Eurographics Italian Chapter Conference (2011)
A. F. Abate, M. Nappi, and G. Tortora (Editors)

Fast neighborhood search on polygonal meshes

L. Rocca[1] and N. De Giorgis[1] and D. Panozzo[1] and E. Puppo[1]

[1]Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy

Abstract

We introduce a spatial index to support the fast retrieval of large neighborhoods of points on a polygonal mesh. Our spatial index can be computed efficiently off-line, introducing a negligible overhead over a standard indexed data structure. In retrieving neighborhoods of points on-line, we achieve a speed-up of about one order of magnitude with respect to standard topological traversal, while obtaining much more accurate results than straight 3D range search. We provide quantitative comparisons of results obtained with our method with respect to known techniques.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations I.3.6 [Computer Graphics]: Methodology and techniques—Graphics data structures and data types

1. Introduction

Polygonal meshes are ubiquitous and techniques for analyzing and processing them are relevant in a wide number of applications. Meshes are usually encoded either with simple data structures, which represent just their connectivity, or by topological data structures, which also support tasks such as navigation and editing. Topological data structures involve a higher storage cost and a certain degree of indirection in accessing data (e.g. through pointers) that, in spite of their optimal asymptotic behavior, may slow down traversal operations.

One basic operation, which is necessary in several tasks, is to determine the neighborhood of points on the mesh, i.e., finding the portion of mesh that lies within a certain distance from a given point. While neighborhoods made of just few rings of faces/vertices are usually sufficient for local analysis and processing, multi-scale and global methods often require finding large neighborhoods, which extend over a relevant portion of the mesh [PPR10].

Range search in 3D can be solved efficiently by means of standard spatial indexes [Sam05], without the need to maintain the mesh in a topological data structure, but it does not provide the correct answer to the neighborhood query. Indeed, points that lie within a given 3D Euclidean distance from a given point P do not necessarily lie close to P on the surface (see Figure 1 for an example).

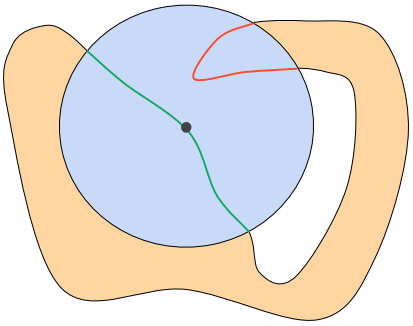

Figure 1: *Range search does not provide the correct answer to a neighborhood query: a ball of a given radius centered at the query point will capture the correct neighborhood (green line), but it may also capture portions of surface that lie arbitrarily far from it (red line).*

A correct solution to the neighborhood query must take into account the surface, not just its embedding space. A rather straightforward idea would be to base the search on geodesic distance, or an approximation of it, measured on the surface. Apart from the heavy computational complexity involved in computing geodesics, also this solution does not give an appropriate answer for most applications. Indeed, the frequency of details on the surface may heavily affect the result. This fact is related with the possible fractal nature of a

boundary of a surface embedded in 3D space and with the scale at which such a surface is considered during computation. For equivalent 2D examples, think for instance about measuring the length of a coastline, which can yield very different results depending on the method used [Man67]; or about shapes that can span a limited area and have an infinite perimeter, like the Koch snowflake [Koc04] (see Figure 2).

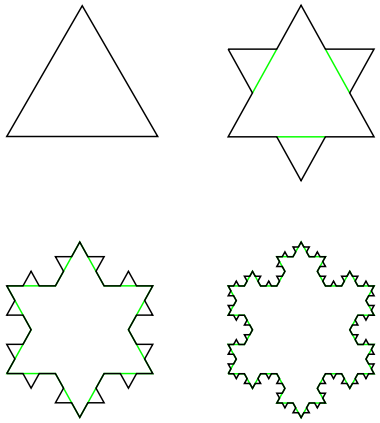

Figure 2: *The first four iterations of the construction of the Koch snowflake: distance along the perimeter can be arbitrarily different depending on the scale at which the shape is represented.*

For most applications, the neighborhood query must take into account both the actual 3D distance of points and surface connectivity: *for a given point P on a mesh M, the neighborhood of P with radius r consists of the set of points that lie within the connected component of M containing P and captured by a ball of radius r centered at P* (i.e., the green line in Figure 1). The connected component allows us to filter out far portions of surface that lie close to P in 3D space, while the 3D ball make the result somehow independent on the scale of representation of the surface.

A neighborhood query defined as above can be answered easily by means of a breadth-first search (BFS) on M, starting at P, and extending up to the boundary of the query ball. Such a search requires to traverse M by adjacencies, hence to encode the mesh with a topological data structure. Apart from the additional cost of maintaining topological information, which involves a non-negligible overhead, BFS may result quite slow for large values of r, thus becoming often the bottleneck of computations that involve neighborhood queries. In fact, the evaluation of topological relations involves a number of random accesses to non-contiguous locations of memory. This makes this kind of operation extremely expensive on modern processors that heavily rely on cache hierarchies to achieve good performances, especially if the involved mesh is large.

In order to provide an efficient solution to the neighborhood query, in this work we propose a spatial index, which takes into account both 3D distance and connectivity on the surface, and does not require maintaining the mesh in a topological data structure. The spatial index consists of a hierarchy of balls of different radii, each containing a connected component of surface, and such that the whole mesh M is spanned by the set of balls belonging to each level of the hierarchy. The spatial index supports a hierarchical search of space that greatly reduces random access to memory, yielding a speed-up of about one order of magnitude with respect to a standard BFS. The reported solution to the query is just approximate, as it may contain also portions of surface that lie outside, but still close, to the desired neighborhood. With respect to a rough answer based on 3D range search, the reported solution is much closer to the exact one, and it is sufficiently accurate for practical purposes.

2. Related work

Standard methods for resolving the neighborhood query rely either on a BFS of the mesh, e.g., through a Dijkstra technique, or on a straight range search in 3D.

Range search is a well studied problem and there exists a large number of methods and spatial data structures to support such task [Sam05]. Most such methods are based on spatial indexes that organize data according to their proximity in the embedding space. In our case, proximity is related to both 3D distance and distance on surface and no known spatial index can be directly adapted to the problem at hand. Among others, spatial indexes based on hierarchies of spheres have been proposed by several authors for different applications (see [Sam05], 4.4.2).

Computation of distances on a surface mesh has also been studied in the literature. In [MMP87], an exact algorithm for computing the geodesic distance from a given point to all other points of a mesh has been presented. Unfortunately, it has a $O(n^2)$ computational complexity and $O(n^2 \log n)$ space complexity, thus resulting prohibitive for relatively large meshes. In [SSK*05] a faster implementation of the same method, which uses approximate distance evaluation based on Dijkstra on the edge network, was proposed. In spite of a relevant speedup, also this version is too cumbersome to be used online. Moreover, an offline computation of geodesic distances for all-vs-all vertices of a mesh would involve quadratic storage cost. The Fast Marching algorithm [Set99] provides an approximation of the geodesic by solving the Eikonal equation. The level set of the solution can be seen as a front advancing with constant speed and can be used as the distance between a set of starting points. The complexity of this algorithm for computing the distance from one point to all the others is $O(n \log^2 n)$, and it requires to navigate the mesh, thus resulting necessarily slower than simple BFS.

The biharmonic distance defined in [LRF10] provides an

approximation of the geodesic that is based on an embedding of points in a high dimensional space of dimension d. This embedding can be computed efficiently off-line and distance on the surface between a pair of points corresponds to Euclidean distance in the embedding, which requires a storage cost of $O(nd)$. Therefore, in principle, any spatial index for high dimensional data could efficiently support online range queries on the embedding space (see [Sam05], 4). Unfortunately, the value of d is quite high, thus yielding an overhead of at least two orders of magnitude over the cost of storing the plain mesh only, thus making this approach unpractical.

3. Spatial index

The main idea behind the data structure we propose consists in precomputing and storing multiple balls of connected vertices, called *clusters*. These clusters are organized in a hierarchy that connects them to their counterparts at different scales, they share an adjacency relationship with their neighbours at the same scale and each of them records the center and the radius of the sphere enclosing the vertices it contains. The main goal is to easily find precomputed groups of vertices contiguosly stored in memory that contain the desired neighborhood.

Each level of the hierarchy contains clusters belonging to the same scale and spans the whole mesh with a perfect covering, i.e. each vertex is referenced by one and only one cluster inside it.

The first level is based on the vertex to vertex adjacency relationship of the original mesh. It is useful only to compute the rest of the spatial index and it can be safely discarded afterwards, while each subsequent level is built on the previous one until a level containing a single cluster spanning all vertices remains. In practice, we will see that it is not necessary to go that far and a small number of carefully chosen levels is sufficient for all our purposes.

In the rest of this section, we will give some basic definitions (3.1), describe how to compute the spatial index (3.2) and how to perform a query on it (3.3).

3.1. Definitions

Cluster. A cluster contains a set of references to vertices that form a connected component in the original mesh, the center and the radius of the sphere that encloses all its vertices and references to its parent in the hierarchy, its children and its siblings.

Base Cluster. A cluster that contains only one vertex v and has no children. Its enclosing sphere is centered at v and has null radius.

Parent. The parent of a cluster C at level l contains at least all vertices inside C and is located at level $l+1$.

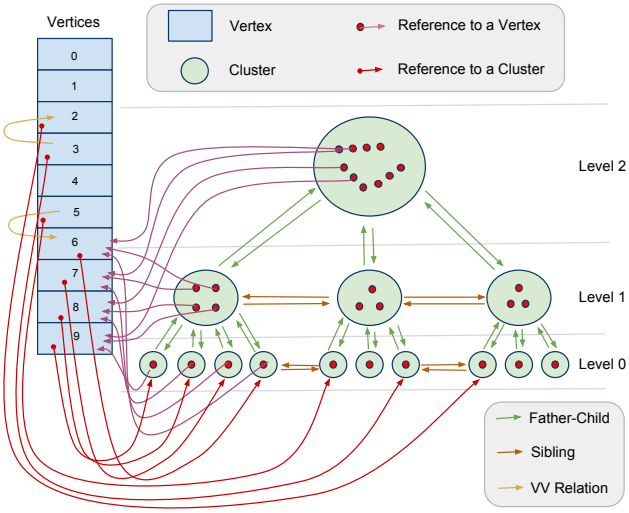

Figure 3: *The spatial index.*

Child. The child of a cluster C at level l contains a subset of the vertices of C and is located at level $l-1$. The union of the vertices inside all children of a given cluster corresponds to the set of vertices inside that cluster.

Base Sibling. Two base clusters containing vertics v and w, respectively, are sibling if and only if v and w are adjacent on M.

Sibling. A cluster C at level l is sibling of another cluster S at level l when one child of C is sibling of a child of S.

Level. A set of clusters. The sets of vertices they contain are disjointed and their union covers all vertices in the original mesh. Clusters inside a level l are siblings to each other and have children at level $l-1$ and parents at level $l+1$.

Spatial Index. A set of N levels, from 0 to $N-1$. A sketch of the whole structure is depicted in figure 3.

3.2. Computing the spatial index

The spatial index is built starting at level 0. This level is composed of *base clusters* and its computation is straightforward, consisting of a traversal of all vertices in the mesh and a visit of the vertex-to-vertex relation in order to build the *base sibling* relationship. This level is the first input to a procedure that computes every other level from the preceding one, which is divided in four main phases:

1. *Create.* For each cluster C inside level k a new cluster of level $k+1$ is created, initialized with C and all its siblings as children. At this stage, everything else is still empty. The newly created clusters are all inserted in a priority

queue Q, ordered on the number of children each cluster has.

2. *Select.* The goal of this procedure is to modify and delete the clusters inside Q until each cluster at level k is a child of one and only one cluster at level $k+1$ inside Q. Such a computation is an instance of the *set cover problem*, which is known to be NP-complete. We therefore employ a greedy algorithm, see algorithm 1. Note that there could be different ways of ordering the priority queue; precomputing the average number of children of the clusters it contains and ordering it based on the distance from the average was empirically found to give a good solution.
3. *Link and sphere.* Now that the new clusters cover every cluster of the previous level in a non-overlapping fashion, for each cluster $C \in Q$ we set its vertices as the union of its children vertices, the parent of its children as itself, and as siblings every cluster different from itself that the siblings of its children have as parent. Then, we compute the center and radius of the minimum bounding sphere of points inside C. — *gg qui citare miniball* —
4. *Fusion.* At this stage, the new level we just created fulfills the definitions given in Subsection 3.1 and it could be used for neighborhood queries. In practice, we found that there may be a relevant number of small clusters even at high level of the hierarchy, which have an impact on performance. We therefore developed a fusion procedure that deletes clusters smaller than a certain threshold, which grows with the scale as we climb up the hierarchy. Vertices belonging to the donor cluster are distributed to the nearest siblings, and suitable care is used to reassign inside the involved clusters the correct parent, children and sibling relationships they share with each other in the current, previous and next level. — *gg qui si poteva anche mettere lo pseudocodice ma mi pareva gia' troppo cosi'! magari se alla fine c'e' ancora spazio...*

This procedure could continue until a level consisting of only one cluster spanning the whole mesh is created, but this is not necessary (see 4.1)

3.3. Performing a query

The input consists of a vertex v, chosen as a starting point, and a distance d. The search sphere S is the sphere of radius d centered at v. A query is divided in three phases:

1. *Climb up.* First of all, the cluster inside the lowest level that contains v is retrieved. Then, the query follows the parent relationship going up through the levels, until a cluster C at the right scale is found. Note that C contains v by construction.
2. *Breadth first search.* A BFS using the sibling relationship is performed starting from C until clusters whose enclosing sphere intersects S are found. We call the set of such clusters SC.
3. *Range search.* The neighborhood of v is the set of all the vertices of SC that lie inside the search sphere S.

Algorithm 1 select (PriorityQueue Q, Level newLevel)

```
1:  while Q ≠ empty do
2:    deletedSons ← false
3:    Cluster C ← Q.top()
4:    for CFi ∈ C.sons do
5:      if CFi.visited = true then
6:        C.deleteSon(CFi)
7:        deletedSons ← true
8:      end if
9:    end for
10:   if deletedSons = true then
11:     if C.sons ≠ empty then
12:       Q.insert(C)
13:     end if
14:   else
15:     newLevel.addCluster(C)
16:     for CFi ∈ C.sons do
17:       CFi.visited ← true;
18:     end for
19:   end if
20: end while
```

The critical point in this procedure is to decide when a cluster at the right scale is found. Since a level contains clusters of varying dimensions up to a certain point, we decided to precompute for each level the average radius ar of the clusters it contains. We stop climbing up levels when $ar \geq d/f$, where factor f controls the balance between the BFS and the range search: if $f < 1$, the query goes higher in the spatial index, the BFS will be shorter and there will be more vertices to filter; if $f > 1$, the query stays lower in the spatial index, the BFS will propagate along more clusters, but there will be less vertices to filter. After experimenting a bit, we empirically found $f = 10$ to give the best results with all datasets.

Note that the query output is approximated, compared to a BFS on the original mesh connectivity. In fact, while the vertices inside SC span a connected portion of M, this component may exceed the search sphere S, and there is no guarantee that it remains connected after the vertices outside S have been discarded. Theoretically, clusters could give exactly the same problem as simple range search (figure 1) - and in fact they sometimes do. The key difference is the scale at which this happens: on the whole mesh, a point inside the range search could be arbitrarily far along the surface from the query point, while in a cluster this is limited to the small connected component it contains and the distance it spans. For a sufficiently large value of f, this implies that the extra portions of M captured by the search will not lie far from the exact solution of the query.

In practice, approximation errors are negligible results for real world multi-scale applications (Section 4). Also note that the parameter f tunes not only performance but also ac-

mesh	#vertices	mesh with topology	plain mesh	spatial index	building time
Cat	27894	1.703MB	0.958MB	0.526MB	0.258s
Victoria	45659	2.787MB	1.568MB	0.866MB	0.491s
Happy	543652	33.182MB	18.665MB	10.163MB	6.652s
Raptor	1000080	61.040MB	34.335MB	18.824MB	12.589s

Table 1: *Datasets used, how much space they use and how long it takes to compute the spatial index on them.*

curacy. As this value grows, clusters found in a given query become smaller and the chance of getting errors decreases. In this regard the value of $f = 10$ guarantees good accuracy too.

4. Results

In this section we present the results obtained with our implementation of the proposed spatial index. Experiments were run on a PC with a 2.67Ghz Core i5 processor equipped with 8Gb of memory, using a single core.

We used four meshes as datasets: the Cat, Victoria, Happy Buddha and Raptor. The first two meshes were chosen because of their shapes rich of protrusions, allowing us to test the correctness of results; while the other two meshes where selected in order to test performances on meshes of reasonably large dimensions.

4.1. Space usage

Early experiments showed that the performance of large queries was unaffected, even when computing just the first five levels of the hierarchy, and keeping only the last three. Therefore, the first two levels (the useless base level and its successor, containing clusters that span 10 vertices at most) are discarded and only three more levels are computed. It is our intuition that huge meshes which can barely fit in the main memory of today's machines could use another level at most.

With these optimizations, the whole structure, comprehensive of an additional reference to the lowest level clusters inside the vertices representation, costs about half than the input mesh (vertices and faces maintained in an indexed structure), as shown in table 1. The two data structures together (mesh plus index) cost less than maintaining the same mesh with its topology (indexed structure with adjacencies).

4.2. Pre-processing time

The time it takes to build the spatial index is shown in table 1. Given the time it takes to run a single query, it is easy to see that if an application needs to perform neighborhood search for each vertex in the mesh, pre-processing time results negligible.

For example, on the raptor dataset, the cost of a single query with our spatial index with radius 65 times the average edge is 0.26ms. This means that it would cost 260s to run it on every vertex. That would take 272s if pre-processing time is added. As a comparison, the cost of running all such queries with standard BFS is of 2250s for a BFS (2.25s for a single query).

4.3. Querying a neighborhood

In order to understand how well queries perform on the spatial index, we compared them to a BFS search on the mesh topology, and to a 3D range search. The latter has been implemented as a simple scan of all vertices of the mesh, without any further optimization. It is used mostly to show how range search gets wrong results with respect to BFS, but it also give an estimate of the cost of traversing all vertices. Of course, range search could be made much faster by adopting any standard spatial index, but results would be the same. Yet, even with simple linear scan it is faster than BFS for relatively large neighborhoods.

Results are shown in Table 2 and Figure 4. The range of the queries went from very small (two times the average edge length in the mesh) to huge (ninety times the average edge). We compare the different methods in terms of correctness and speed.

Correctness. The BFS always returns the correct answer. A expected, the cluster query tends to behave more like a BFS than the range search. The graphs of the number of vertices found for the query on the cat's tail and victoria's hand show that results are similar to the ones obtained with a BFS. Both make a big jump when the arm of victoria and the tail of the cat end, the only difference being that for the cluster query this happens at a scale which is slightly smaller than that of the BFS, but still quite large in absolute terms. On the contrary, the range search starts capturing extra (wrong) vertices at a much smaller scale.

Speed. The graphs show a strong boost for our method. The running times of both the cluster query and the BFS are functions of the number of output vertices, but the cluster query runs an order of magnitude faster at almost all scales. The BFS remains competitive just for very small radii of search (up to about 3 times the average edge length). It is interesting to notice that for relatively large radii (with respect to the size of the mesh) the BFS soon becomes slower even

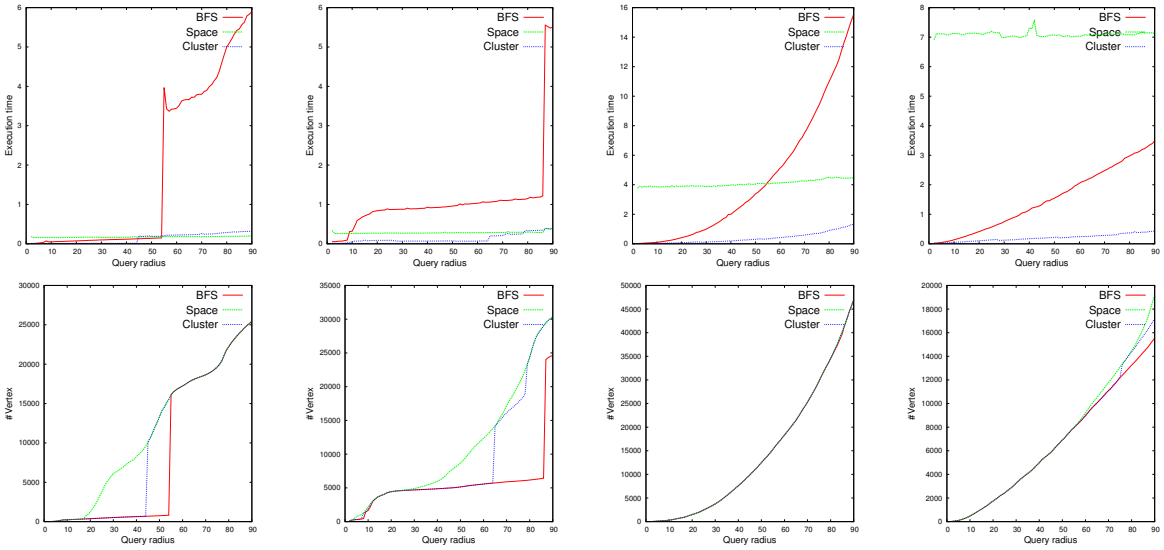

Table 2: *Query performances in the different datasets: (a) Cat, (b) Victoria, (c) Happy, (d) Raptor. The query radius goes from two times the average edge to ninety times the average edge. In the first row, the time elapsed; in the second row the number of vertices found.*

than range search based on linear scan. Only in the case of the raptor the BFS remains competitive with respect to range search, because the mesh is so big that the query always spans a relatively small portion of it, even with the largest radius.

Interestingly, by profiling our query (Subsection 3.3) over the whole benchmark, we found that the *range search* phase takes up the 85% of the running time, whereas the *climbing up* phase has a negligibile cost, and the *breadth first search* phase takes up the rest of the time. This suggests that performance could be further improved with small additional space overhead, by endowing each cluster with a standard spatial index supporting 3D range search.

5. Conclusions

We have introduced a spatial index that is specifically tailored to support neighborhood queries on polygonal meshes. The spatial index can be coupled with a standard indexed structure for the mesh, yielding a total storage cost smaller than maintaining a topological data structure, and it can be built off-line efficiently. The performance of on-line queries beats by one order of magnitude a standard BFS, while returning an approximated result that is much closer to the exact one than that obtained with a simple 3D range search. Performance could be further improved by combining our hierarchy of clusters with a standard spatial index inside each cluster.

We believe that our spatial index can be successfully applied in a number of geometry processing tasks, especially those connected with global or multi-scale processing. We are planning to adopt this data structure to improve the performance of our multi-scale method for curvature and crease estimation [PPR10].

References

[Koc04] KOCH H. V.: Sur une courbe continue sans tangente, obtenue par une construction geometrique elementaire. *Arkiv for Matematik 1* (1904), 681–704. 2

[LRF10] LIPMAN Y., RUSTAMOV R. M., FUNKHOUSER T. A.: Biharmonic distance. *ACM Trans. Graph. 29* (July 2010), 27:1–27:11. 2

[Man67] MANDELBROT B.: How long is the coast of britain? statistical self-similarity and fractional dimension. *Science 156*, 3775 (1967), 636–638. 2

[MMP87] MITCHELL J. S. B., MOUNT D. M., PAPADIMITRIOU C. H.: The discrete geodesic problem. *SIAM J. Comput. 16* (August 1987), 647–668. 2

[PPR10] PANOZZO D., PUPPO E., ROCCA L.: Efficient multi-scale curvature and crease estimation. In *Proceedings of Computer Graphics, Computer Vision and Mathematics* (Brno, Czech Rapubic, September 7-10 2010). 1

[Sam05] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. 1, 2, 3

[Set99] SETHIAN J.: *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*. Cambridge monographs on applied and computational mathematics. Cambridge University Press, 1999. 2

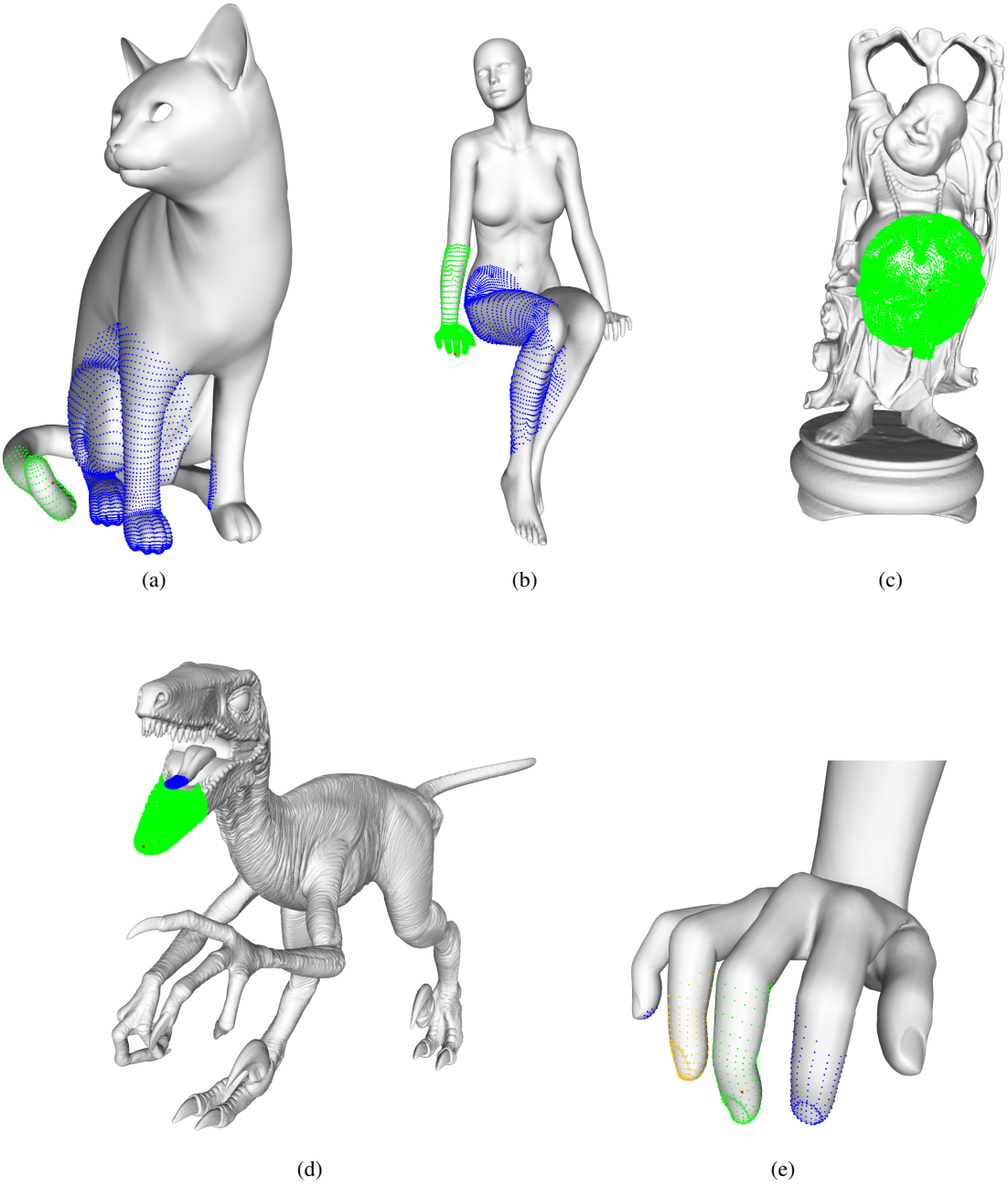

Figure 4: *Examples of queries: (a) Cat, radius = 40; (b) Victoria, radius = 50; (c) Happy, radius = 85; (d) Raptor, radius = 70, (e) Victoria's hand, radius = 7.3. The input vertex, a red dot, is the same used in the graphs in figure 2. The BFS result is shown in green, the cluster search in yellow and the range search in blue. When results coincide, green vertices cover yellow ones, which in turn cover the blue ones.*

[SSK*05] SURAZHSKY V., SURAZHSKY T., KIRSANOV D., GORTLER S. J., HOPPE H.: Fast exact and approximate geodesics on meshes. *ACM Trans. Graph. 24* (July 2005), 553–560. 2