Origami Simulation in The Web Browser Using WebAssembly

Umar Ahmed
umar.ahmed@mail.utoronto.ca
University of Toronto
Toronto, Ontario, Canada

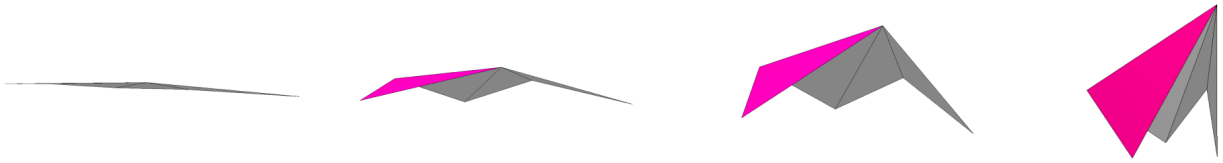

Figure 1: Simulation of mountain and river folds on a simple mesh using WebAssembly implementation

ABSTRACT

In this paper, I present a re-implementation of an origami fold simulator originally proposed by Ghassaei et al. in their paper titled "Fast, Interactive Origami Simulation using GPU Computation" [1]. My implementation is similar to the original one in that it makes use of web technologies like WebGL to visualize the simulation, but deviates from the original paper by implementing the actual simulation logic on a separate CPU thread using the WebWorker API and WebAssembly as the runtime target rather than on the GPU using GLSL.

CCS CONCEPTS

• **Computing methodologies** → **Physical simulation**.

KEYWORDS

origami, physical simulation, webassembly

ACM Reference Format:
Umar Ahmed. 2020. Origami Simulation in The Web Browser Using WebAssembly. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Origami is the art of paper folding, typically attributed to the Japanese culture. Origami art pieces typically start with a flat, square sheet of paper or foldable material to which an origami artist applies a series of folds which combine to form a, usually three-dimensional, sculpture.

To teach others how to create complex sculptures with multiple folds, origami artists typically share what is a known as a "crease pattern", which is a diagram that depicts all or most of the creases in the final origami piece.

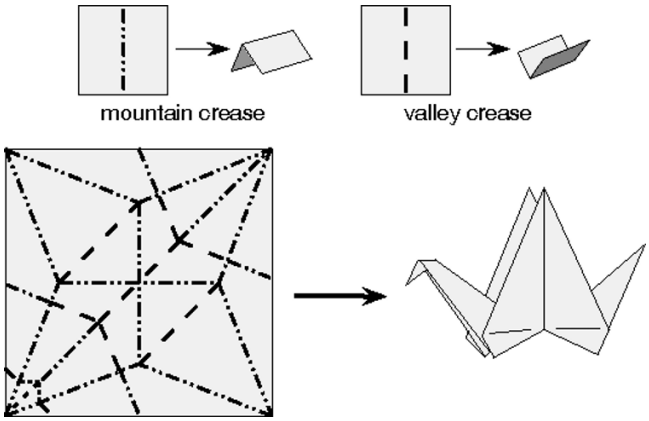

Figure 2: The crease pattern for the classic flapping bird model, with mountain and valley creases indicated, via The Conversation. (https://bit.ly/2LM6u79).

Using physical simulation, origami artists can visualize the effects of different crease patterns on the final, folded state of the

material. They can assess goals like fold complexity, global developability, and stresses on the material, which can be used to optimize the crease pattern.

In this work, I re-implement the algorithm designed and implemented by Ghassaei et al., originally implemented on the GPU using WebGL fragment shaders, instead on the CPU in a separate thread via the WebWorkers API. The implementation of the algorithm is written in the AssemblyScript programming language that was designed to compile to a new efficient bytecode format for the Web called WebAssembly.

In the following sections, I will outline the algorithm that was implemented, how my implementation varies from the original, and the results of some simulations on different crease patterns.

2 RELATED WORK

My implementation primarily follows the algorithm outlined in the paper: "Fast, Interactive Origami Simulation using GPU Computation" by Ghassaei et al., and references the code implementation provided at https://github.com/amandaghassaei/OrigamiSimulator. The main deviation is that while the authors of that paper chose to implement the algorithm using the GPU via WebGL, I chose to implement the algorithm using WebAssembly in a separate WebWorker thread.

3 METHOD

3.1 Meshing

We start by producing a triangulated mesh from our crease pattern, which consists of a series of creases of different types:

- `mountain`: Produces a concave fold when viewed from above
- `river`: Produces a convex fold when viewed from above
- `facet`: Additional crease added to enforce triangulation and driven flat by constraints
- `boundary`: Edges of the paper

Each of the edges in the resulting mesh are modelled as a beam-jointed truss with constraints on joint angle and beam length.

3.2 Constraints

3.2.1 Axial. For each of the edges, we model an axial constraint using a mass-spring model where each of the edge endpoints is a mass and the edge between them is a linear-elastic spring. Using Hooke's Law, we get the following equations for the forces acting on each of the endpoints:

$$F_{axial_1} = k_{axial}(l - l_0)\hat{l}_{12} \tag{1}$$

$$F_{axial_2} = -k_{axial}(l - l_0)\hat{l}_{12} \tag{2}$$

where k_{axial} is a user-defined constant representing the spring stiffness, l is the current length of the edge, l_0 is the length of the undeformed edge, and $\hat{l}_{12}$ is defined as the unit vector from node 1 to node 2.

3.2.2 Crease. To drive folding, we introduce a constraint on the dihedral angle between neighboring triangular faces in the mesh. The target fold angle θ_{target} is user-defined on a per-edge basis and we drive the mesh towards this angle using a linear-elastic

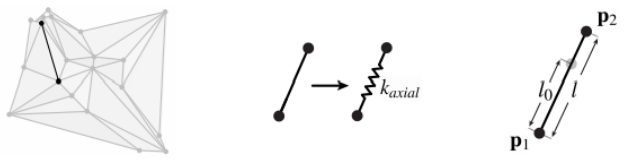

Figure 3: Edges are formulated as linear-elastic springs between endpoints

torsional spring:

$$F_{crease} = -k_{crease}(\theta - \theta_{target})\frac{\partial \theta}{\partial p} \tag{3}$$

where k_{crease} is a user-defined constant representing the torsional spring stiffness, θ is the current angle, and where the partial derivatives are defined as:

$$\frac{\partial \theta}{\partial p_1} = \frac{n_1}{h_1} \tag{4}$$

$$\frac{\partial \theta}{\partial p_2} = \frac{n_2}{h_2} \tag{5}$$

$$\frac{\partial \theta}{\partial p_3} = \frac{-\cot \alpha_{4,31}}{\cot \alpha_{3,14} + \cot \alpha_{4,31}}\frac{n_1}{h_1} + \frac{-\cot \alpha_{4,23}}{\cot \alpha_{3,42} + \cot \alpha_{4,23}}\frac{n_2}{h_2} \tag{6}$$

$$\frac{\partial \theta}{\partial p_4} = \frac{-\cot \alpha_{3,14}}{\cot \alpha_{3,14} + \cot \alpha_{4,31}}\frac{n_1}{h_1} + \frac{-\cot \alpha_{3,42}}{\cot \alpha_{3,42} + \cot \alpha_{4,23}}\frac{n_2}{h_2} \tag{7}$$

where α's are the interior angles of the triangular faces as defined in the diagram below, n_1 and n_2 are the unit normal vectors as depicted below, and h_1 and h_2 are the lengths of the beams as pictured below as well.

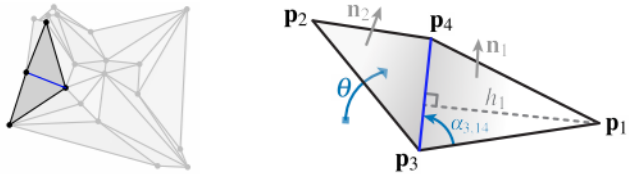

Figure 4: Constraint formulation for a single crease

3.2.3 Face. To increase stability, we add a constraint on each of the interior angles of each triangular face in the mesh according to the following formula:

$$F_{face} = -k_{face}(\alpha - \alpha_0)\frac{\partial \alpha}{\partial p} \tag{8}$$

where k_{face} is a user-defined stiffness constant, α is the current interior angle, α_0 is the interior angle of the undeformed face, and the partial derivatives are defined as follows for the example angle on each it's neighbors:

$$\frac{\partial \alpha}{\partial p_1} = \frac{n \times (p_1 - p_2)}{\|p_1 - p_2\|^2} \tag{9}$$

$$\frac{\partial \alpha}{\partial p_2} = -\frac{n \times (p_1 - p_2)}{\|p_1 - p_2\|^2} + \frac{n \times (p_3 - p_2)}{\|p_3 - p_2\|^2} \tag{10}$$

$$\frac{\partial \alpha}{\partial p_3} = \frac{n \times (p_3 - p_2)}{\|p_3 - p_2\|^2} \tag{11}$$

We use similar equations for each of the other two interior angles.

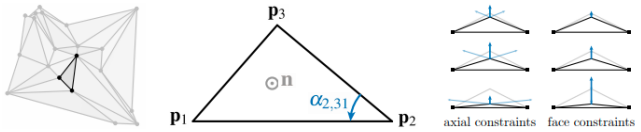

Figure 5: Face constraint formulation for angle at p_2

3.3 Damping

In order to speed up convergence to a static state, we introduce a damping force to each endpoint of an edge as outlined in the original paper defined as:

$$F_{damping} = c(v_{neighbour} - v) \tag{12}$$

where c is the damping coefficient, and where $v_{neighbour}$ and v are the velocities of the nodes on the edge.

3.4 Acceleration

In order to compute the acceleration, we add up all the forces for each vertex and divide by the mass:

$$a = \frac{F_{total}}{m} = \frac{\sum_{beams} F_{axial} + \sum_{creases} F_{crease} + \sum_{faces} F_{face} + \sum_{edges} F_{damping}}{m}$$

3.5 Time Integration

As recommended by the paper, we use an explicit time integration scheme using the following update equations:

$$v_{t+\Delta t} = v_t + a\Delta t \tag{13}$$

$$p_{t+\Delta t} = p_t + v_{t+\Delta t}\Delta t \tag{14}$$

4 IMPLEMENTATION

My implementation of the folding simulation relies on running the simulation logic in a separate thread in the web browser. The main thread of the web application sets up the geometry and visualization using the popular Three.js library. It then spins up a new thread using the WebWorker API and sends it the mesh information via message passing interface.

Once the worker thread has the data it needs, it instantiates the WebAssembly code, setting up the appropriate memory locations and writing data, including the vertex positions, edges, faces, and creases, to shared buffers.

Back on the main thread, we begin the draw loop and request the next time step of the simulation from the worker. The worker calls into the WebAssembly module which implements time integration using the method previously described. The worker sends back the updated vertex positions and the main thread updates the appropriate positions using Three.js BufferAttributes. The result is a fluid animation of the mesh from a flat state to the desired folded using the constraints provided.

5 RESULTS

5.1 Simple fold

The first pattern that I experimented with was a simple square pattern with a single mountain crease along one diagonal.

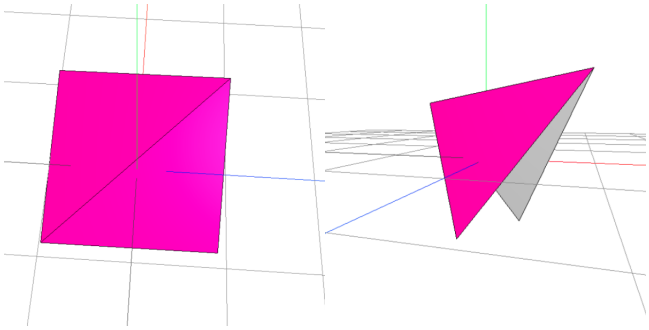

Figure 6: Left: flat pattern, Right: folded state

5.2 Complex fold

The second pattern that I experimented with was a more complex pattern with two mountain creases along the diagonals of the mesh, and a valley crease in the middle.

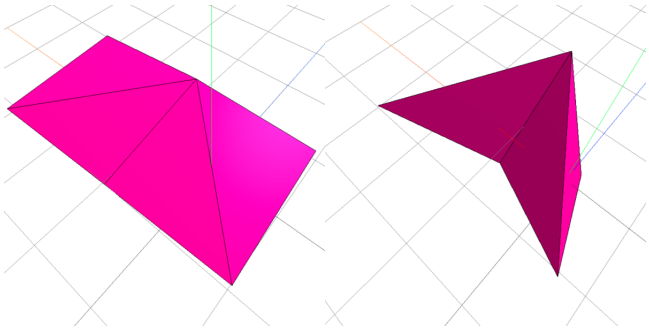

Figure 7: Left: flat pattern, Right: folded state

ACKNOWLEDGMENTS

To David Levin and TAs, for the amazing instruction in the CSC417 course this year.

REFERENCES

[1] Amanda Ghassaei, Erik D Demaine, and Neil Gershenfeld. 2018. Fast, interactive origami simulation using GPU computation. *Origami* 7 (2018), 1151–1166.