Computing Popularity Maps with Graphics Hardware

Marta Fort [*] J. Antoni Sellarès [*] Nacho Valladares [*]

Abstract

The popularity of a point is a measure of how many of a set of moving objects have visited the point. The popularity map is the subdivision of the plane into regions where all points have the same popularity. In this paper we propose an algorithm to efficiently compute popularity maps that takes benefit of the Graphics Processing Unit parallelism capabilities. We also present experimental results obtained with the implementation of our algorithm.

1 Introduction

Mobile devices are able to generate trajectories of moving objects (for example vehicles, people or animals), called entities, available as points representing a position in space in a certain instant of time. Trajectory databases, in many cases rather large in volume, contain valuable and implicit knowledge that needs to be extracted. Several exact and approximate algorithms, based on computational geometry techniques, to detect group movement patterns have been proposed [8, 6, 3, 1, 7].

We are interested in the problem of detecting popular regions among trajectories, they are places that are visited by many entities. The localization of popular regions has multiple applications in real life. In traffic planning, we are interested in the most congested places. In tourism management, we want to know the locations of a historical town which are more visited by tourist. In marketing, we want to ensure the effectiveness of an advertisement in a mall determining how many people have seen it. Depending on the application it is convenient to know the exact number of times that an entity has visited the place or only to know if it has visited the place or not. In the traffic planning example it is necessary to count the number of times that a car has been in a place. In the tourism management example it does not matter whether a tourist has been in the place once or more than once, we are just interested in how many different tourists have been there. In the marketing example, both criteria could be applied.

Popular regions were first studied in [2]. The authors present a continuous model where entire trajectories, described by polylines whose vertices are the positions of entities at consecutive time steps, are taken into account. Given a set T of n trajectories with τ time steps each, $r > 0$ a real value and $k > 0$ an integer, a point p is a (r,k)-popular point if there are at least k different trajectories of T that intersect the square $S(p,r)$. The parameter r models the proximity between the point and the trajectories and parameter k measures the point popularity. Note that the entities do not have to be is the square simultaneously. A (r,k)-popular region is defined as a maximal connected set of (r,k)-popular points. It is not difficult to see that popular regions are polygons. Denote $\mathscr{P}_{r,k}(T)$ the collection of (r,k)-popular regions. In [2] an algorithm, rather difficult to implement, to compute $\mathscr{P}_{r,k}(T)$ that takes $O(\tau^2 n^2)$ time and requires $O(\tau n + V)$ space, where V denotes the total number of vertexes of $\mathscr{P}_{r,k}(T)$, is presented. No results of the implementation are reported. Finally, the paper remarks that another natural way of defining a popular place is by using a disk instead of a square and that more sophisticated techniques are needed to handle this new problem.

In [5] authors presented algorithms, that take benefit of the Graphics Processing Unit (GPU) parallelism capabilities, to detect popular regions in the continuous model when a disk $D(p,r)$ of center p and radius r is used to determine proximity instead of a square. The paper uses a **weak criterion** to count intersections: a trajectory intersecting the disk $D(p,r)$ multiple times counts only once.

In this paper we use a **strong criterion** for counting the number of intersections: the number of intersections between a trajectory t and the disk $D(p,r)$ is the number of maximal sub-trajectories of t contained in $D(p,r)$.

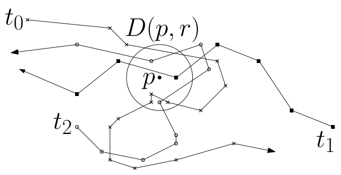

Figure 1: The number of intersections is three for the weak criterion and five for the strong criterion.

The GPU inherent parallelism and the ability to work independently alongside the CPU as a co-processor make it a compelling platform for computationally demanding tasks.

In this paper, by working towards practical solutions, we use Computational Geometry techniques together with the GPU capabilities to detect and visualize popular regions with good running times. Although our proposed approach makes reported solutions approximated, it is acceptable since data of moving entities are also approximated. We also present and discuss experimental results obtained with the implementation of our algorithm.

[*]Email: {mfort,sellares,ivalladares}@ima.udg.edu. Univ. de Girona, Spain. Partially supported by the Ministerio de Ciencia y Innovación under grant TIN2010-20590-C02-02.

1.1 Graphics pipeline and Cg

The OpenGL graphics pipeline [9] is divided into several stages. The input is a list of 3D geometric primitives expressed as vertexes defining points, lines, etc. with attributes associated. The output is a buffer (also called image) corresponding to a two dimensional $H \times W$ grid whose cells are called pixels, H and W are the height and width screen resolution, respectively. In the first stage of the pipeline, per-vertex operations take place. Each input vertex is transformed from 3D coordinates to window coordinates obtaining 2D primitives. The following stage (rasterization) rasterizes every obtained primitive according to the screen resolution, and fragments, with their attributes, are obtained. Different primitives can be projected in the same 2D space and several fragments can belong to the same pixel position. The last stage, the fragment stage, computes the color, alpha and depth value of each fragment, these values determine the final output. The color of each pixel is obtained by taking into account all fragments corresponding to the given pixel. Finally, the depth and stencil tests, among other tests, take place to determine whether a fragment is painted or not. They are executed in order and only when are enabled. Depth test uses an internal GPU buffer called depth buffer. It is used to discard fragments based on the comparison between the already stored value in the (x,y) buffer position and the incoming one, storing, on its per defect configuration, the closest fragment.

The operations performed in the graphics pipeline conform a sequence of non user controlled consecutive operations. Shader languages like Cg provide the user the ability of modifying some pre-established operations, to adapt the pipeline to the user needs by using 'shaders', small programs which modify the stage behavior.

2 Popularity maps

Let T be a set of n trajectories $T = \{t_0, \dots, t_{n-1}\}$. Each trajectory t_i is a sequence of τ points in the plane, $t_i : p^i_0, \dots, p^i_{\tau-1}$, where p^i_j denotes the position of entity e_i at time j with $0 \leq j \leq \tau - 1$. We assume a continuous model in which the movement of an entity e_i from its position p^i_j to its position p^i_{j+1} is described by the straight-line segment joining p^i_j and p^i_{j+1}, this movement is supposed to be done in a constant speed. The trajectory t_i is described by the polyline, which may self-intersect, whose vertices are the trajectory points $t_i : p^i_0, \dots, p^i_{\tau-1}$.

For a given parameter $r > 0$, the **popularity of a point** p is the total number of intersections between the trajectories of T and the disk $D(p,r)$ of center p and radius r. The **popularity map**, $\mathscr{M}_r(T)$, is the partition of the plane in maximal connected regions so that all points of a region have the same popularity. Notice that the set of points of a given popularity may be composed of several independent connected regions bounded by straight-line segments and circular arcs.

For each edge $e^i_j = p^i_j p^i_{j+1}$ of the polyline representing trajectory t_i we consider the offset region $O_r(e^i_j)$ obtained sweeping along e^i_j a disk of radius r such that the center moves on e^i_j. Then we define $P_r(e^i_j)$ as $P_r(e^i_j) = O_r(e^i_j)$ if $j = 0$ and $P_r(e^i_j) = O_r(e^i_j) \setminus D(p^i_j, r)$ otherwise. The popularity map $\mathscr{M}_r(T)$ can be obtained from the arrangement of regions $P_r(e^i_j)$, $0 \leq j \leq \tau - 1$, $0 \leq i \leq n-1$ (Figure 2.a). Notice that the intersection between the regions $P_r(e^i_j)$ and $P_r(e^i_{j+1})$ determined by two consecutive edges e^i_j, e^i_{j+1} of trajectory t_i, defines a region such that a disk centered in any of their points and radius r intersects t_i in two maximal subtrajectories, one contained in e^i_j and the other in e^i_{j+1}, and therefore the number of intersections between t_i and the disk is two.

To compute $\mathscr{M}_r(T)$ we will discretize the plane into $H \times W$ points such that each point corresponds to a pixel in the graphics pipeline. The idea is to send regions $P_r(e^i_j)$ to the GPU as a set of rectangles and disks. Inside the pipeline all primitives will be rasterized into fragments according to the screen resolution $H \times W$. Then, we will store the number of fragments corresponding to each pixel. In this way we obtain the popularity of each pixel and consequently a popularity map discretization $\mathscr{M}_r(T)$ (Figure 2.b).

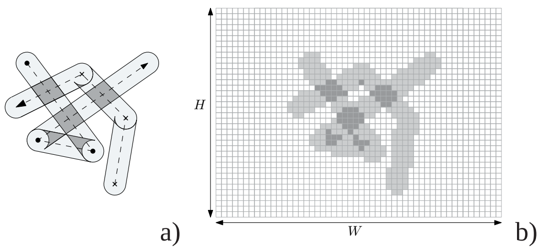

Figure 2: a) Popularity map. b) Discretized popularity map.

The disk $D(p^i_j, r)$ is painted as a square centered at p^i_j of side $2r$. Then a fragment shader is activated such that fragments whose Euclidean distance to p^i_j is bigger than r are discarded.

2.1 Computing popularity maps

Given a value of parameter r, we could compute the popularity map under the strong criterion with a similar algorithm to the one used in [5]. There, each trajectory is painted at a different depth and the stencil buffer is used to count the number of fragments with different depth that correspond to each pixel. To handle the strong criterion we should paint each trajectory edge, instead of the whole trajectory, at a different depth. To avoid overflows, the stencil buffer has to be read to CPU every 255 time steps which is too much often. Thus we propose an alternative method which avoids this read backs to CPU.

The idea is to paint each $P_r(e^i_j)$ with a different depth value and via fragment shader increment, for each fragment, the color value of its pixels. When all $P_r(e^i_j)$ are painted we can determine how many fragments correspond to each pixel depending on its final color.

The algorithm needs to add color to each pixel for each fragment. Since we can not read from a texture and at the same time write on it, we use two textures. Texture A with the previous color of each pixel, and texture B to store the new accumulated color. At each rendering pass the accumulated colors of B are copied to texture A to be recovered for the next time step as the last color stored.

Since we have 3 channels of color (RGB) with 8 bits per channel $[0\dots255]$ we add 1 to the first channel for each fragment until the red channel is overflowed. Then we increment the green channel with 1 and we set the red channel to 0, adding again to red channel until it is overflowed again. The same process is applied to the blue channel when the green channel is overflowed. We can store up to 2^{24} values which is more than enough in most applications.

The algorithm proceeds as follows. We paint each offset region $O_r(e^i_j)$ at a different depth value $z = z^i_j$ from further ($z = 1$) to closer ($z = 0$) (Figure 3). The fragment shader is activated during all the process not just for adding the color values for all fragments but also to paint a disk when needed as explained in Section 2. Since we have a depth precision of 32 bits we can paint up to 2^{32} time steps at different depth levels.

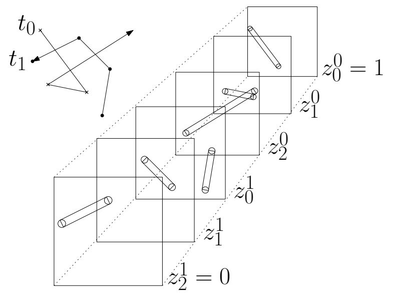

Figure 3: Offset regions rendered from z=1 to z=0.

Notice that the disk corresponding to p^i_j, $1 \le j \le \tau - 2$, is painted twice, one for each trajectory edge it defines, with different depth values. This causes the color value to be incremented twice where it should count only once. In addition the overlapped regions between rectangles and disks must be avoided too. To solve this problem we add a parameter to the shader to inform of whether the disk fragments have to increment the color or only modify the depth value. The shader updates the depth value but does not increment the color when painting the first disk. When the rectangle and the second disk are painted, both, color and depth values are updated. This way the overlapped fragments between the two disks and the rectangle will be discarded by the depth test and the disks painted twice are counted just once.

While the number of time steps is smaller than 2^{32} and the popularity of a point is smaller than 2^{24} not read backs to CPU have to be done. This turns to 0 the number of read back from GPU in most applications.

2.2 Complexity analysis

We will use the following notation. We denote by P_x the time needed to render and color x fragments, A_x the time spent to make x accesses to a texture, C_x the one needed to copy x pixels from texture to texture.

The number of painted disks for each trajectory is $O(2\tau)$, from which, τ make accesses to texture values to update the color. They represent $8\tau r^2 nHW$ painted pixels and $\tau 4\pi r^2 HW$ texture accesses. Concerning the rectangles, both, the number of accesses to a texture and pixels painted is $2LrHW$, where L is the sum of all the trajectories length. Finally the information of texture B is copied to texture A a total of τ times per trajectory, providing $HW\tau n$ copied values. Thus, the time complexity is $O(P_{(8\tau r^2 n+2Lr)HW} + A_{(n4\pi r^2\tau+2Lr)HW} + C_{n\tau HW})$ and no extra space is needed.

3 Popular regions

Let T be a set of n trajectories, $r > 0$ be a real value and $k > 0$ an integer. A point p is a (r,k)-**popular point** if the total number of intersections between the disk $D(p,r)$ of center p and radius r and the trajectories of T is at least k. We define a (r,k)-popular region as a maximal connected set of (r,k)-popular points. A (r,k)-popular region is bounded by straight-line segments and circular arcs. It is not difficult to see that a (r,k)-popular region is a maximal connected region conformed by regions of the popularity map $\mathscr{M}_r(T)$ whose points have popularity at least k. We denote $\mathscr{P}_{r,k}(T)$ the collection of (r,k)-popular regions (Figure 4).

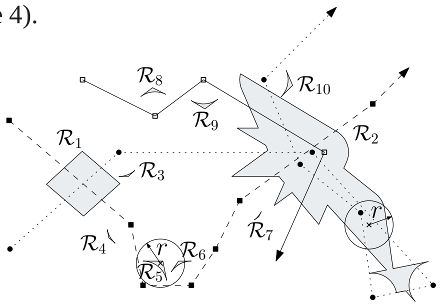

Figure 4: Example with 3 trajectories. Points marked with crosses are $(r,2)$-popular points. We have $\mathscr{P}_{r,2}(T) = \{\mathscr{R}_1, \mathscr{R}_2, \dots, \mathscr{R}_{10}\}$.

A discretization of $\mathscr{P}_{r,k}(T)$ can be easily obtained from the discretized $\mathscr{M}_r(T)$. We assign to a pixel of $\mathscr{P}_{r,k}(T)$ value 1 if its corresponding pixel in $\mathscr{M}_r(T)$ has popularity at least k and value 0 otherwise.

3.1 Popular regions visualization

We could visualize $\mathscr{P}_{r,k}(T)$ as a binary image painting in black, pixels whose $\mathscr{P}_{r,k}(T)$ value is 0 an in white those with value 1. Alternatively, in order to obtain a better visual information, we can visualize $\mathscr{P}_{r,k}(T)$ from the discretized $\mathscr{M}_r(T)$: pixels of $\mathscr{M}_r(T)$ with value less than k

are painted white and the rest of pixels are painted according to its popularity. Then, instead of having just two colors, we uniformly distribute the popularity range values among the whole RGB range (red, green and blue) (See Figure 5.b). In particular, when $k = 1$ we obtain the visualization of the popularity map $\mathscr{M}_r(T)$.

4 Results

Tests have been computed in a Intel Core 2 CPU 2.13GHz, 2GB RAM and a GPU NVidia GeForce GTX 480. Each running time reported in Table 1 is the average of 10 executions with the same parametrization.

The algorithm has been tested under different data sets. 'Animals Sim.' is a synthetic data sets generated with Netlogo [11] where 10.000 animals move on a terrain with no movement restrictions interacting each other to get closer. A total of $200,000$ time steps are tracked. 'Buses' is a set of school buses moving in Athens metropolitan area extracted from [10] with 145 buses registered during $66,096$ time steps. Finally 'State Fair' is a daily GPS track collected from the human movement in NC State Fair held in North Carolina [4], 19 persons are tracked with a total of $5,861$ time steps. Table 1 shows the running times needed to compute $\mathscr{M}_r(T)$ plus $\mathscr{P}_{r,k}(T)$ and to visualize $\mathscr{P}_{r,k}(T)$ at resolutions 512×512 and 1024×1024.

			Computation (s)		*Visualization (ms)*	
T	r	k	512^2	1024^2	512^2	1024^2
State fair	2	5	0.344	0.440	3.836	12.609
	5	5	0.351	0.439	3.293	10.564
	10	5	0.346	0.437	2.604	8.764
	5	2	0.346	0.437	2.506	8.938
	5	10	0.347	0.437	4.053	12.589
	5	50	0.346	0.450	6.445	21.304
Buses	2	5	3.401	4.469	2.507	9.241
	5	5	3.399	4.470	2.315	8.864
	10	5	3.400	4.408	2.249	8.716
	5	2	3.402	4.465	2.217	8.588
	5	10	3.398	4.474	2.457	9.179
	5	50	3.401	4.585	3.078	10.661
Animals Sim.	2	5	10.148	12.808	12.824	57.552
	5	5	10.164	12.812	12.744	55.121
	10	5	10.168	12.805	12.435	50.165
	5	2	10.114	12.788	12.126	47.694
	5	10	10.166	12.755	12.842	57.822
	5	50	10.125	13.138	12.692	58.314

Table 1: Computational (in seconds) and visualization (in milliseconds) times for different data sets of trajectories.

From the table we conclude that the computation and visualization times are fairly affected by r or k. Note that the bigger the number of time steps, the bigger the running times. This is what we expect since the number of renders and fragment processing is directly proportional to the number of time steps. The provided inputs, from $5,861$ to $200,000$ time steps, show a good scalability of our algorithm. We can not compare our execution times against others because, from the best of our knowledge, no other implementations exist.

It is easy to see that the error produced by our algorithm due to space discretization into pixels is inversely proportional to the discretization size and directly proportional to the covered area of the input data. For instance if the input data covers a squared area of 16 Km^2 the error produced is of 3.9 meters per pixel. This is a reasonable error because GPS signals also produce similar errors. To decrease the error we can increment the discretization size, this has GPU hardware limitations. A possible solution is to subdivide the area into sub-areas and apply the algorithm for each one increasing the discretization size.

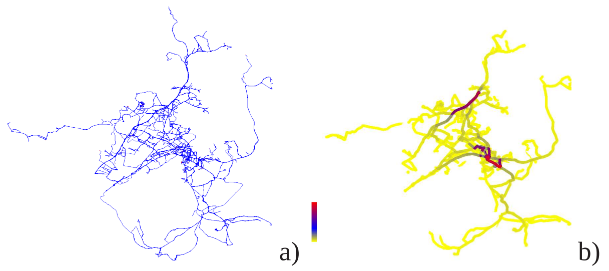

Figure 5: a) Buses dataset. b) $\mathscr{P}_{r,k}(T)$ visualization. Yellow, grey/blue and red regions means low, median and high popularity respectively.

References

[1] M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle. Reporting Leaders and Followers among Trajectories of Moving Point Objects. *GeoInformatica*, 12(4):497–528, 2008.

[2] M. Benkert, B. Djordjevic, J. Gudmundsson, and T. Wolle. Finding popular places. *Int. Journal of Computational Geometry and Applications (IJCGA)*, 20(1):19–42, 2010.

[3] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Computational Geometry*, 41(3):111–125, 2008.

[4] C. Dartmouth. CRAWDAD. `http://crawdad.cs.dartmouth.edu/index.php`.

[5] M. Fort, J. A. Sellarès, and N. Valladres. Computing popular places using graphics processors. In *Proc. SSTDM-10 in cooperation with IEEE ICDM-10*, pp 233-240, 2010.

[6] J. Gudmundsson, M. J. van Kreveld, and B. Speckmann. Efficient Detection of Patterns in 2D Trajectories of Moving Points. *GeoInformatica*, 11(2):195–215, 2007.

[7] J. Gudmundsson, and M. J. van Kreveld. Computing longest duration flocks in trajectory data. In R. A. de By and S. Nittel, editors, *GIS*, pages 35–42. ACM, 2006.

[8] P. Laube, M. van Kreveld, and S. Imfield. Finding REMO - Detecting Relative Motion Patterns in Geospatial Lifelines. *Developments in Spatial Data Handling: 11th Int. Sympos. on Spatial Data Handling*, pp 201–215, 2004.

[9] M. Segal and K. Akeley. The design of the opengl graphics interface. Technical report, Silicon Graphics Computer Systems, 1994.

[10] Y. Theodoridis. R-Tree portal. `http://www.rtreeportal.org`.

[11] U. Wilensky. NetLogo. `http://ccl.northwestern.edu/netlogo`.