for the voxel space, the shape of the brush, and a shortest path map for the add-grow brush. We use two channels (red and green) for the voxel space texture to store density and material ID separately. On this voxel space we dispatch compute shaders for every modeling operation; we do not keep an additional copy of the shape in system memory.

3.1. Rendering and Shading

We render directly by raymarching efficiently through the voxel space using a 3D Digital Differential Analyzer [AW87]. We use a deferred shading approach to render the scene: a framebuffer object (FBO) holds textures to store geometry (G-buffer), material ID, surface normals, and depth information. We use screen-space techniques to achieve smooth rendering, which decouples shading from scene complexity.

3.2. Brushes (Implementation)

We consider $\theta = 45^\circ$ and set $\mathcal{B}_r$ to be a 3×3 structuring element. Performing a dilation on a voxel then leads to filling its eight surrounding neighbors within the same slice. Eroding a slice will retain only voxels with a full neighborhood. With these morphological operations we can now implement the *trim*, *preserve*, and *grow* operations for our brushes.

A *global trim operation* (entire voxel space) can be implemented by sweeping from bottom to top (layer 0 is always supported):

```
for ($int\ layer = 1;\ layer < RESOLUTION;\ layer{+}{+}$) do
    trimShaderProgram->setUniformValue("layer", layer);
    glDispatchCompute(RESOLUTION/32, RESOLUTION/32, 1);
```

The shader removes all voxels without support from below:

```
layout(rg8ui, binding=0) uniform uimage3D voxelTex;
uniform int layer;
layout(local_size_x=32, local_size_y=32, local_size_z=1) in;
void main(void) {
pos = (gl_GlobalInvocationID.x, gl_GlobalInvocationID.y, layer);
if (    ⋀       imageLoad(voxelTex, pos.x+i, pos.y+j, pos.z-1).x == 0)
  i,j∈{−1,0,1}  { imageStore(voxelTex, pos, ivec4(0, 0, 0, 0)); }}
```

We implemented the *add-trim brush* by dispatching a shader inside the brush volume and then adding only voxels with support from below (no subsequent trim operation required):

```
pos = brush_position + gl_GlobalInvocationID; m = materialID;
if (    ⋁       imageLoad(voxelTex, pos.x+i, pos.y+j, pos.z-1).x == 1)
  i,j∈{−1,0,1}  { imageStore(voxelTex, pos, ivec4(1, m, 0, 0)); }
```

The *remove-preserve brush* is implemented by dispatching a shader inside the brush volume, layer by layer, from *top to bottom*, and then dilating empty space from the layers above:

```
pos = brush_position + gl_GlobalInvocationID;
if (    ⋁       imageLoad(voxelTex, pos.x+i, pos.y+j, pos.z+1).x == 0)
  i,j∈{−1,0,1}  { imageStore(voxelTex, pos, ivec4(0, 0, 0, 0)); }
```

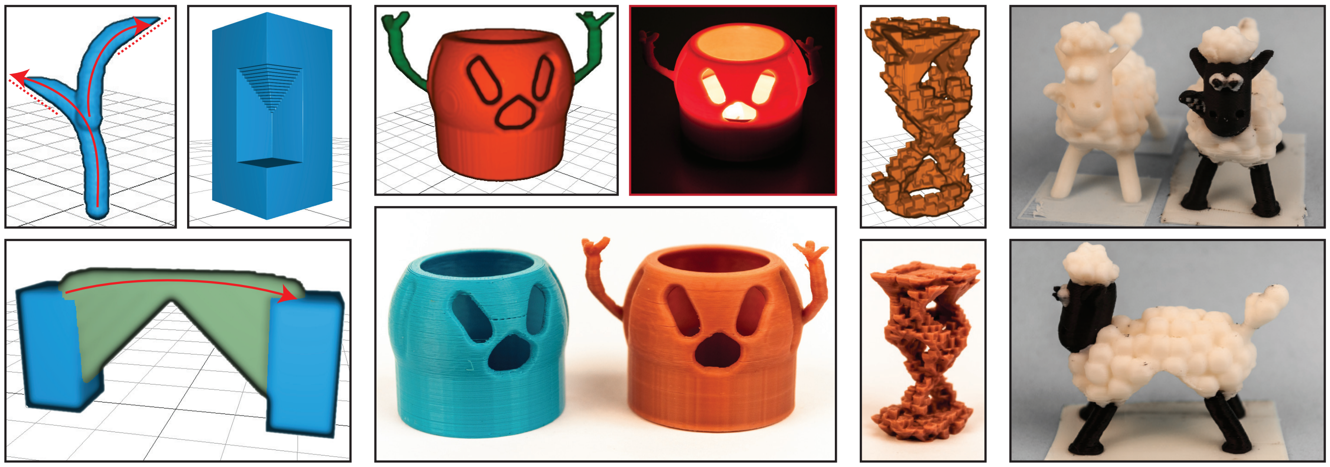


Figure 2: Left: *examples of an add-trim stroke, a remove-preserve operation where a cube is subtracted from a pillar, and an add-grow stroke which automatically creates a bridge underneath.* Center: *the* PUMPKIN *model can be used as a tea-light holder.* Center Right: *a more intricate sculpture.* Right: *more prints of the* SHAUN *model. All shapes here print support-free.*

The *grow operation* for the *add-grow brush* is more involved. We allocate a 3D path map that spans the entire voxel space. First, for every voxel on the ground plane, we compute the distance to the closest filled voxel on the ground plane and store this information in the ground layer of the path map. Next, we sweep through the voxel space from bottom to top, propagating information inside the path map. For every voxel v in layer $k+1$ we look at its 3×3 neighborhood in the layer k below and detect voxel u which stores the minimal path cost. We increase this cost by one and store it for v. In a second channel of the path map, we encode the direction how to navigate from v back to u; there are nine possible directions.

The user starts sketching a stroke path S in free space while we map the mouse cursor position to the corresponding voxel on the sketch plane. During sketching the amount of voxels in S is monotonically increasing. For every new voxel v_i added to S we dispatch a compute shader which then navigates voxel by voxel from v_i through the path map until it reaches the surface or the ground. This forms a new supporting path P. We emit the brush centered at every voxel $p_i \in P$. This ensures that all added matter along S is always supported (see Section 2, *grow*). We update the path map every time the user releases the mouse button after sketching.

4. Results and Conclusion

We developed and tested our modeling tool on a system with an Intel Core i7-4790K CPU and an NVIDIA GeForce GTX 980 graphics card. We are well able to achieve interactive frame rates for all modeling operations at a Full HD resolution of 1920×1080.

Figure 1 depicts an interactive modeling session for the SHAUN model. Modeling this shape without prior practice runs took half an hour, including several *undo* operations. A user typically starts modeling from the ground. For SHAUN, we started forming the legs and then the main body part before finishing with the head. The final print is almost 6 cm tall and roughly 7 cm long. Figure 2 shows more results, including examples how we implemented the brush operations. The PUMPKIN model was printed on an Ultimaker 2 in under two hours. The intricate arms do not require support structures. We also printed a more detailed and intertwined sculpture.

Overall, we found that interactivity provides a great modeling experience, while support-free shapes greatly simplify printing. We would now like to investigate how to use brushes to paint with structure, i.e., adding visual complexity using textures [RCM*14] and support-free geometric brush details to affect surface characteristics.

This work was supported by ERC grant ShapeForge (StG-2012-307877). We would like to thank our entire research group for various help with printing!

References

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *Proceeding of Eurographics 1987* (1987), pp. 3–10. 3

[DGLC15] DINH H. Q., GELMAN F., LEFEBVRE S., CLAUX F.: Modeling and toolpath generation for consumer-level 3D printing. In *ACM SIGGRAPH 2015 Courses* (2015), pp. 17:1–17:273. 2

[DHL14] DUMAS J., HERGEL J., LEFEBVRE S.: Bridging the gap: Automated steady scaffoldings for 3D printing. *ACM Transactions on Graphics 33*, 4 (July 2014), 98:1–98:10. 1

[FCG99] FERLEY E., CANI M.-P., GASCUEL J.-D.: Practical volumetric sculpting. In *Implicit Surfaces* (1999). 2

[GH91] GALYEAN T., HUGHES J.: Sculpting: An interactive volumetric modeling technique. *Proc. SIGGRAPH 25*, 4 (1991), 267–274. 1

[HJW15] HU K., JIN S., WANG C.: Support slimming for single material based additive manufacturing. *Computer-Aided Design 65* (2015). 1

[HLDC] HORNUS S., LEFEBVRE S., DUMAS J., CLAUX F.: *Tight printable enclosures for additive manufacturing.* Inria report, 2015. 1, 2

[HLZCO14] HU R., LI H., ZHANG H., COHEN-OR D.: Approximate pyramidal shape decomposition. *ACM Trans. Graph. 33*, 6 (2014). 1

[HMA15] HERHOLZ P., MATUSIK W., ALEXA M.: Approximating free-form geometry with height fields for manufacturing. *Computer Graphics Forum 34*, 2 (2015), 239–251. 1

[RCM*14] REINER T., CARR N., MECH R., STAVA O., DACHSBACHER C., MILLER G.: Dual-color mixing for fused deposition modeling printers. *Computer Graphics Forum 33*, 2 (2014), 479–486. 4

[SU14] SCHMIDT R., UMETANI N.: Branching support structures for 3D printing. *ACM SIGGRAPH 2014 Talks Program* (2014). 1

[VGB14] VANEK J., GALICIA J., BENES B.: Clever support: Efficient support structure generation for digital fabrication. *Computer Graphics Forum 33*, 5 (2014), 117–125. 1