Seventh Hungarian Conference on Computer Graphics and Geometry, Budapest, 2014

Learning Shape Matching for Augmented Reality

Marton Szemenyei[1] and Ferenc Vajda[1]

[1] Department of Control Engineering and Informatics, Budapest University of Technology and Economics, Budapest, Hungary

Abstract

Tasks involving 3D shape are relatively common in Virtual and Augmented Reality systems. In this paper we present a learning shape matching method which we will use to replace real-world objects with virtual ones that are similar in shape. We will use this algorithm in an Adaptive Augmented Reality system that will fill a real-world scene with virtual objects so that the objects would fit into the scene. In our method, we first segment the 3D scene into primitive shapes, then we construct a graph encoding the properties and the relations of the shapes. We then use a graph kernel function to make our problem compatible with common learning methods, such as support vector machines. We have also tested the method using our own dataset consisting of stereo images, and have found satisfying results.

Categories and Subject Descriptors (according to ACM CCS): I.4.8 [Image Processing And Computer Vision]: Object recognition

1. Introduction

Tangible User Interfaces[1] and Tangible Augmented Reality[2] has been an area of intense research in the last decade. The goal of these systems is to allow users to interact with virtual objects by manipulating real objects, which results in a natural interface. Dynamic Shader Lamps[3] and Dynamic Object Texturing[4] are two noteworthy examples, both applying virtual textures on real objects. Other important examples are Tangible Bits[1], and the Virtual Round Table (VRT)[5], which uses real objects as placeholders for virtual ones.

Our system is somewhat similar to VRT: Using a stereo camera pair, we aim to create an Adaptive Augmented Reality system that finds the most fitting place for virtual objects in an unknown environment.

Our goal with this paper is to present our solution to achieving a logical pairing of virtual and real objects. The main criterion of the pairing method is that the real and virtual objects must have similar physical properties, so that the user could effortlessly manipulate the real object as he would do with the virtual one. However, many important physical properties, like mass, roughness of surface cannot be measured visually, therefore we must restrict our collection of determining physical properties to shape and size.

It logically follows, that the object pairing method needs to be based on shape matching, or shape similarity. Nonetheless, there is a large number of shape matching methods. Our aim for this work was to create a shape similarity method that requires no a priori information on the virtual objects, but can learn from instances of labeled real objects. Also, we wanted to make no assumptions on whether large-scale or small-scale shape attributes are important for pairing, therefore we attempted to encode all information on the shape of an object, and allowed the learning algorithm to make that decision.

Figure 1: *A stereo image pair of an example scene*

Our method first converts the stereo image pair (Figure 1) into a 3D point cloud for further processing. Then, we use a modified RANSAC algorithm by Schnabel[6] to seg-

ment the point cloud into primitive shapes, like planes, cylinder, spheres, etc. Then we construct a graph from these primitives, where nodes represent the primitives themselves, while edges encode the spatial relationship between the nodes. Our algorithm then uses our modified random walk graph kernel[7] to insert the problem into a support vector machine.

2. Related Work

In this section we will give an overview of the work related to ours. In the first part of this chapter we will discuss shape matching methods. The second part will be devoted to the problem of learning on graphs.

Shape matching is one of the basic problems of computer vision and Augmented Reality. There are a number of methods available, like the RANSAC[11] algorithm, however, for this method a reference model of the object needs to be given. Numerous learning shape matching methods are available as well, that use 3D shape distributions[9], or features[10] to learn, therefore they don't need reference models. However, in order to use these methods a separate segmentation step is needed first.

One other method, created by Schnabel[6] uses a different approach. They created a modified RANSAC algorithm, that is able to find multiple instances of several different primitive shapes in a point cloud efficiently, and they use it to segment the point cloud. From that, they construct a topology graph to encode neighborhood relations of the primitives. Finally, they use brute-force graph matching to compare the graph to a reference model.[8] In our method (see Section 3.) we replaced the brute-force matching part with a learning algorithm in order to allow more flexibility and avoid depending on a priori information.

Learning on graphs is an important task in machine learning. Still, it is a difficult problem, since most machine learning algorithms work well with vectors, but not with structured objects, like graphs. The difficulty is that the simplest ways to assemble a vector from a graph is not invariant to the ordering of the graph nodes.[12] Additionally, graphs of different sizes will result in vectors of different lengths, which adds further technical difficulty.

One of the possible solutions is described in detail by White[12] who examines the different spectral representations of graphs and combines them to create a generative part-based model for learning. However, this representation only makes the feature vector constructed from the graph semi-invariant to the node order, and alignment step is still needed for matching.[12]

Another possible solution is using a kernel function. A kernel function is a positive semi-definite, symmetric function, that returns the similarity of two arguments.[7] One of the basic graph kernels is the random walk kernel, which measures two graphs by performing random walks along its edges. The similarity score of the two graphs is derived from the probability of performing the same walk on the two graphs simultaneously. The score is calculated as the following:[7]

$$K = \sum_{i=1}^{N} w(i)\mathbf{e}^T A^i \mathbf{s} \tag{1}$$

$$w(i) = e^{-\xi i} \tag{2}$$

Where A is the adjacency matrix of the direct product of the two graphs, $\boldsymbol{s}$ and $\boldsymbol{e}$ are the probabilities of starting and ending a walk on a given node, and $w(i)$ is the relative weight of i long walks, ξ is a hyperparameter, and N is the maximum length of the walks considered. It follows from the equation that the random walk kernel is invariant to node ordering. Moreover, since the kernel calculates the direct product of the graphs, it can compare graphs of different sizes.

3. Learning Shape Matching

In this section we will describe the shape matching algorithm in detail. As mentioned earlier, there are two major parts of the algorithm: The first is the process of assembling the graph, while the second is the random walk kernel implementation. In the third part of this section we will describe our method for training and testing the algorithm.

3.1. Assembling the Graph

The first step in constructing the graph was to use the modified RANSAC algorithm of Schnabel to segment the point cloud into primitive shapes, for which we used the implementation available online. One difficulty is that with point clouds that have noisy surfaces, and are reconstructed from a single range image, the RANSAC algorithm frequently miscategorizes the primitives. This means, that the learning algorithm has to be able to learn the most typical mistakes.

The next step is building a graph from the primitive shapes (an example graph is shown in Figure 2). The nodes in this graph will be the primitives that the segmenting algorithm has found. However, each of these nodes has a number of additional features that describe the primitive shape. Moreover, different types of primitives always have a different set of features, which would mean that the nodes of the graph have feature vectors of different sizes, and meaning. To avoid this complication we use just one feature vector, which is a concatenation of the unique vectors of each primitive type. The undefined values of the vector (e.g. if the primitive is a cylinder, then the features of a plane, sphere, etc.) are set to zero. In our implementation, we considered the following features of the primitive shapes:

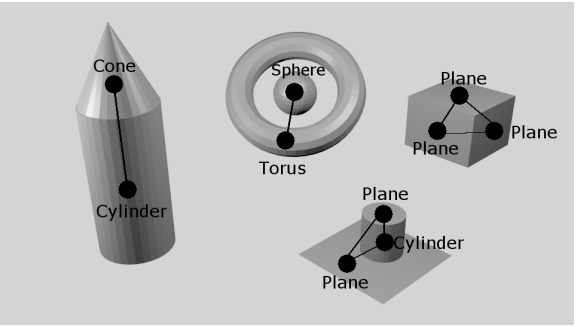

Figure 2: *The graph constructed from primitive shapes (only significant edges are drawn)*

Plane:	Area Diameter Bounding Box Area
Sphere	Radius
Cone	Radius Height
Cylinder	Radius Height Angle
Torus	Inner Radius Body Radius

In addition to that each nodes have a coordinate system, which is defined by a special point of the primitive, and a direction (usually the direction of an axis, or a surface normal). The only exception is the sphere, which has no unambiguous direction. However, this means that we can define a transformation between the primitive shapes i.e. the nodes of the graph as the transformation between their coordinate systems. These transformations will be the edges of the graph, encoded by a quaternion and a translation vector.

This way we will always construct a full graph, since the information whether two nodes are neighbors will be encoded in the features of the edge between them. The main advantage of this is, that the adjacency property between two nodes becomes a real value instead a binary one, which means that we are unlikely to have problems later caused by missing edges, resulting in a more robust solution.

3.2. Random Walk Kernel

The next step is to insert this graph into a random walk graph kernel. However, we need to make a slight modification to the original algorithm, because the random walk kernel originally works with weighted graphs, where the weights are real numbers, not vectors. To achieve this, we need to define a kernel function between nodes, and between edges. Then, we need to rewrite the random walk kernel to use these subkernels wherever the original method requires the product of two node or edge weights.

Thus, the kernel function between the edges will be used to calculate the Adjacency matrix of the direct product graph. However, the edge kernel will only use two features of an edge between two nodes: The $\cos(\alpha/2)$ part of the quaternion, and the squared distance between the nodes. The reason for excluding the rest is that the exact orientation of a node is ambiguous, and we may introduce significant noise into the algorithm. To get the exact form of the edge kernel we are using a variation of the RBF kernel:

$$E_v = e^{(-\gamma\cdot\Delta\omega-\delta\cdot\Delta d-\mu\cdot d)} \tag{3}$$

Where γ, δ and μ are the hyperparameters of the edge kernel, $\Delta\omega$ is the absolute difference of the cosine of the angles, Δd is the absolute difference of the squared distances of the two edges, d is the average of the squared distances. We have introduced the average distance into the kernel because nodes that are close in space are likely to be the part of the same object, whereas distant nodes are not. Therefore, we chose a measure of edge similarity which punishes long edges regardless of their similarity.

We will also construct the vector containing the starting and ending probabilities of a walk using a kernel function defined between the nodes: To create the vector, we will compare every node of graph A to every node of graph B, thus assigning a weight to every node of the direct product graph. This means that we will be more likely to start a walk on nodes that are similar, than on ones that are dissimilar. But first, we need to overcome the problem that we have constructed a feature vector for nodes from features that differ in scale significantly. There are area-type, distance-type and angle-type features, which means that a simple kernel functions, like polynomial of RBF are not sufficient. To tackle this problem, we calculate an inverse goodness value from the features, by dividing the absolute difference of the two feature values by their average. This normalizes all factors into the [0-1] range. Then, we applied an RBF kernel to the inverse goodness value as well.

$$W = \sum_{i=1}^{M} \frac{|a_i - b_i|}{a_i + b_i} \tag{4}$$

$$N_v = e^{-\theta\cdot W} \tag{5}$$

Where θ is another hyperparameter. The random walk kernel has two more hyperparameters: The first one is N, the maximum length of walks considered, while the other is the slope of the exponential weight function $w(i)$. There is one more modification that we have introduced to the kernel. We have observed that the score of the kernel function depended significantly on the length of the graphs, which was undesired. In order to compensate for this, we have modified the final

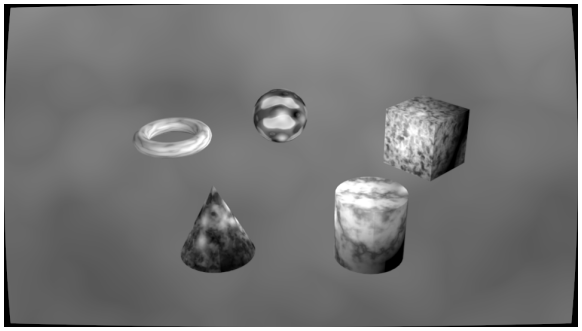

Figure 3: *The example scene created in Blender*

score of the kernel to be:

$$K_f = \frac{K}{(m \cdot n)^{\varepsilon}} \tag{6}$$

Where m and n are the lengths of the two graphs being compared, and ε is the seventh hyperparameter. There is the possibility of using an exponential or logarithmic scaling of the graph sizes, but the polynomial variant gave us the best results.

3.3. Training and Testing

To train and test the algorithm, we have created a virtual scene in Blender (shown in Figure 3) consisting of five objects: A cube, a torus, a cylinder, a sphere and a cone. We have used a virtual scene for testing, because our focus was to test the performance of the learning algorithm, and we didn't want camera-specific problems, etc. to interfere. We have taken 200 stereo images of the scene from different angles, and labeled the set with bounding boxes. For each shape we created 400 graphs: a positive and a negative example for each image. In order to find the optimal values of the seven hyperparameters of the algorithm we have implemented a cross-validation training method using the SVM implementation of MATLAB. We have used 280 training examples 60 validation and 60 test examples. Since optimizing with seven parameters would take a vast amount of time, we made a very rough pre-optimization step using only a fraction of our data, which allowed us to narrow down the range in which the optimum might be.

4. Results

The results of our cross-validation and testing are shown in Table 1. As seen there, we were able to achieve relatively low error rates, which means that the learning has been successful. Moreover, the hyperparameters required to achieve near optimal performance are close for all five shapes, which means that the process of tuning hyperparameters won't have to be redone when the algorithm has to learn to recognize more shapes. The best sets of hyperparameters we have found during cross-validation are shown in Table 2.

Primitive	Training	Validation	Test
Cube	7.5%	11.66%	15%
Sphere	2.14%	3.33%	8.33%
Cone	4.28%	8.33%	10%
Cylinder	4.28%	6.66%	11.66%
Torus	1.78%	6.66%	6.66%

Table 1: *The Error Rate of the Algorithm*

	θ	γ	δ	μ	ε	N	ξ
Cone & Sphere	0.01	0.01	0.01	10^{-4}	2.5	10	13
Others	0.01	0.01	0.01	10^{-4}	3.5	10	16

Table 2: *The Found Parameters*

5. Conclusions

In this paper we have introduced and described a method for shape matching that uses a primitive graph and a kernel function to learn. We have shown that the algorithm is capable of achieving satisfying results, even in a dataset with many occlusions.

In the future we are considering to introduce further features that will hopefully improve the accuracy of the method. We are also planning to modify the algorithm, so that it would be capable of localizing the detected shape in a large scene, since this will be essential if this method is to be used in our Augmented Reality system.

References

1. Hiroshi Ishii, Brygg Ullmer, Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms, *Proceedings of CHI*, 1997.

2. Mark Billinghurst, Hirokazu Kato, Ivan Poupyrev, Tangible Augmented Reality, 2001.

3. Deepak Bandyopadhyay, Ramesh Raskar, Henry Fuchs, Dynamic Shader Lamps : Painting on Movable Objects, *The Second IEEE and ACM International Symposium on Augmented Reality*, 2001.

4. Kresimir Matkovic, Thomas Psik, Ina Wagner, Denis Gracanin, Dynamic Texturing of Real Objects in an

Augmented Reality System *Proceeding of IEEE Virtual Reality 2005*,pp. 245–248, 2005.

5. Wolfgang Broll, Eckhard Meier, Thomas Schardt, The Virtual Round Table - a Collaborative Augmented Multi-User Environment, *Proc. of the ACM Collaborative Virtual Environments*, pp. 39–46, 2000.
6. Ruwen Schnabel, Roland Wahl, Reinhard Klein, Efficient RANSAC for Point-Cloud Shape Detection, *Computer Graphics Forum*, **26**(2):214–226, 2007.
7. S.V.N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, Karsten M. Borgwardt, Graph Kernels, *Journal of Machine Learning Research*, pp. 1201–1242, 2010.
8. R. Schnabel, R. Wahl, R. Wessel, R. Klein, Shape Recognition in 3D Point Clouds, *Computer Graphics Technical Report*, 2007.
9. Robert Osada, Thomas Funkhouser, Bernard Chazelle, David Dobkin, Matching 3D Models with Shape Distributions, *International Conference on Shape Modeling and Applications*, pp. 154–166, 2001.
10. Aleksey Golovinskiy, Vladimir G. Kim, Thomas Funkhouser, Shape-based Recognition of 3D Point Clouds in Urban Environments, *International Conference on Computer Vision*, 2009.
11. Martin A. Fischler, Robert C. Bolles, Random Sample Consensus: A Paradigm for Model Fitting with Apphcatlons to Image Analysis and Automated Cartography, *ACM Graphics and Image Processing*, **24**(6):381–395, 1981.
12. David H. White, Generative Models for Graphs, 2009.