Refolding Planar Polygons

Report III

Maurice Aarts
m.a.p.aarts@student.tue.nl

Rimco Boudewijns
r.c.boudewijns@student.tue.nl

October 26, 2012

Course: Advanced Component Computer Graphics – 2IV05
Mentor: Huub van de Wetering, h.v.d.wetering@tue.nl, MF 6.087

Abstract

This paper describes an implementation of an algorithm for the refolding of multiple polygons with a guarantee of non-intersection. The implementation builds on prior results from single polygon refolding. The intersection avoidance machinery is adapted to handle multiple polygons without introducing extra complexity. It is still independent from a distance metric that determines the overall character of the interpolation sequence. A user interface is presented such that multiple interpolation sequences can be edited and calculated simultaneously.

Contents

1 Introduction **3**
1.1 Previous Work . . . 4

2 Constraints **6**
2.1 Vertex Distance . . . 6
2.2 Minimal distance . . . 6
2.3 Fixed vertices . . . 6
2.4 Fixed angles . . . 6

3 Implementation Details **7**
3.1 User Interface . . . 7
3.2 Data Structures . . . 9
3.2.1 QList and Thrust::Host_vector . . . 9
3.3 Metrics . . . 9
3.3.1 Distance . . . 9
3.3.2 Energy . . . 9
3.4 Algorithm . . . 10
3.4.1 Preprocessing . . . 10
3.4.2 Speed Limitations . . . 10
3.4.3 Frame Decision . . . 10
3.4.4 Approach Step . . . 10
3.4.5 Projection Step . . . 10
3.4.6 Avoidance Step . . . 11
3.4.7 Post-processing . . . 11
3.5 CUDA . . . 11

4 Results **13**
4.1 Functionality . . . 13
4.2 Refolding Process . . . 13
4.3 CUDA . . . 14

5 Limitations **16**
5.1 Refolding . . . 16
5.2 CUDA . . . 16

6 Conclusions and Future Work **17**

A Document Change Records **19**

1 Introduction

In this paper we describe an algorithm for generating an interpolation or "animation" from one simple (planar, non-intersecting) polygon to another simple (planar, non-intersecting) polygon, with a guarantee of non-intersection in the intermediate steps. The algorithm is based on prior work by H. Iben, J. O'Brien and E. Demaine as described in their paper "Refolding Planar Polygons"[1], and has been expanded to apply to scenes with more than one simple polygon such that a scene with two or more simple polygons can be "morphed" into another scene with a compatible pair of simple polygons.

First of all, what are simple polygons? Simple polygons are two-dimensional (flat-shape) polygons which have no self-intersections and of which the edges form a closed path. This means that each vertex has exactly two edges associated with it. Figure 1 shows a couple of examples of both valid and invalid simple polygons.

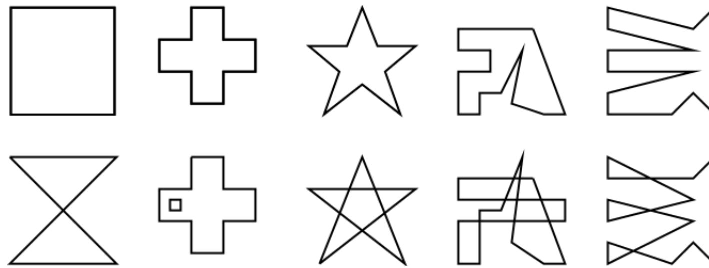

Figure 1: The top row shows simple, non-intersecting planar polygons. The polygons in the bottom row are not considered simple polygons.

In this paper we will use the words 'scene', 'frame' and 'key-frame' interchangeably, but there are a few subtle differences which we will note here.

- **Scene:** A scene is a set containing one or more polygons in a certain (user-reconfigurable/editable) state, which can be displayed as an image to the user.
- **Frame:** The polygons in a scene change during an interpolation. Each 'step' of such an interpolation has its polygons in a certain configuration. Such an intermediate state is called a frame, and is not editable by the user.
- **Key-frames:** These special 'frames' denote from which frame of a scene to which other frame we are going to do an interpolation. They are practically identical to scenes, except that they cannot be edited by the user. Key-frames are used as either the starting point or the end of an interpolation.

Every frame consists of a certain configuration of valid simple polygons, thus every frame can also be converted to a key-frame for use in another interpolation, and key-frames can be converted to scenes so that the user may edit them and use them elsewhere.

A "morph", "animation", or "interpolation" is the interpolated sequence of intermediate polygons that show the transformation of the 'starting' polygon key-frame into the 'final' polygon key-frame. Such an interpolation is done by moving the vertices from one position to another and creating a new frame based on the previous one with some vertices moved around. As each edge is related to two vertices, the edges will automatically shift along with the vertices and thus the shape remains a polygon. We strictly enforce that edges may not intersect during this interpolation. We can then build a sequence of such steps so that we obtain a complete sequence of frames from the initial key-frame to the final key-frame without any intersections during the entire interpolation.

Figure 2 shows an interpolation for a diamond shaped polygon to a square polygon, as well as the interpolation from a 'pointy' Greek cross to a St. George's variant. During the interpolation the vertices move along the arrows to their 'final' destination. Note that the cross actually morphs

from both outer frames towards the center frames which contain an intermediate state; this is an effect of the way the calculation is implemented and the output is interpreted, and will be further explained in Sections 1.1 and 3.4.

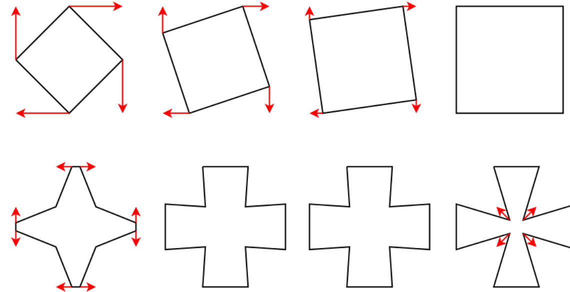

Figure 2: An animation sequence for a diamond to a square (top), and for a 'pointy' Greek cross to a St. George's cross (bottom).

As stated earlier, a scene can only be morphed into another scene if their polygons are mathematically "compatible pairs". A "compatible pair" is a set of polygons such that there is a 1-to-1 mapping between the vertices of the two polygons. Each vertex has a corresponding vertex in the other polygon using a 1-to-1 mapping. In this paper we will ensure this vertex correspondence by drawing the start and end scenes simultaneously, but other possibilities for assigning vertex correspondence do exist. We will assume that the scenes created using our implementation indeed contain a (set of) compatible pair(s), and as such the vertex correspondence problems are outside of the scope of this paper.

When doing such an interpolation, it is possible to constrain certain factors in the calculations so that the morph appears differently than without the additional constraints. The types and effects of such constraints are covered in more detail in Section 2 of this paper.

Interpolating one simple polygon to another is a calculation intensive process. Many of these calculations can be done in parallel so we will implement the necessary algorithms on both the CPU[1] and GPU[2] using NVIDIA's CUDA Thrust libraries. Using the GPU for the parallel operations shorten the required running time dramatically for large datasets.

1.1 Previous Work

This paper is based mainly on the work done by H. Iben, J. O'Brien and E. Demaine as described in their paper "Refolding Planar Polygons"[1]. Iben et al developed an algorithm that does polygon refolding on a set of simple polygons while maintaining the guarantee that there are no intersections. They built their algorithm around energy metrics based on the distance d between a vertex and the edges in the polygon. Assuming the vertex is called C and the edge has endpoints A and B, then the distance function is given as:

$$d = \begin{cases} \sqrt{(C_x - A_x)^2 + (C_y - A_y)^2} & \text{if } r \leq 0 \\ \sqrt{(C_x - (A_x + r(B_x - A_x)))^2 + (C_y - (A_y + r(B_y - A_y)))^2} & \text{if } 0 < r < 1 \\ \sqrt{(C_x - B_x)^2 + (C_y - B_y)^2} & \text{if } r \geq 1 \end{cases}$$

with

$$r = \frac{(C_x - A_x)(B_x - A_x) + (C_y - A_y)(B_y - A_y)}{(B_x - A_x)^2 + (B_y - A_y)^2}$$

In the energy metric, each vertex has a certain amount of charge e based on the distance between it and the edges in the polygon for which that particular vertex is not an endpoint, calculated as $e = \frac{1}{d^2}$. This charge goes to infinity if the polygon approaches a self-intersection. By taking the

[1] CPU: Central Processing Unit.

[2] GPU: Graphical Processing Unit.

sum of these vertex charges, it is possible to move the vertices in such a way that the total energy of the polygon only decreases until a low energy state is reached by using a simple optimization strategy such as gradient descent.

A convex polygon with all it's vertices equidistantly spaced has the lowest energy, as all it's vertices are as far away from each other as possible, so every polygon that is unfolded will move towards this fully convex state. By calculating the total energy of both the initial polygon and the final polygon, you can then choose which polygon has the highest energy and choose to move its' vertices so that the total energy of that polygon decreases. After such an interpolation, you can then again calculate the total energies and repeat the unfolding until both polygons have reached an intermediate state where they have an identical energy and configuration[3]. Figure 2 shows an example of two polygons merging to an intermediate state in the bottom row; the 'pointy' Greek cross unfolds to a normal Greek cross, thereby lowering its energy. Next the St. George's cross also unfolds to a state with lower energy, which happens to match the state of the first cross. As the crosses now match we have completed the interpolation. We have a sequence of steps from the left frame to the middle, and a sequence from the right frame to the middle frame, so to animate the sequence from left to right, we would simply show the interpolation from the left to the middle frames, and then append the interpolation from the right to the middle frame backwards. This approach also ensures that we do not have to unfold each polygon to the fully convex state and then refold it to the final state, as there is almost always an intermediate polygon that can be used with a lower energy than the two polygons in the key-frames. By using this energy-based approach, Iben et al. have proved that their algorithm ensures that no self-intersections can occur and that their algorithm must terminate successfully. The algorithm itself is explained in detail in Section 3.4. Additionally Iben et al. also built in support for constraints in their implementation, allowing certain properties to be enforced during the interpolation.

[3]For pseudocode of this algorithm, please refer to the paper by Iben et al.[1]

2 Constraints

The system could support a number of constraints that allow the user to control parts of the unfolding and refolding operations. Each constraint is enforced as strictly as possible with two exceptions. If the constraint prevents the algorithm from being able to complete the interpolation from start to finish, the constraint will be temporarily ignored until it can be enforced again. This is to prevent the algorithm from deadlocking during an interpolation. A constraint may also be ignored if another constraint conflicts with the first. The different types of constraints are explained in detail below, along with an example of when such a constraint may be temporarily ignored.

2.1 Vertex Distance

The vertex distance constraint enforces that the length between two specified vertices remains constant during the refolding process. This means that a square with sides of length 2 that is rotated 90 degrees between the start and end key-frames actually rotates, instead of having the vertices simply slide over to their new positions during which the square may temporarily become a rectangle or parallelogram. This constraint may be ignored if the distance between constrained vertices is different in the initial and final key-frames.

2.2 Minimal distance

The minimal distance constraint enforces that each vertex only moves the minimum distance required to obtain the next scene. This means that the selected vertices will move as little as possible during the interpolation from the initial to the final key-frame. This may mean that some unselected vertices have to zigzag from their initial position to their destination to avoid having to move the constrained vertices unnecessarily. This constraint may be ignored if there are multiple constrained vertices that have to move past each other, causing a conflict. In such a case one or more of the vertices may still have to travel a longer path than strictly necessary.

2.3 Fixed vertices

The fixed vertices constraint enforces that certain vertices are immobile with respect to the scene and that other vertices must move around them. This means that the constrained vertices do not move at all, and are effectively pinned to the background. This constraint will be ignored if there are multiple constrained vertices such that they are incompatible with the final key-frame, or if the distance between two pinned vertices is such that a set of edges with fixed angles as described in Section 2.4 cannot pass between them in order to complete the interpolation.

2.4 Fixed angles

The fixed angles constraint enforces that the angle between two edges at specified vertices remains constant. This ensures that certain shapes hold while the rest of the polygon is refolded to the new form. This means that for instance a square will remain rectangular during the transformation, and will not reform into a parallelogram or trapezoid. This constraint may be ignored if the constrained angles are such that they restrict the movement of those edges to their final positions, or if the constrained angles are incompatible with the final key-frame.

3 Implementation Details

3.1 User Interface

The user interface of the program as shown in Figure 3 allows the user to design custom polygons for both the initial and final shapes, as well as key-frames for the intermediate transitions. This allows the user to direct the animation by enforcing certain intermediate polygons.

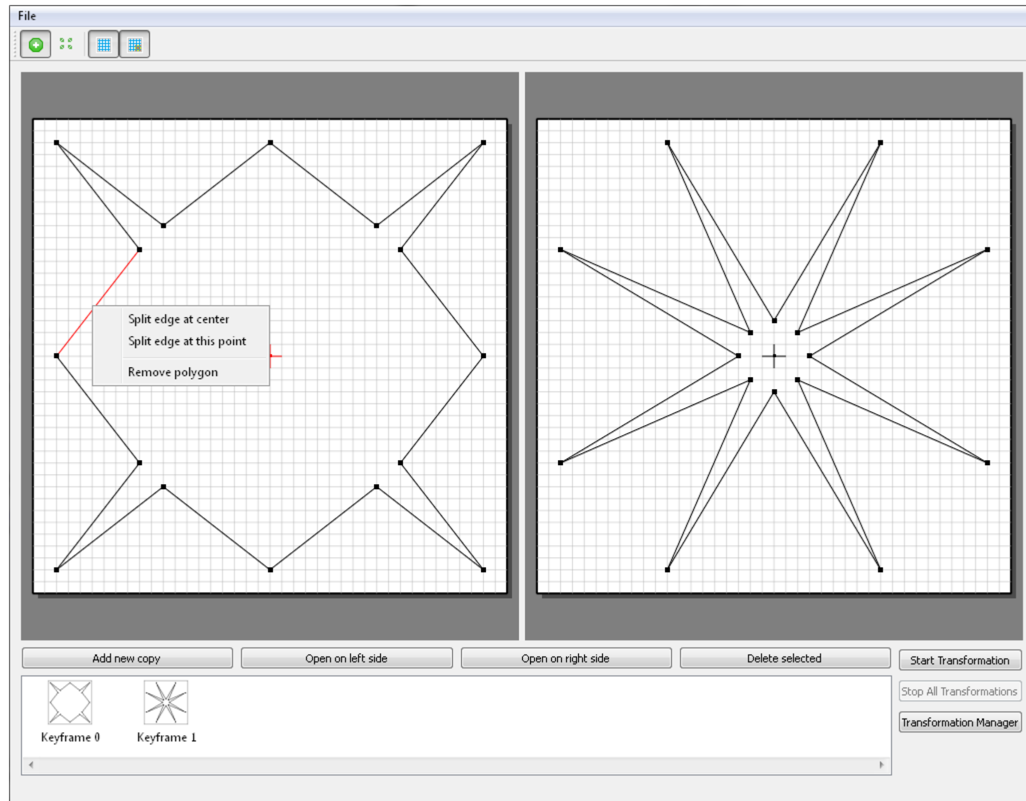

Figure 3: Main user interface for the algorithm.

To manage all running and finished transformations, the interface as shown in Figure 4 is available. The left side of the manager shows all transformations and the settings for the transformations. The right side shows the current state of the selected transformation. The red lines show the direction to the final destination, the green lines show the direction in which the energy would decrease and the blue lines show in which direction the transformer is trying to move.

If the transformation is finished, it can be reviewed by selecting the transformation and clicking on *View Selected*. This results in the interface as shown in Figure 5. The left side shows a list of all generated frames, saving options and the paths of all vertices. The right side shows the selected frame.

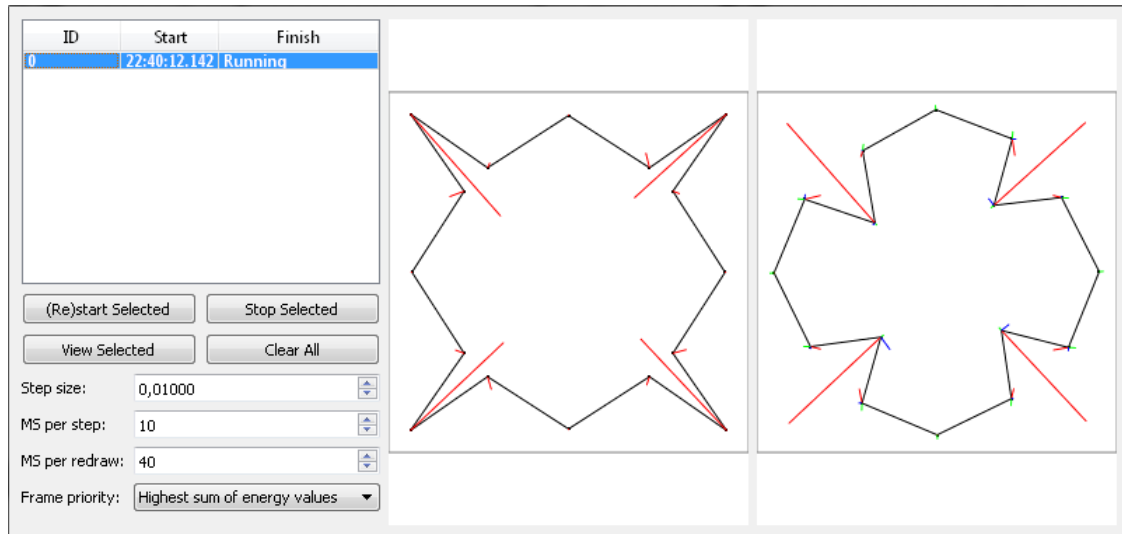

Figure 4: Transformation Manager which shows an overview of all transformations.

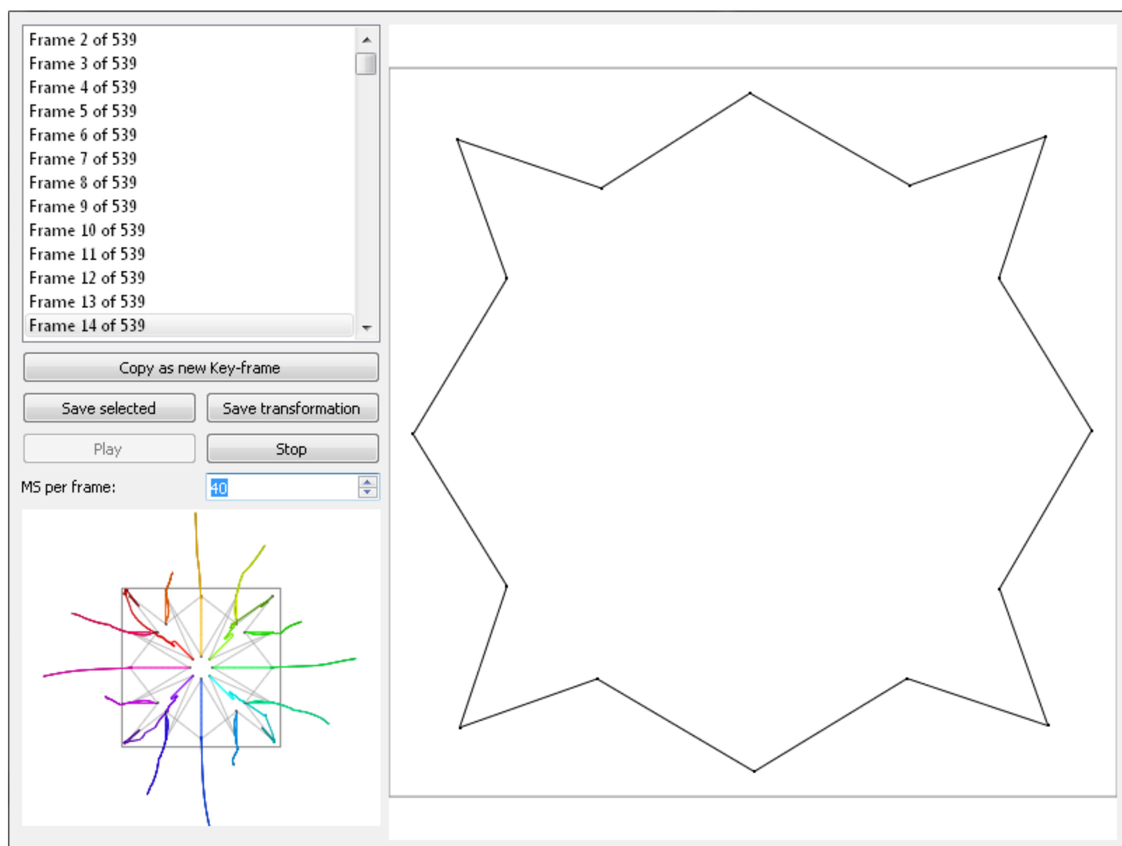

Figure 5: Transformation Viewer which can play back the calculated transformation and shows the paths of all vertices.

3.2 Data Structures

Polygons are internally stored as QLists of vertices to provide the information needed by the algorithm. Each scene or frame consists of a QList containing the polygons of the scene and helper functions. The order of the lists of two scenes determines the mapping between vertices of the original and the target scene. Each polygon also contains a set of Thrust::host_vectors, which are similar to the QLists except they are the native data format for the CUDA architecture. These vectors contain the vertices coalesced, such that one vector contains only the x-coordinates and the other contains the y-coordinates. This allows CUDA to do rapid parallel memory accesses without having to de-tuple the contents of a QList for every operation.

3.2.1 QList and Thrust::Host_vector

QList is one of Qt's generic container classes. It stores a list of values and provides fast index-based access as well as fast insertions and removals. Internally, QList is represented as an array of pointers to items of type T. If T is itself a pointer type or a basic type that is no larger than a pointer, or if T is one of Qt's shared classes, then QList stores the items directly in the pointer array. The algorithmic complexities are as follows:

Container	Index lookup	Insertion	Prepending	Appending
QList	O(1)	O(n)	Amort. O(1)	Amort. O(1)
Vector	O(1)	O(n)	O(n)	Amort. O(1)

In the table, "Amort." stands for "amortized behavior". For example, "Amort. O(1)" means that if you call the function only once, you might get O(n) behavior, but if you call it multiple times (e.g., n times), the average behavior will be O(1). For a QList, it can be brought down to O(1) by calling QList::reserve() with the expected number of items before you insert the items.

The host_vector from the CUDA thrust library is less efficient than the QList with respect to prepending data to the vector, but as this rarely happens it is not considered a problem. The advantage of the host_vector is that it allows the possibility of doing a direct memory copy from a Thrust::host_vector in system RAM to a Thrust::device_vector into the graphics card's memory without any additional calls.

3.3 Metrics

3.3.1 Distance

The Euclidean distance is chosen as distance metric for each vertex in the scene because it is simple and effective. This can be easily changed because the distance is separated from the algorithm itself. The Euclidean distance results in natural looking transitions without odd behavior. The polygon distance is defined as a list of distances between all vertices of a polygon. The frame of scene distance is defined as a list of all polygon distances.

3.3.2 Energy

The energy of each vertex is calculated using the the distance of a vertex to all other edges not directly connected to the vertex. To calculate the total energy value of a polygon, the following formula is used:

$$\sum_{i=1}^{N} \sum_{j \notin \{i,i-1\}}^{N} \frac{1}{dist(v_i, e_j)^2}$$

In this formula, N is the number of vertices, v_i is the vertex and e_i is the edge between v_i and v_{i+1}.The shortest distance between an vertex and edge is calculated by $dist(v_i, e_j)$. (The final sum also includes the vertices of other polygons such that polygons won't intersect each other.)

3.4 Algorithm

3.4.1 Preprocessing

Before the real computation can begin, the given scenes must be preprocessed such that alterations can be performed quickly. To accomplish this, the scenes are converted to frames which are smaller optimized versions. An optimized version only contains the QLists with vertexes and the metric calculation functions without the functions to edit the scene as needed by the user interface. These frames are put in two separate lists. One list will contain all steps from scene A to the intermediate frame and the other list will contain all steps from scene B to the intermediate frame. Each successful algorithm step adds a new frame to one of the lists according to the direction chosen as described in Section 3.4.3.

3.4.2 Speed Limitations

Because the steps that are generated can be viewed real-time, the speed should be limited. The first limiter limits the speed at which frames are generated such that the calculation itself is slowed down. The second limiter limits the number of frames per second that are sent to the viewer such that OpenGL has enough time to render each frame. This implies that not all calculated frames are displayed during calculation but this is no problem as it is a common fact that the the human eye cannot perceive frame rates of over 55 fps.

3.4.3 Frame Decision

The multiple polygon solution introduces a new problem because the high energy polygons can be on different frames in situations with more than one polygon on a frame. To determine which of the high energy polygons should be moved, multiple strategies can be chosen by the user:

- Sum: The frame with the highest sum of energy values is chosen
- Max: The frame containing the polygon with the highest energy value is chosen
- Count: The frame containing the most high energy polygons is chosen

Because this decision can result in a deadlock, the opposite frame is always checked if the selected frame cannot successfully move anymore. For example, if there are three polygons of which two have a high energy level on one side but both can't move without increasing some energy value. The third polygon with the high energy value on the other side can possibly move successfully. The chosen strategy influences this process.

3.4.4 Approach Step

The first actual step in the algorithm is the approach step, in which each high energy polygon in the scene is moved towards its low energy counterpart. The movement of each vertex is limited to the global maximum step size such that it won't jump to its target in one big step. After moving all vertexes, the new energy value of the frame is calculated and compared with the original value. If the energy values for all polygons in the new frame is lower lower or equal, the step is marked successful. If the step was not successful this process is repeated for each separate vertex such that they can move independently. If this still doesn't result in a new frame with lower energy values, the next step as described in Section 3.4.5 is performed.

3.4.5 Projection Step

If the direction directly approaching the target frame is not possible, it is gradually rotated in the direction of the gradient of the energy function. Eventually, a direction such that the energy is kept constant or decreases by a small amount should be found. A fraction of the approach direction is added to this projected direction to try moving to the vertex towards the target.

The influence of the approach direction is gradually decreased for each vertex individually until it approaches zero. However, the perpendicular step could still increase the energy because this is not completely predictable. If none of the vertices can move in any of the proposed directions without increasing the energy value of some polygon, the step was not successful and the step as described in Section 3.4.6 is performed.

The perpendicular motion of this step also solves problems similar to the one presented in Figure 6 because the perpendicular direction is always to the left or right. This choice will also determine the direction in which the objects rotate around each other.

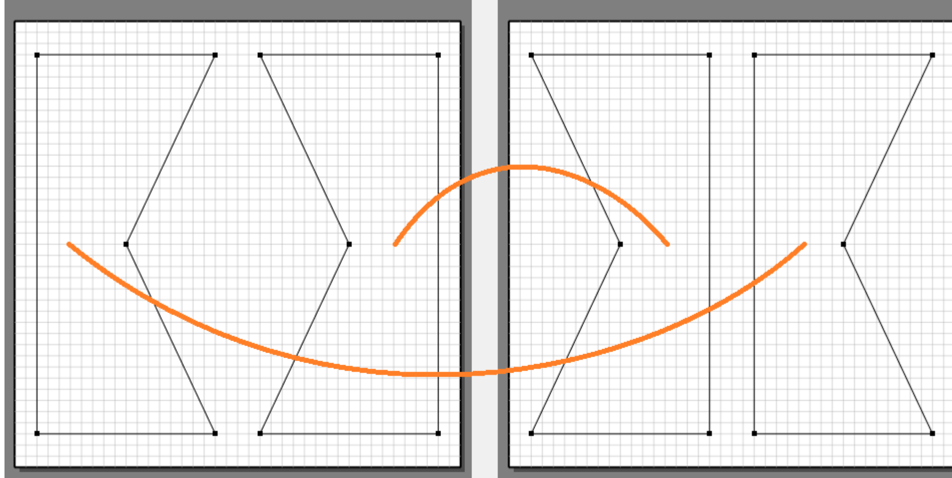

Figure 6: Two polygons that should be swapped.

3.4.6 Avoidance Step

If none of the steps moving towards the target can be performed without increasing the energy value of any polygon, vertices are moved in the direction that should decrease the energy on all polygons. This always results in a valid step, but can result in explosive motions and should be used as little as possible.

3.4.7 Post-processing

As described in Section 3.4.3, the algorithm is always run on the opposite frame to check if the decision made prohibited movement. After this check, the resources used are released and the final frames are sent to the viewers. After calculation, the list of intermediate frames is also accessible as a single list of frames starting from scene A transforming to scene B.

3.5 CUDA

The CUDA implementation uses the CPU based implementation as described above, but replaces certain calculations that can be done in parallel. These calculations include the energy calculations, the distance calculations and the approach direction calculations, as they all iterate over a list of vertices and evaluate the result of a repeated function over each vertex. CUDA allowed us to replace them with versions of those algorithms that can be run more efficiently on the GPU. Instead of doing the energy calculations for each vertex sequentially, CUDA is capable of loading a large number of vertices at the same time, and evaluating the same function over each vertex simultaneously. This allows the GPU to do a large number of vertex energy calculations at the same time, which speeds up the calculations significantly. The distance calculations and approach direction calculations are pipelined in a similar way to increase the throughput to allow faster access to the necessary function results.

CUDA can be implemented in a large number of ways, both by doing direct memory and device access, or by using a prebuilt library that interfaces with the device directly. One such library is CUDA Thrust[4], which is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's high-level interface greatly enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs, along with interoperability with established technologies such as CUDA, TBB, and OpenMP. Using the Thrust library it was relatively straightforward to implement the CUDA based version of the algorithms alongside the CPU based versions such that they could be run together or independently.

[4] http://thrust.github.com/

4 Results

4.1 Functionality

Because there were many possible features which had to be implemented in a limited time frame, not all were finished. The list of possible features and their status is presented in the following table:

Item	N/A	Partially	Completely
Basic user interface			X
Create and edit polygons			X
Create transformation between polygons			X
Change transformation speed and settings			X
CUDA computation		X	
Show paths of transformation			X
Show single frames of transformation			X
Multiple key-frames			X
Save computed frame as key-frame			X
Save transformation as picture			X
User selectable constraints	X		
Distributed computation	X		
Custom user selectable constraints	X		
User selectable distance metric	X		
Save transformation as movie	X		

4.2 Refolding Process

The current implementation successfully transforms polygons without creating intersections in the intermediate frames. The algorithm is not yet optimized and is speed limited, but can generate the transformations within a second.

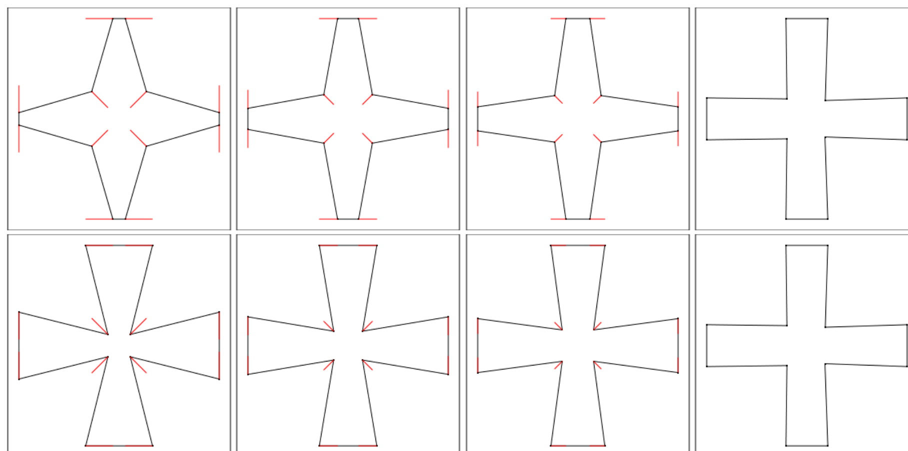

Figure 7: Successful transformation of the sawtooth polygon.

As seen in Figure 7, the implementation transforms the cross sample polygon as described in Section 1. The start and end scenes gradually morph into an intermediate frame which resembles the Swiss cross. The transformation as seen in Figure 8 shows that the implementation transforms the sawtooth sample polygon as expected. The teeth swap without intersecting each other.

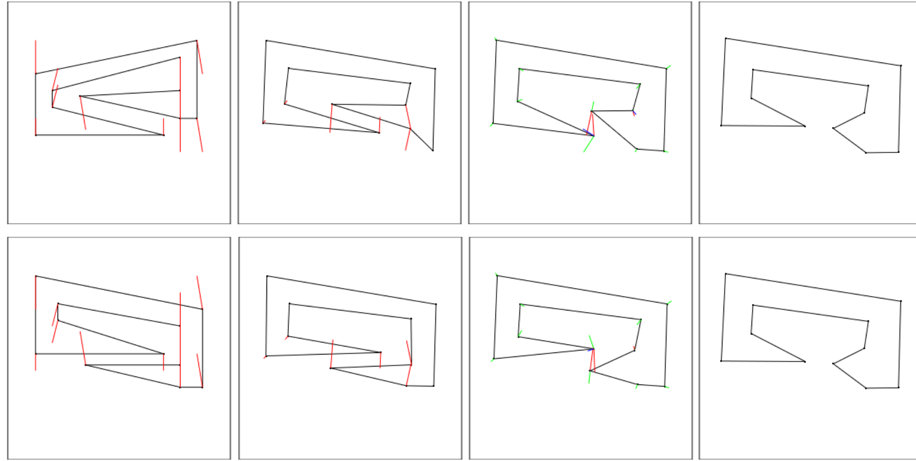

Figure 8: Successful transformation of the sawtooth polygon.

4.3 CUDA

The CUDA implementation is almost fully complete, except for the algorithm that prevents multiple polygons from intersecting each other, however the algorithms for a single polygon are fully implemented. In the CUDA based version, there is 1 large function that calculates the distances between a vertex and an edge, the energy for that combination, and the approach direction all in parallel. We used this function to run a benchmark against the similar CPU based variants of these functions to see what the actual impact was of using CUDA as opposed to using only the CPU for all the calculations. To do so we simply ran the combined energy/distance/approach direction function for CUDA, and the energy and approach direction functions for the CPU using datasets of 1 up to 3500 vertices. The benchmark system was laptop with 12GB RAM, an Intel i7 Q740 quadcore 1.73GHz (Hyperthreading) CPU and an NVIDIA GeForce GT 330M videocard (48-CUDA cores, 1GB, 128bit memory interface). The results of the benchmark are shown in Figure 9.

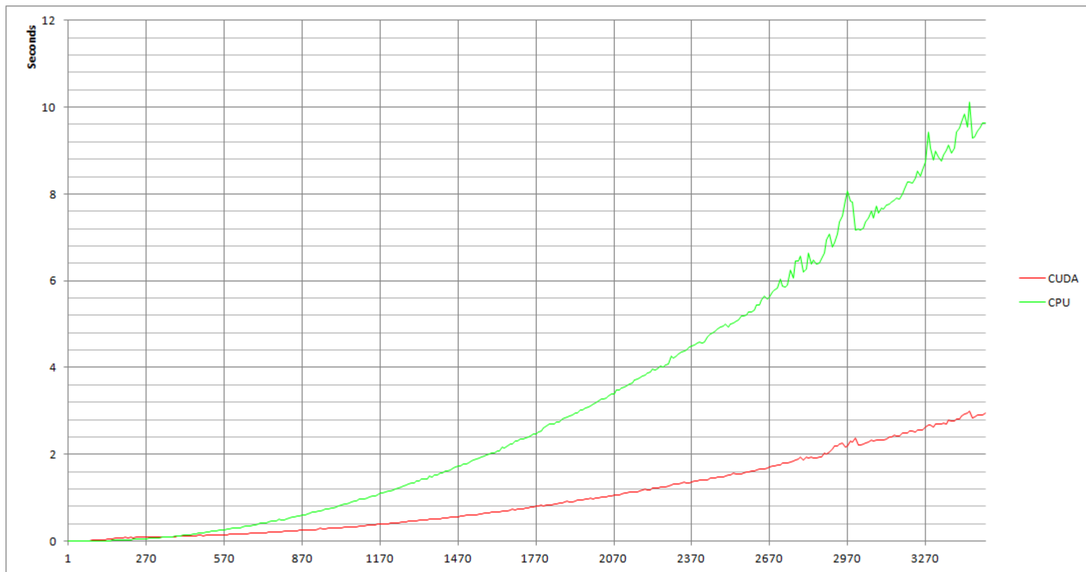

Figure 9: Comparison of the running times (in seconds) for CUDA and the CPU on the benchmark system for datasets from 1 to 3500 vertices.

As the results show, the CPU is initially more efficient, but around datasets of 350 vertices CUDA and the CPU are evenly matched and perform at the same speed, even though the CUDA variant is actually doing slightly more calculations in it's benchmark function, and the necessary copies of the input data from System RAM to the GPU's RAM. From about 400 vertices onwards the CUDA version is significantly faster than the CPU, and at approximately 700 vertices the CUDA implementation is a factor 2 faster. Note that the fluctuations in the graph near 2970 vertices and onwards were caused by external factors. From these results we can conclude that the CPU is sufficient for the refolding of ordinary planar polygons of up to around 400 vertices, but if the system were to be used for larger datasets, or perhaps to model 3-dimensional polyhedra refolding it would definitely help to use the GPU for the operations that can be parallelized.

5 Limitations

5.1 Refolding

The current implementation of the algorithm has no concrete limitations but won't necessarily compute the optimal solution. The optimal transformation is defined as the transformation in which the vertices travel the shortest distance without intersections. An example of a suboptimal solution is presented in Figure 10. The optimal solution would directly move the vertices towards their endpoint without any intersection avoidance, but the algorithm can't detect that this path is possible.

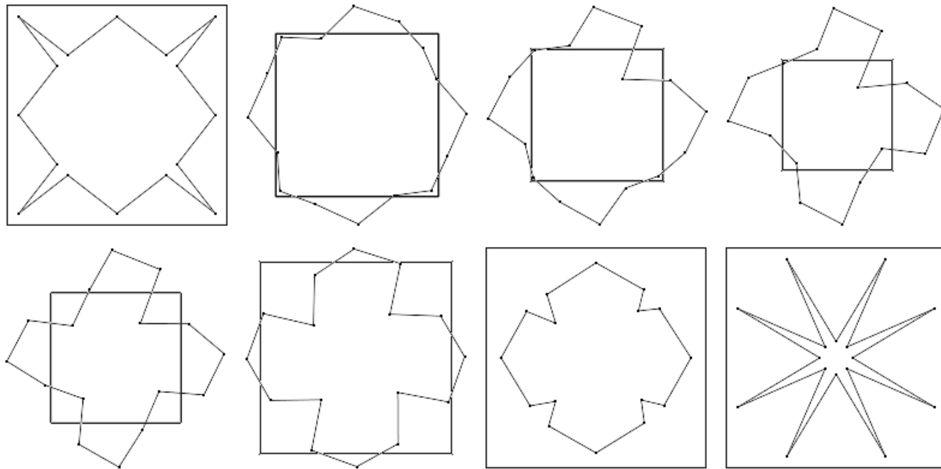

Figure 10: Transformation of a star which is not optimal.

5.2 CUDA

One of the largest limitations of the current CUDA implementation is that it requires a large amount of memory for the energy calculation, as it basically does a cartesian product of the coordinates with the edges to be able to compute them all simultaneously. On the benchmark system this limited us to approximately 4500 vertices per calculation. This can be easily resolved by doing larger calculations in multiple steps, by simply chopping the cartesian-product-dataset in parts that do fit in memory and then combining the results in an additional step. This should still be significantly faster than using the CPU. Additionally, the GPU used was a relatively low-end model and only supported single-precision floating point calculations. Newer NVIDIA cards have full support for double-precision mathematics and have a much greater amount of cores that can be used for parallel processing. The benchmark system only had 48 CUDA-cores, while many of the current high-end models have 1250 or more, along with a doubled or quadrupled amount of video memory.

6 Conclusions and Future Work

The methodology presented in [1] presents a fairly simple solution for the problem of refolding. The implementation of the algorithm still requires a large code-base because of the many needed steps to improve the solutions. The extension which enables multiple polygons in one scene is not very difficult but allows more interesting scenes to be created. The extension can use the exact same algorithm if the energy values and descent directions are adjusted accordingly. The CUDA implementation replacing some of the operations results in faster calculated information needed by the algorithm and is a fairly simple drop-in replacement. The overall implementation combines the easy-to-use user interface with the power of the algorithm.

Future work is needed to investigate which optimizations are possible to find the best possible path such that the algorithm won't result in unexpected paths. Another extension would include (fully customizable) constraints which are not compatible with the current implementation of the algorithm. Furthermore, the user interface can always be extended such that the editing process is even more simplified. Future work could also include a version of the algorithm that fully utilizes CUDA's parallel architecture so that for instance certain steps of the algorithm (such as the projection and avoidance steps) might be calculated in parallel to find the result faster. Another possibility for future work is to expand the existing algorithms to 3-dimensional simple polyhedra; using the parallel architecture of the GPU the number of calculations necessary for such an algorithm should still be feasible.

References

[1] H. N. Iben, J. F. O'Brien, and E. D. Demaine, "Refolding planar polygons," *Discrete and Computational Geometry*, vol. 41, pp. 444–460, Apr. 2009.

A Document Change Records

Page	*Section*	*Change*
3	Introduction	Reduced number of terms, improved readability.
5	Preliminaries	Removed Preliminaries and distributed information amongst other sections.
6	Polygon Parameterization	Moved to Implementation Details.
6	Implementation Details	Updated with new user interface and operations.
8	Implementation Details	Added Data Structures.
8	Implementation Details	Explained the different versions of the metrics.
9	Algorithm	Added example deadlock.
10	CUDA	Added implementation details.
11	Results	Added table of implemented features.
12	CUDA Results	Added graph of running times.
13	Refolding Limitations	Added example explaining optimal solutions.
13	CUDA Limitations	Detailed CUDA limitations.
14	Conclusions	Added CUDA conclusions.