Vertex Shader Domain Warping with Automatic Differentiation

Dave Pagurek van Mossel
Butter Creatives

May 12, 2024

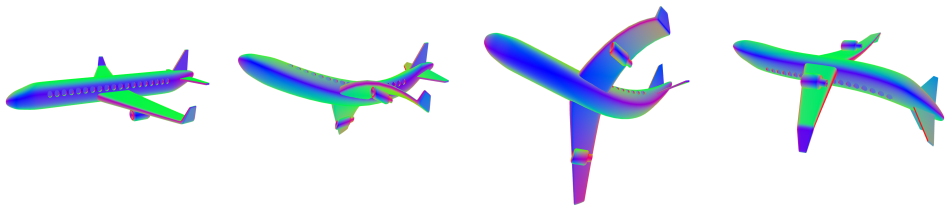


Figure 1: An airplane model undergoing an animated twist warp. Its vertex positions and normals are updated in the vertex shader, with the fragment shader visualizing the absolute normal direction as a color. After applying the warp, the output normals are accurate.

Abstract

Domain warping is a technique commonly used in creative coding to distort graphics and add visual interest to a work. The approach has the potential to be used in 3D art as mesh vertices can be efficiently warped using a vertex shader in a WebGL pipeline. However, 3D models packaged for the web typically come with baked-in normal vectors, and these need to be updated when vertex positions change for lighting calculations to work. This is typically done via finite differences, which requires parameter tuning to achieve optimal visual fidelity. We present a method for 3D domain warping that works with automatic differentiation, allowing exact normals to be used without any tuning while still benefiting from hardware acceleration.

1 Introduction

Creative coding is an art form that uses the computer and source code as an expressive medium. The process of writing creative code is often characterized by exploration and iteration (projects made in Processing [11] are called "sketches" for this reason [5]) and thus many commonly employed techniques are ones with easily tweakable parameters that can generate a wide variety of visual results.

One such technique is *domain warping*, used often in 2D and some 3D contexts [13]. Given a function $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ that defines an offset for any point in space, using the warp function $w(\vec{x}) = \vec{x} + f(\vec{x})$, one can warp any function whose domain is $\mathbb{R}^n$ by giving it $w(\vec{x})$ as input instead of $\vec{x}$. It provides a framework that invites exploration, as any offset function can be used, producing varied, interesting results from combinations of simple mathematical building blocks. While some carefully tuned offset functions may have physical motivations, such as the wave equations of Dynamic Kelvinlets [3], this is not a requirement. Arbitrary functions are useful tools in the creation of surreal or abstract art, a common style in creative coding [4, 9, 15].

This technique has the potential to fit nicely into a 3D workflow on the web: to warp the domain of a 3D

mesh, one can implement an offset function in a vertex shader, which is the step in the pipeline responsible for transforming vertex positions into screen space before rasterization. Running on the GPU, this is likely the most efficient method of domain warping a triangle mesh. Unfortunately, meshes packaged for a WebGL pipeline typically bake in a normal vector for each vertex. If one applies domain warping to vertex positions without updating the baked normals, they will no longer be perpendicular to the surface of the mesh (visualized in Figure 2b) and will cause later lighting calculations in the fragment shader to be inaccurate. Using finite differences to approximate updated normals requires parameter tuning for each use to avoid visual artifacts, making it difficult to use in a general system accepting arbitrary meshes and offset functions. We present an algorithm that leverages automatic differentiation to generate an updated normal in any circumstance without parameter tuning, enabling 3D domain warping to be used more easily in web-based creative code environments.

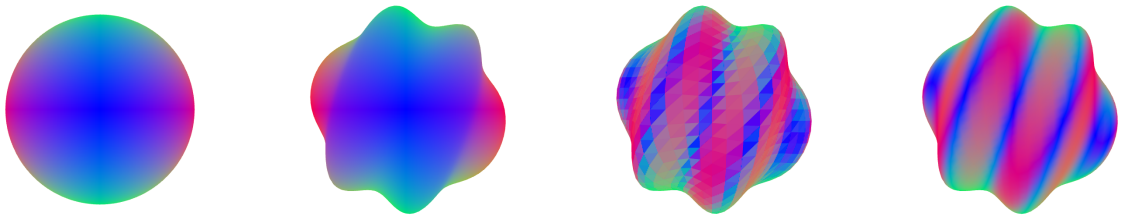
2 Background

Domain warping. Domain warping in 3D is a common technique used when raymarching signed distance functions (SDFs) [13], found often in the demoscene and on platforms like Shadertoy [8]. Unlike triangle meshes, SDFs do not have baked normals that need updating: models are represented by a function $f : \mathbb{R}^3 \mapsto \mathbb{R}$ describing, in theory, the distance to the surface of the shape at a given point in space, and in practice, a conservative estimate of said distance. Shapes are then rendered by marching (or "sphere tracing") light rays through the scene, detecting intersections using the scene SDF [7]. While domain warping an inexact SDF can be as simple as function composition, one cannot make use of existing triangle-based 3D models without a nontrivial conversion that comes with performance tradeoffs [1].

Perturbing normals. In a mesh-based WebGL pipeline, perturbations that update normals are used at small scales to add extra texture to the surface of a model. Traditionally, this technique is called *bump mapping* and involves parameterizing the surface of a model into two axes, u and v, and providing a bump height $h : \mathbb{R}^2 \mapsto \mathbb{R}$ defined for each parameter value [2]. The bump map suggests a new surface, where each point is moved along its normal by the height defined by the bump function, yielding $\vec{p'} = \vec{p} + h([u_p \quad v_p]^T)\hat{n}$. One can then compute a perturbed normal $\hat{n}' = \frac{\partial \vec{p'}}{\partial u} \times \frac{\partial \vec{p'}}{\partial v}$. This formulation does not depend on a specific parameterization of the surface, so bump maps can be applied in a fragment shader by using a local screen-space parameterization [10], relying on derivatives with respect to neighboring pixels (`dFdx(position)` and `dFdy(position)` in GLSL.)

Bump mapping intentionally only affects surface lighting and not the geometry, and given this small scale, only handles 1D height maps. If one wants to apply similar techniques at larger scales using 3D offsets, updated formulae are required. At larger scales, the vertices must be updated in the vertex shader, where one does not have screen-space derivatives available. If one were to recalculate normals using screen-space derivatives in the fragment shader based on the updated vertex positions, the linear interpolation between vertices becomes apparent, making normals appear faceted instead of smooth, shown in Figure 2c.

Finite differences. SDF raymarching code typically calculates normals via finite differences, approximating the normal ($\nabla f(\vec{p})$ when $\vec{p}$ is on the surface) with, using the X axis as an example, $\frac{1}{h}\left(f(\vec{p} + [h\ 0\ 0]^T) - f(\vec{p})\right)$ for a small value h. While this method of calculating normals can also work in a mesh-based WebGL pipeline, it requires simple but tedious tuning of h to work well. To paint a picture of what picking h involves, its value needs to be small enough to not lose detail, but not so small that numerical precision issues become visible. Since SDF raymarching is done per pixel, to be as large as possible without losing visual detail, a recommended heuristic for picking h is to try to make its footprint in screen space approximately one pixel: small enough that more accurate normals would not be visible, and large enough to prevent aliasing. [14]



(a) Input

(b) Warp without adjusting normals

(c) Warp with normals from screen-space derivatives

(d) Warp affecting vertices and normals

Figure 2: When applying a sine wave warp to the vertices of a sphere (a), the baked normals will be incorrect (b) unless they are updated to account for the warp. Relying on screen-space derivatives of position yields faceted normals (c), while computing normals in the vertex shader allows for smooth normals (d).

In the demoscene community, which produced many contemporary SDF techniques, file size constraints are common (including the entire genre of 64K intros), so fine-tuning of parameters is a common and acceptable tradeoff for smaller code. Our method, aimed instead at an audience interested in easy experimentation and expression, sidesteps this with a setup that leverages exact derivatives. Comparatively, normals achieved through automatic differentiation will be more accurate while having equivalent computational cost [6], motivating our approach and its use of automatic differentiation instead of approximation.

3 Method

The artist provides an offset function $f : \mathbb{R}^3 \mapsto \mathbb{R}^3$ defining, for an input point in 3D space, a 3D offset to add to its location. We assume f is continuous and differentiable. Each vertex position $\vec{p}$ on the mesh M will be mapped to an updated position $\vec{p'} = w(\vec{p}) = \vec{p} + f(\vec{p})$.

Each $\vec{p}$ from the input mesh comes with a corresponding baked normal $\hat{n}$. We are solving for the vector $\hat{n}'$, representing the surface normal of the deformed mesh M' at point $\vec{p'}$. This must be done in a way that exact derivatives that can be feasibly calculated in a shader, with only knowledge of per-vertex properties and not the whole mesh.

Our method achieves these goals in the following steps:

1. We express derivatives of the warp function given a local tangent and bitangent, which we then use to generate a warped normal. Importantly, the structure of the derivative formula depends only on the warp function and accepts *any* local parameterization, enabling one warp shader to work on all meshes.

2. We provide a formula for a surface tangent and bitangent given only the surface normal, allowing inputs to Step 1 to be generated in a vertex shader with limited knowledge of the mesh.

3. We describe a system to use automatic differentiation to statically generate a shader with the derivatives required for Step 1.

3.1 Updated Normals

Assume one has two unique vectors $\hat{u}$ and $\hat{v}$ tangent to the surface of the input mesh M at p such that $\hat{u} \times \hat{v} = \hat{n}$. One can approximate $\hat{n}'$ by finding the result of w on points near $\vec{p}$, shifted slightly along a linear approximation of the surface at $\vec{p}$ by a small

value h:

$$\hat{n}' \approx \text{normalize}\left(\frac{w(\vec{p}+h\hat{u}) - w(\vec{p})}{h} \times \frac{w(\vec{p}+h\hat{v}) - w(\vec{p})}{h}\right)$$

Taking the limit as $h \to 0$, this is equivalent to taking directional derivatives of $w(\vec{p})$ in the directions $\hat{u}$ and $\hat{v}$:

$$\begin{aligned}\hat{n}' &= \text{normalize}\left(\frac{\partial w(\vec{p})}{\partial \hat{u}} \times \frac{\partial w(\vec{p})}{\partial \hat{v}}\right) \\ &= \text{normalize}\left(\frac{\partial(\vec{p} + f(\vec{p}))}{\partial \hat{u}} \times \frac{\partial(\vec{p} + f(\vec{p}))}{\partial \hat{v}}\right)\end{aligned}$$

Since, by definition, $\hat{u}$ and $\hat{v}$ are tangent to the surface at $\vec{p}$, the directional derivative of $\vec{p}$ in the direction of $\hat{u}$ or $\hat{v}$ will be $\hat{u}$ or $\hat{v}$ itself, respectively:

$$\hat{n}' = \text{normalize}\left(\left(\hat{u} + \frac{\partial f(\vec{p})}{\partial \hat{u}}\right) \times \left(\hat{v} + \frac{\partial f(\vec{p})}{\partial \hat{v}}\right)\right)$$

Since f is defined in terms of Cartesian coordinates and not the coordinate space defined by $\hat{u}$ and $\hat{v}$, we reframe the above expression to be in terms of the Jacobian of the offset, $J = \left[\frac{\partial f(\vec{p'})}{\partial x} \;\; \frac{\partial f(\vec{p'})}{\partial y} \;\; \frac{\partial f(\vec{p'})}{\partial z}\right]$:

$$\hat{n}' = \text{normalize}\left((\hat{u} + J\hat{u}) \times (\hat{v} + J\hat{v})\right) \tag{1}$$

To define J, we calculate all the partial first-order derivatives of f, which will be implemented using automatic differentiation, described later in Section 3.3. Importantly, this is independent of the choice of tangent vectors: whichever ones we pick, the formula for J does not change, so we do not need to run automatic differentiation at runtime. It is sufficient to run it once at compile time, and our runtime choice of tangent vectors only adds a matrix multiplication with J.

3.2 Picking Tangent Vectors

Without knowledge of the rest of the mesh vertices and their connectivity, we can make assumptions about the local surface around $\vec{p}$. The normal $\hat{n}$ implies that there is a surface plane S passing through $\vec{p}$ and normal to $\hat{n}$. We can parameterize this plane into a tangent $\hat{u}$ and bitangent $\hat{v}$ by finding any vector $\hat{w}$ not equal to $\pm\hat{n}$, taking the cross product between it and $\hat{n}$ to get one vector parallel to S, and then taking the cross product between that and $\hat{n}$ to get another, different vector parallel to S:

$$\hat{w} = \begin{cases}[0 \;\; 1 \;\; 0]^T, & \hat{n} = [\pm 1 \;\; 0 \;\; 0]^T \\ [1 \;\; 0 \;\; 0]^T & \text{otherwise}\end{cases}$$

$$\hat{v} = \text{normalize}(\hat{v} \times \hat{n}) \tag{2}$$

$$\hat{u} = \hat{v} \times \hat{n} \tag{3}$$

Since $|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}| \sin\theta$, to get a *unit* tangent vector from a cross product, one must normalize the result if the two inputs are not orthogonal. Since $\hat{w}$ and $\hat{v}$ are guaranteed by construction to not be parallel but may not be orthogonal, we must normalize the result of their cross product in Equation 2 to get $\hat{v}$.

By combining Equations 2 and 3 with Equation 1, we can fully compute a new $\hat{n}$ that can be used for lighting.

3.3 Automatic Differentiation

The derivatives in Section 3.1 are being used to compute the Jacobian of the offset function $\in \mathbb{R}^{3\times3}$ based on an input position $\in \mathbb{R}^3$. Given that the outputs are greater in number than the inputs, we opt for forward-mode automatic differentiation.

The derivatives will be used in a shader, which has limited language capabilities and resources. Rather than creating structs in GLSL to create dual numbers and defining GLSL functions to operate on them, we instead differentiate at compile time. We express the offset function in a host language outside of the shader, and from the host language, generate GLSL source code for the shader that computes both the offset and its derivatives. For each operation in the computation graph, we can generate one line of GLSL to compute and store the output of the computation, plus an additional line storing the derivative of that graph node with respect to each independent variable. In practice, we output more condensed code, as we are able to compact multiple computation graph nodes up to a specified depth before outputting GLSL, as we find this easier to inspect and

debug. Listing 1 shows an example of this, with and without condensed output.

4 Implementation

Targeting the creative coding community, our implementation comes in the form of the library *p5.warp*: a plugin for the web graphics library p5.js [12] written in JavaScript and GLSL. Artists specify an offset function f in Javascript through our API, a shader generator using the builder design pattern. The library provides builder methods corresponding to mathematical operators and functions, allowing a computation graph to be built with concise syntax. The builder implements static forward-mode automatic differentiation by outputting GLSL code containing expressions that compute both f and its Jacobian J.

The artist does not need to write any GLSL and only needs to interact with our builder API. Artists are free to use loops, make temporary variables, or create and call functions while interacting with the builder. These JavaScript constructs act as macros with respect to the generated shader code, evaluated at the shader's compile time to generate static GLSL. The API also provides access to inputs such as the current time in milliseconds, the mouse position, or the canvas size, which get turned into shader uniforms. Artists may define a warp in either model or world space. Listing 2 shows an example of a twist warp that animates back and forth over time, using the `millis` uniform to access time and using a JavaScript function as a macro.

The output of the builder is spliced into a vertex shader, where the normal-updating math from Section 3 is implemented. The main subroutine of the shader for a warp function defined in model space is shown in Listing 3.

5 Results

Visuals. Figure 1 shows frames of an animated warp applied to an airplane model, giving it a twisting motion with cartoon squash and stretch. Exam-

Listing 1: Generated GLSL code for the Jacobian of the offset function $f([x \;\; y \;\; z]^T) = [0.5\sin(0.005t + 2y) \;\; 0 \;\; 0]^T$, with and without condensing the number of intermediate variables in the output. The automatic output has been manually formatted and intermediate variables have been manually renamed for readability.

```
// One node per variable:
float v1 = millis * 0.005;
float v2 = position.y * 2.0;
float d_v2_by_d_y = 2.0 * 1.0;
float v3 = v1 + v2;
float d_v3_by_d_y = 0.0 + d_v2_by_d_y;
float v4 = sin(v3);
float d_v4_by_d_y =
  cos(v3) * d_v3_by_d_y;
float v5 = v4 * 0.5;
float d_v5_by_d_y = 0.5 * d_v4_by_d_y;
vec3 offset = vec3(v5, 0.0, 0.0);
vec3 d_offset_by_d_y =
  vec3(d_v5_by_d_y, 0.0, 0.0);
mat3 jacobian = mat3(
  vec3(0.0),
  d_offset_by_d_y,
  vec3(0.0)
);

// Condensed:
vec3 offset = vec3(
  sin(time * 0.005 + position.y * 2.0)
    * 0.5,
  0.0,
  0.0
);
vec3 d_offset_by_d_y = vec3(
  0.5 * cos(
    time * 0.005 + position.y * 2.0
  )) * (0.0 + (2.0 * 1.0)),
  0.0,
  0.0
);
mat3 jacobian = mat3(
  vec3(0.0),
  d_offset_by_d_y,
  vec3(0.0)
);
```

Listing 2: An example of defining a twist about the horizontal axis in p5.warp.

```
const twist = createWarp(
  function({
    glsl,
    millis,
    position
  }) {
    function rotateX(pos, angle) {
      const sa = glsl.sin(angle);
      const ca = glsl.cos(angle);
      return glsl.vec3(
        pos.x(),
        pos.y()
          .mult(ca)
          .sub(pos.z().mult(sa)),
        pos.y()
          .mult(sa)
          .add(pos.z().mult(ca))
      );
    };

    const rotated = rotateX(
      position,
      position.x()
        .mult(0.02)
        .mult(millis.div(1000).sin())
    );
    return rotated.sub(position);
  }
);
```

Listing 3: A vertex shader snippet calculating an updated normal.

```
void main() {
  // Start from attributes
  vec3 position = aPosition;
  vec3 normal = aNormal;

  // Splice in auto-generated code.
  // This defines `vec3 offset`, and
  // three `vec3`s for each column of
  // the Jacobian: `dodx`, `dody`,
  // and `dodz`
  ${outputOffsetAndDerivatives()}

  position += offset;
  vec3 w =
    (normal.y == 0. && normal.z == 0.)
      ? vec3(0., 1., 0.)
      : vec3(1., 0., 0.);
  vec3 v =
    normalize(cross(w, normal));
  vec3 u = cross(v, normal);
  mat3 jacobian =
    mat3(dodx, dody, dodz);
  normal = normalize(cross(
    u + jacobian * u,
    v + jacobian * v
  ));

  // Apply camera transforms
  // and output
  gl_Position =
    uP * uVM * vec4(position, 1.);

  // Pass on to fragment shader
  vNormal = uN * normal;
}
```

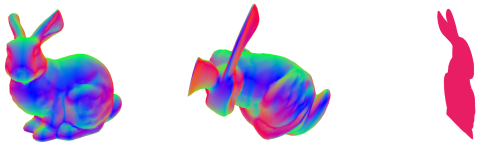
ples of other warps applied to models can be seen in figures 2d, 3, 4 and 5. For easier inspection of accuracy, they have been visualized with a fragment shader that outputs the absolute value of the updated normal as the color. Figure 7 shows warps applied to models as part of larger compositions, with lighting calculations done in their fragment shaders.

(a) Input (b) Warped

Figure 3: A cube (a) passed through a warp made with sine waves in each axis (b).

Performance. As a test of a typical use case, we measured the performance of the animation shown in Figure 1. It runs p5.warp on a mesh with around 14,000 vertices using a warp with 150 nodes in its computation graph, rendering onto a 1200×1200 pixel canvas. It runs at 60 frames per second, the imposed browser limit on a 60Hz display, on a 2021 MacBook Pro with an M1 Pro chip, a 2015 Intel MacBook Pro with integrated GPU, and a 2021 Motorola Edge Android phone in Google Chrome.

We stress-tested our method by increasing both the number of vertices in the model and the number of nodes in the computation graph. The results are listed in Table 1. When increasing the vertex count of the model, the MacBooks maintained 60fps, and the tab on the phone crashed before losing 60fps, suggesting that hardware memory limitations may be the primary bottleneck for model size rather than the performance of p5.warp with standard warp complexity. We also experimented with increasing the warp complexity by stacking random sine waves until reaching a target number of nodes in the computation graph. This had a more noticeable effect on frame rates: the phone begins to struggle at 1,500 nodes, and the 2015 MacBook begins to show signs of slowing down at 3,000. We consider these to be acceptable limits, as our manually coded warps have a smaller computation graph by a factor of 10, leaving sufficient performance buffer room.



(a) Input [16] (b) Warped with a twist (c) Warped by flattening

Figure 4: A bunny (a) retaining correct normals when passed through warps that twist about the X axis (b) and even after being flattened into the YZ plane (c).

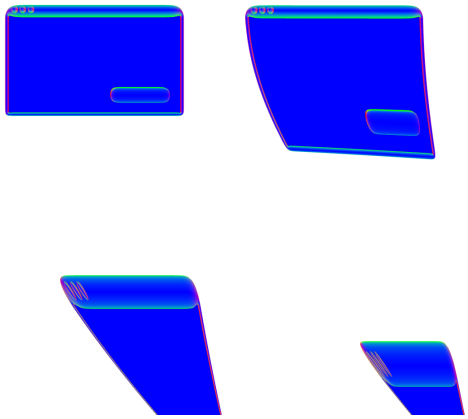


Figure 5: Frames of an animated warp inspired by the "genie" animation while minimizing a window introduced in Mac OS X.

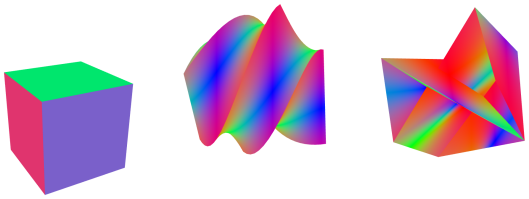
5.1 Discussion

Model resolution. Given two points $\vec{p}$ and $\vec{q}$ connected by a straight edge on a mesh M, the ideal form of the domain warped mesh has $\vec{p'} = w(\vec{p})$ and $\vec{q'} = w(\vec{q})$ connected by a curve $C(t) = w(t\vec{p} + (1-t)\vec{q}),\ t \in [0,1]$. Since the outputted mesh in WebGL will still connect $\vec{p'}$ and $\vec{q'}$ with the straight line seg-

	2021 M1 Pro MacBook Pro	2015 Integrated GPU Intel MacBook Pro	2021 Motorola Edge
14,000 vertices, 150 nodes	60fps	60fps	60fps
500,000 vertices, 150 nodes	60fps	60fps	Crash
14,000 vertices, 1,500 nodes	60fps	60fps	26fps
14,000 vertices, 3,000 nodes	60fps	50fps	10fps

Table 1: Frame rates achieved using our method on a variety of hardware, model sizes (number of vertices), and warp complexities (number of nodes in the computation graph.)

ment $L(t) = tw(\vec{p}) + (1-t)w(\vec{q})$, the visual quality of a deformed mesh depends highly on the distance between the ideal curve $C(t)$ and its approximation $L(t)$. The effect of model resolution on a high frequency warp is shown in Figure 6.



(a) Input (b) Warped, 20 edges/side (c) Warped, one edge/side

Figure 6: A cube (a) passed through a twist warp when the cube has been subdivided into 20 edges per side (b) and one edge per side (c). Without sufficient polygon detail, the linear interpolation between vertices fails to capture the full warp, producing self-intersections and less accurate normals interpolated over faces.

Continuum approximation. Warp functions are defined over $\mathbb{R}^3$ as a continuum without knowledge of the true shape of the model. While this method has been used to implement efficient hand-animated *imitations* of physical phenomena such as soft body springiness or cloth in the wind, this method is not suited for more accurate physics *simulations* where forces propagate through the true shape of the mesh.

Degenerate offsets. Rendering is rarely affected by the cases where the equations in Section 3 would produce zero-magnitude vectors as normals. Equation 1 breaks down if either operand of the cross product is a zero vector, or if both operands are equal to each other. The former case happens if, for some tangent vector $\hat{w}$, $J\hat{w} = -\hat{w}$, or in other terms, if J has an eigenvalue of -1. Rearranging the equation for the latter case, we find it is a special case of the former case:

$$\begin{aligned} \hat{u} + J\hat{u} &= \hat{v} + J\hat{v} \\ J(\hat{u} - \hat{v}) &= -(\hat{u} - \hat{v}) \\ J\hat{w} &= -\hat{w}, \qquad \hat{w} = \hat{u} - \hat{v} \end{aligned}$$

A warp where $\exists\ \hat{w} \mid J\hat{w} = -\hat{w}$ is one that maps some axis to a constant. As an example, take $f(\vec{p}) = -\vec{p} + \vec{k}$, which offsets any input to a constant in all axes, turning it into the point $\vec{k}$. Its Jacobian matrix is $-I_3$, which has an eigenvalue of -1.

The previous example choice of f causes all normals to be degenarate, but by mapping all vertices to a single point, the mesh will have no surface area and no fragments will render, eliminating the problem. If all vertices are mapped to a line, which also has no surface area, similarly, no fragments will be rendered. If all vertices are mapped to a plane, as would happen with $f(\vec{p}) = [-p_1, 0, 0]^T$, which flattens to the YZ plane, then some fragments will be rendered. Input normals tangent to the plane will have tangent vectors that get mapped to zero, being perpendicular to the plane. A face whose vertices are all such normals, if the normals are not askew from the face normal, is itself perpendicular to the YZ plane, so it will not get rendered, as it warps into a new face with zero area. Given a face where just some of its vertices have such normals, fragments in the middle of

the face will have normals interpolated between some degenerate and some valid normals. In practice, this is not a problem: degenerate normals are zero vectors, and in the barycentric interpolation performed by WebGL, one or two components being zero simply reduces the magnitude of the weighted sum of the remaining vector components. Since the normal vector is normalized back to unit length in the fragment shader after interpolation, resulting normals will still be valid. Figure 4c shows the result of flattening to the YZ plane, where all resulting normals point in the X axis.

To summarize, offset functions that produce degenerate normals do not render fragments unless they collapse inputs to a plane. As long as faces whose vertices all have parallel normals form a surface that is itself perpenducular to those normals, plane-collapsing functions still produce correct results due to barycentric interpolation and normalization of normals.

6 Conclusion and Future Work

Domain warping, a common and versatile technique in creative coding, is difficult to use in 3D due to the need to recompute surface normals. We describe a method of computing updated normals that can be implemented in a vertex shader and present a library implementing such a shader, providing automatic generation of the derivatives is requires. One can then write any warp without worrying about incorrect lighting or parameter tuning, enabling artists to freely iterate.

With support for the WebGPU API now enabled by default in Google Chrome and the access to modern hardware capabilities it provides, there may be potential in the future to create new render pipelines using compute shaders that address current WebGL limitations. Our method currently lacks hardware-accelerated adaptive subdivision based on the warp function, which could perhaps be enabled with more GPU flexibility.

Finally, the ability to warp using arbitrary functions raises the human-computer interaction question of how to best to let artists sculpt abstract warp functions. Artists in the creative coding community are open to exploring math directly, but future work could include studies of more intuitive forms of warp function control.

References

[1] J.A. Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005. doi:10.1109/TVCG.2005.49.

[2] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, aug 1978. doi:10.1145/965139.507101.

[3] Fernando De Goes and Doug L. James. Dynamic kelvinlets: Secondary motions based on fundamental solutions of elastodynamics. *ACM Trans. Graph.*, 37(4), jul 2018. doi:10.1145/3197517.3201280.

[4] Matt DesLauriers. This is created with dense vertical polylines warped by a noise function. Here is a close-up of the final renders, as well as three generations using lower line counts to visualize the algorithm., October 2018. URL: https://twitter.com/mattdesl/status/1047094498562625536.

[5] Benjamin Jotham Fry. *Computational Information Design*. PhD thesis, Massachusetts Institute of Technology, 4 2004. URL: https://benfry.com/phd/dissertation-110323c.pdf.

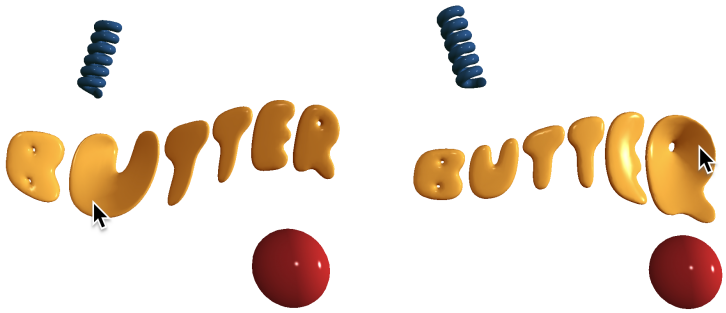
[6] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[7] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 289–296, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/74333.74363.

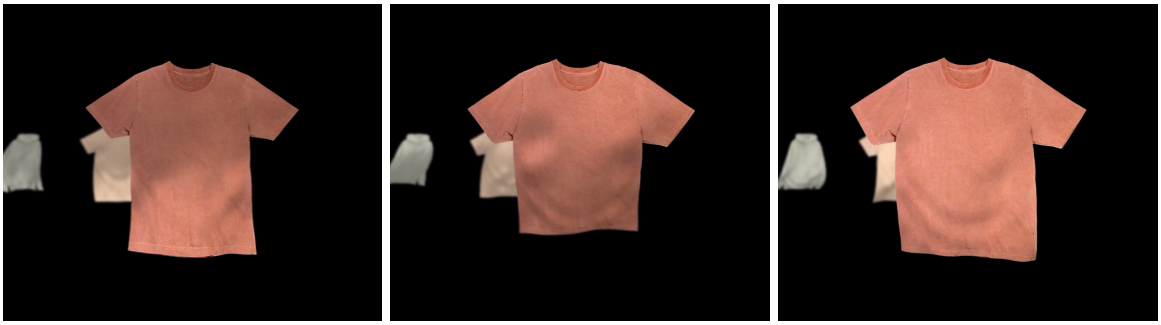
(a) Repeated, warped copies of a 3D scan of a person, creating a surreal composition.

(b) A warped spoon, playing on *The Matrix* and René Magritte's *The Treachery of Images.*

(c) A recursive hand structure using world-space warping to make fingers wiggle.



(d) A bulge effect added to 3D text based on the mouse location.



(e) Pretend cloth physics applied via a warp to a rotating carousel of shirts, where each is a textured plane.

Figure 7: Examples of domain warping using p5.warp in different sketches, using lighting calculations in their fragment shaders that make use of updated normals.

[8] Pol Jeremias and Inigo Quilez. Shadertoy: Learn to create everything in a fragment shader. In *SIGGRAPH Asia 2014 Courses*, SA '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2659467.2659474.

[9] Ben Kovach. Art Blocks gave me the opportunity to stretch this system to its limits. Edifice plays extensively with color, layout, texture, perspective, and domain warping to produce images that still consistently surprise me after generating tens of thousands of outputs. Here's Ropsten 41, November 2021. URL: https://twitter.com/bendotk/status/1455232555561754626.

[10] Morten S. Mikkelsen. Bump mapping unparametrized surfaces on the gpu. *Journal of Graphics, GPU, and Game Tools*, 15:49 – 61, 2010. URL: https://mmikk.github.io/papers3d/mm_sfgrad_bump.pdf.

[11] Processing Foundation. Processing, 2001. URL: https://processing.org.

[12] Processing Foundation. p5.js, 2014. URL: https://p5js.org.

[13] Inigo Quilez. Domain Warping, 2002. URL: https://iquilezles.org/articles/warp/.

[14] Inigo Quilez. Normals for an SDF, 2015. URL: https://iquilezles.org/articles/normalsSDF/.

[15] Gaëtan Renaudeau. Cirque de lumières, 2021. URL: https://gen.art/drops/lumieres.

[16] Stanford University. The Stanford 3D Scanning Repository, 1994. URL: https://graphics.stanford.edu/data/3Dscanrep/.

Index of Supplemental Materials

Source code for p5.warp and its GLSL-building capabilities can respectively be found at:

- https://github.com/davepagurek/p5.warp
- https://github.com/davepagurek/glsl-autodiff

A snapshot of both repositories is present in the supplemental material in a `.zip` file.

Author Contact Information

Dave Pagurek van Mossel
Butter Creatives
dave@davepagurek.com