Harris Operator in 3D

Project Report: Software Engineering

6 January, 2019

Abstract

Using Qt Creator, OpenGL, GSL and an established research paper as reference, the process of applying the Harris Operator in a 3D mesh has been recreated and implemented over the course of this project. All faces and vertexes of a mesh with triangular connectivity are rendered, the k-ring neighborhood is generated, the Harris response for each vertex is calculated and the final interest points are computed. The results are displayed through a basic graphical user interface. For a range of different meshes, the interest points generated match the expected results, i.e. the corners of the model, and by doing so, the target of this project has been accomplished.

Pierpaolo Vendittelli

Anindya Shaha

Khrystyna Faryna

MSc. Medical Imaging and Applications (MAIA)
Centre Universitaire Condorcet
Université de Bourgogne

Contents

1 Introduction

1.1 Objectives

The project strives to recreate the extension of the Harris operator in 3D meshes in C++, as proposed by Ivan Sipiran and Benjamin Bustos [1] in 2011, considering the following modifications:

- Replacing the role of the Computational Geometry Algorithms Library (CGAL) in the source code using our own **class representation for 3D meshes**, that can efficiently compute various neighborhoods (points, faces), represent relevant edges, as well as any other additional information.
- Computing the **k-ring neighborhood** and the **Harris response** for each vertex in a 3D model, along with the corresponding interest points across its surface.
- Rendering all 3D elements (object, neighborhood, edges, interest points) using **OpenGL**.
- Displaying the final results in a **QT Graphical User Interface (GUI)**.

1.2 Preamble Repositories

The following online repositories have been set up to facilitate communication and the cross-coordination of resources:

- **Trello** (Project Management): `https://bit.ly/2z45vpD`
- **Bitbucket** (Code Repository): `http://bit.ly/2Rs6KJO`
- **Google Drive** (Cloud Storage): `https://bit.ly/2FddTZq`

1.3 Dataset

The dataset used over the course of the project consists of **Object File Format (OFF)** models from the Princeton Shape Benchmark repository [2] and the "Geometry" and "Models" packages from Ryan Holmes [3], available under public domain. It should be noted that only OFF models with 3 vertices per face are used to test k-ring neighbourhood computation using triangular connectivity.

2 Methodology

2.1 Reworking Original Source Code

Goal

The original source code, published by Ivan Sipiran and Benjamin Bustos, incorporated the option to generate 3 different types of neighborhood (**ring**, **spatial**, **adaptive**). To achieve the same, it utilized CGAL for handling adaptive/spatial meshes and the **GNU Scientific Library (GSL)** for mathematical computations (finding eigenvectors/eigenvalues, Principal Component Analysis, etc.). In this stage of the project, the aim was to prepare a functional copy of the code in order to understand the working principle and have a clear idea of the expected results for our own model.

Proceedings

Hence, the source code was thoroughly studied and all elements pertaining to adaptive/spatial functionality, including CGAL, was removed. **GSL for Windows (GnuWin32)** [3] was installed and integrated into the QT .pro file. However, a segmentation fault still persisted. Upon debugging, it was found that the error resulted from a command (`Vertex.cpp`: *Line 29:* `Q.push(v1))` that pushed back the queue in an instance of the vertex class, during the neighborhood generation process. When this line was supressed, the program ran smoothly, but the resulting interest points were naturally compromised.

Results

Although, the original goal was unsuccessful, significant progress was made as follows:

- *detectInterestPoints* function of `HarrisDetector` class was heavily adapted into our final program, to successfully calculate the Harris response of each vertex, in a given model.
- The modified source code could be run on a Linux platform (eg. Ubuntu), without any segmentation faults, where its results and working steps could be tracked and studied.

2.2 3D Render & Display

The way we render the mesh is simple. We have two data structures in our $MyGLWidget.h$: $Vertices$, $Faces$. When we read the `.OFF` file, we copy the coordinates of the vertices into an instance of $Vertices$ and the faces into an instance of $Faces$. After that, we use the $Draw()$ function to render the mesh. We are able to draw triangles (faces), using $GL\_TRIANGLES$, and display the interest/candidate points using GL_POINTS. We are able to draw lines (edges) for displaying the connected neighbourhood of a vertex using GL_LINES.

Additionally, when we want to display a mesh, we need to calculate the normal vector for each face of the polygon we are rendering and apply it via $glNormal3f$. The normalization process gives us the direction in which each face is "pointing". The lighting/contrast we need in order to see the object and differentiate contours, is affected by this vector.

The function $glScalef$ has the function of zooming (or defining the scale of visualization). As we don't have a "zoom in/out" slider (or event), we calculated a scaling factor which is fixed and sufficient for displaying every mesh computed.

Algorithm 1 Rendering Mesh

Require: Vertex Indexes for Each Face
1: **for** each Face **do**
2: Generate the Normal Vector
3: $glBegin(GL\_TRIANGLES)$
4: $glNormal3f(nx, ny, nz)$;
5: $glVertex3f(Vertex(x, y, z)$
6: $glVertex3f(Vertex(x, y, z)$
7: $glVertex3f(Vertex(x, y, z)$
8: $glEnd()$
9: **end for**

Algorithm 2 Rendering Edges

Require: Vertex Indexes for k-Ring
1: **for** each Vertex in k-Ring **do**
2: Find its Adjacent Points
3: **for** each Adjacent Point **do**
4: **if** Adjacent Point $\subset$ k-Ring **then**
5: $glBegin(GL\_LINES)$
6: $glVertex3f(Vertex(x, y, z))$;
7: $glVertex3f(AdjacentPoint(x, y, z))$;
8: $glEnd()$
9: **end if**
10: **end for**
11: **end for**

2.3 Ring Neighborhood

The 1st ring neighborhood of a point is comprised of its direct adjacent points. For the 2nd ring neighborhood, we can think of it as the 1st ring members, plus the 1st ring neighborhood of each 1st ring member (assuming that all duplicate members are removed). In other words, if we are to compute the k-ring neighborhood of a vertex, we simply need to compute Ring-1 for each (k-1)-ring member. To compute the same, it is convenient to calculate the adjacent points of every vertex in a given mesh and store their indexes alongside the vertex coordinates, in a dedicated vertex structure, beforehand. This is the core purpose of our $getNeighborhood_Prime$ function.

Algorithm 3 $getNeighborhood\_Prime$
: Computes Ring-1 (Adjacent Points) for every Vertex

Require: Vertex Indexes in the Mesh
Require: Face Indexes in the Mesh
1: $intEdgesinRing = 0$;
2: **for** each Vertex i in the Mesh **do**
3: **for** each Face j in the Mesh **do**
4: **if** Vertex 1, 2 or 3 of Face j == Vertex i **then**
5: $vertex[i].adjacentpts.insert(face[j].v1)$;
6: $vertex[i].adjacentpts.insert(face[j].v2)$;
7: $vertex[i].adjacentpts.insert(face[j].v3)$;
8: $EdgesinRing++$;
9: **end if**
10: **end for**
11: **end for**

Using this information, we can now compute the k-Ring for any/every given vertex in a mesh. To verify our results, we use the rendering platform and the basic GUI from the previous stage to visualize the neighborhood. This has been illustrated in Figure 1.

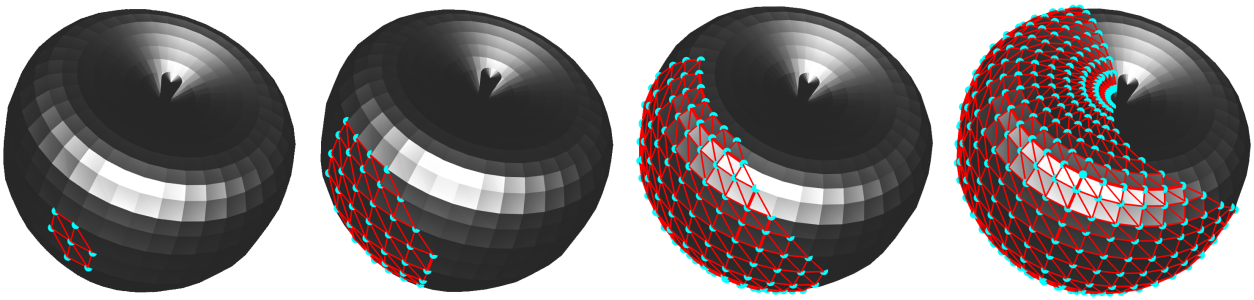


Figure 1: Ring-1 (a), Ring-3 (b), Ring-6 (c) and Ring-12 (d) for Vertex #693 in `"Apple.OFF"`.

Algorithm 4 *getkRinghood*
: Computes k-Ring for every Vertex

Require: Vertex Indexes in the Mesh
Require: Number of Rings k
for each Vertex i in the Mesh **do**
  if $k == 1$ **then**
    Return $vertex[i].adjacentpts$
  else if $k > 1$ **then**
    for each Ring m **do**
      if m==1 **then**
        $set < int > FinalRing = vertex[i].adjacentpts$;
      else if m > 1 **then**
        Find Ring-1 (Adjacent Points) of (m-1)-Ring Members
        Store in $FinalRing$
      end if
    end for
    Return $FinalRing$
  end if
end for

2.4 Harris Response Computation

In this stage, we use the heavily adapted function (*detectInterestPoints*) from the source code to create two new functions: *computeHarrisResponse* and $findCandidates$. The first function uses Principal Component Analysis to calculate the response for each point in the mesh and store it in the "response" attribute of the vertex structure. The second will be elaborated upon in the next section.

Algorithm 5 *computeHarrisResponse*
: Calculates the Harris response for Each Vertex in the Mesh

Require: Total Number of Vertices of the Mesh *noverts*
Require: User-Defined #Rings k and Harris Operator H
for each Vertex i in the Mesh **do**
  Calculate the $k - Ringhood$
  Store the Coordinates of Each Point in the Ring in a Structure
  Calculate the $Centroid$
  Translate the Centroid at $[0, 0, 0]$
  Apply PCA using the $CovarianceMatrix$
  Shift all the Points to XY Plane
  Solve Linear Systems using `GSL`
  Calculate Harris Response
  Store Response as an Attribute of Vertex i
end for

2.5 Interest Point Selection

Once the Harris response for every vertex has been calculated and stored in their corresponding structures, we can then find our candidate interest points, and in turn, calculate the final interest points using our function $findCandidates$.

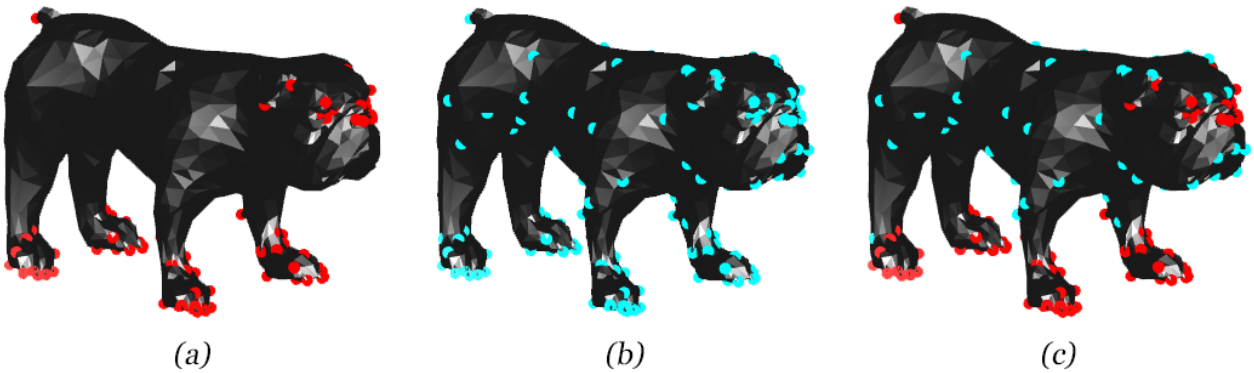
Candidate Points

To calculate the candidates points, we simply compare the Harris response of each vertex with the Harris response of its first ring neighbourhood (assuming $ring - maxima$=1). If the response is higher, then it is a candidate point.

Algorithm 6 Find Candidate Points

Require: Harris Response of each Vertex
Require: Total Number of Vertices in the Mesh *noverts*
Require: Updated *AdjacentPoints*
Require: Directory to store Candidate Points
Require: Directory to store Interest Points

```
Update AdjacentPoints by removing the Vertex itself from its Ring-1 Neighbourhood
for each Vertex in the Mesh do
    Store the Harris Response of Vertex i in a variable r1
    Store the Harris Response of AdjacentPoints in an Array compare
    Calculate k = max of compare
    if R1 i > k then
        Vertex i is a Candidate Point candidatev.insert(i)
    end if
end for
```



(a) (b) (c)

Figure 2: Interest Points [red] (a), Candidate Points [blue] (b) and both set of points (c) overlaid on `"m90.OFF"` for k=2, H=0.04 and $fract$=0.05.

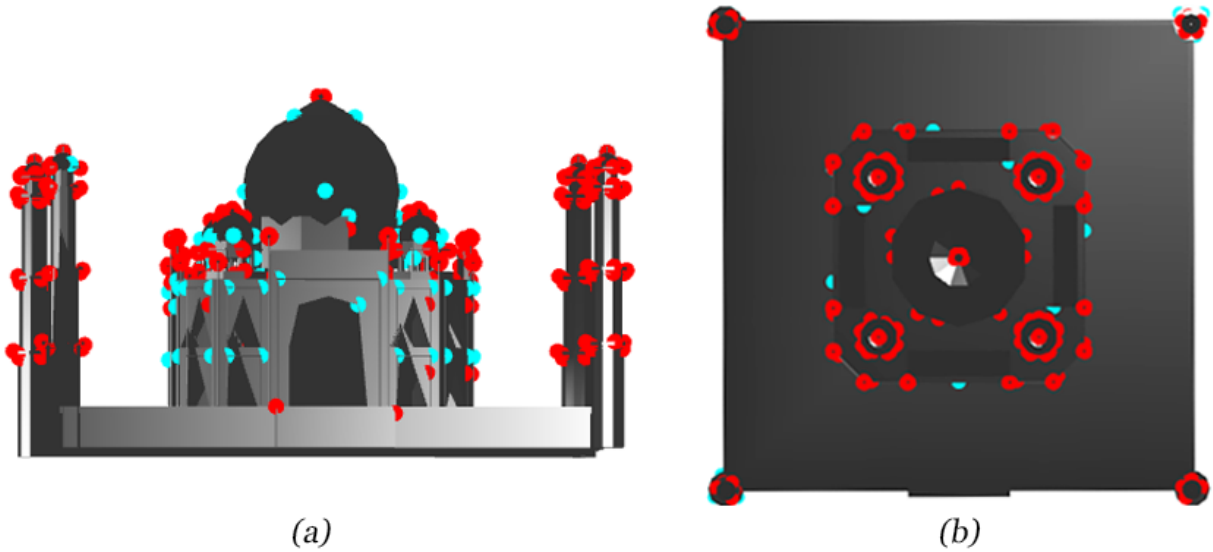
Interest Points

Next, we need to sort the Harris response for all candidate points in decreasing order and take the top fraction (as defined by the user) as our interest points. Since this task is executed only one time near the end of the program, we use a simple approach that is easy to track, rather than one prioritizing computational efficiency.

Algorithm 7 Determine Interest Points

Require: Vertex Indexes of Candidate Points
Require: Total Number of Vertices in the Mesh $noverts$
Require: Interest Points Fraction $fract$

Total #Interest Points $tot = round(fract * noverts)$;
Initialize Array for Responses $float * resp = new\ float[candidates.size()]$;
Initialize Array for Vertex Indexes $ind * vert = new\ int[candidates.size()]$;
Run Iterator through $set < int > candidates$ and populate $resp$ and $vert$ in Same Order
Bubble Sort $resp$ in Decreasing Order
Swap $vert$ every time $resp$ is swapped using the Same Index
Initialize Container for Interest Points $vector < int > finalpoints$
for $(inti = 0; i < tot; i++)$ **do**
    $finalpoints.push_back(*vert + i)$;
end for



(a) (b)

Figure 3: Interest Points [red] and Candidate Points [blue] marked over `"m399.OFF"`, for k=1, H=0.04 and $fract$=0.05, as seen from a frontal (a) and aerial (b) view, verifying the expected results of corner detection.

2.6 Graphical User Interface

One of the main goals of the project was to adapt the code of Ivan Sipiran and Benjamin Bustos, via a Graphical User Interface (GUI). The GUI should permit the user to:

- Select the Mesh Object for calculating the Interest Points.
- Choose the Degree of Ring Neighborhood for calculating the Harris Response.
- Display and interact with the Rendered Model.

The designed GUI does exactly this. It was created using the Design Mode on Qt and it is composed of two forms: *StartingForm* and *SettingsForm*. The *StartingForm* plays only one role: initialize the program and bring you to the Settings Form. Upon pressing the **START!!** button, we trigger the event that brings us to the *SettingsForm*, which will allows us to play with our meshes.

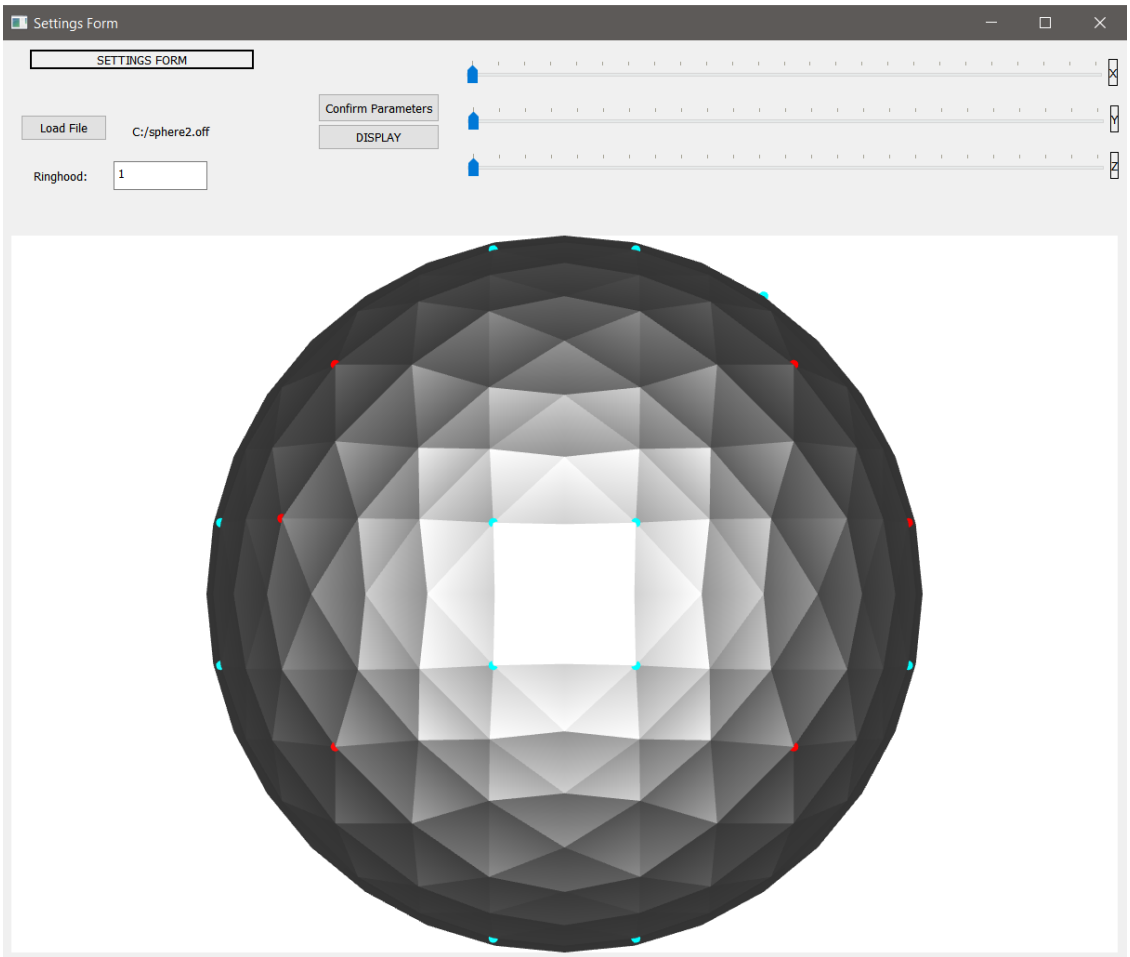


Figure 4: Graphical User Interface: Settings Form

As seen in Figure 4, we have different buttons and different sliders for each function. The first button is `Load File`, which allows the user to find the $.OFF$ file and load it into the program. This reads the $QString$ containing the directory of the mesh file, converts it into a $std :: string$ and saves it as a private attribute of the $SettingsForm$ class called $mesh\_file$.

Once loaded, we can be sure that the full path of the directory is copied, because it is made visible on the right of the `Load File` button. Subsequently, we have a $TextEdit$, in which the user can input the number of ringhood desired. By pressing the `Confirm Parameters` button, the program loads the integer that represents the ringhood. The program reads the $QString$ containing what is written in the $TextEdit$, and after checking if it is a number, saves it into a private attribute of the $SettingsForm$ class called $noring$.

Finally, we have the `Display` button which begins the computation for the Harris response, detects the candidate and interest points, saves them in two `.txt` files and renders the mesh. The event to show the rendering widget is triggered and the result is shown. Three sliders allow the user to navigate the model across three dimensions.

3 Conclusion

After a semester of working, we achieved the expected results. We were able to:

- Display a 3D Mesh with our own data structure.
- Calculate the first and $k - ringhood$ for any given vertex.
- Compute the Harris response of each vertex, and thereafter chose if a vertex is an Interest Point or not.
- Design a GUI capable of choosing different meshes and changing the parameter of the ringhood.

3.1 Accuracy & Limitations

The results we achieved are good. The candidate points we calculated and the interest points we took from the candidates, match the expected result. They indeed pick the corners of the mesh, which are said to be the interest point as per the definition of the Harris Corner Detector [5].

Despite these achievements, our program has many drawbacks.

First of all the GUI gives too few options. In the next version, we can develop a better GUI without zooming/lighting/scaling problems and with more options such as selecting different values of the Harris operator and the fraction of interest point chosen.

The algorithm itself fails sometimes for unknown reason. It may be memory allocation, a capacity issue with the GSL library integration, non-heterogeneous mesh faces, etc. Also, the way we compute the interest point is **SLOW**. It is fast for a mesh with few points (around

12000) and a low level of ringhood such as 1, 2 or 3. But for larger structures, it is slow and in the worst case scenario, even fail entirely.

When we increase the dimension of the mesh or the degree of neighborhood, the programs slow down exponentially and sometimes crashes. The main reason why the program is slow is likely due to the operation of reading the `.OFF` file twice, and reading the interest/candidate points from two others `.txt` files. The operation of calculating the k-ringhood itself is slow when the k is a big number.

As a future improvement, we recommend trying to mix the two data structures we developed, individually, into one. Doing this will make the program faster and will avoid the necessity of reading the `.OFF` file twice, as well as two separate `.txt` files for the interest points. Furthermore, in our program, data was passed across the different classes and functions for simplicity and understanding. However, this is vastly inefficient. Rather it can be passed by reference or pointers for better memory allocation and computational speed.

3.2 Discussion

In conclusion, through the course of this project, we learned how to manipulate a mesh, how to understand, adjust and implement source code developed by others, how to create a simple GUI to "play" with different meshes and 3D rendering. As a result, we now share a clearer understanding of how the Harris Operator works and its importance across the vast range of standard computer vision applications.

4 Acknowledgements

The team would like to thank all the people who shared their time and effort in assisting us with this project. In particular, we would like to thank Ahmed Gouda, Zohaib Salahuddin and Jhon Mauro for their suggestions and counsel. Finally, we would also like to thank Prem Prasad for the joint collaboration we had in working together, while still solving our projects in different ways.

5 References

[1] Sipiran, I., Bustos, B. (2011) *"Harris 3D: A robust extension of the Harris operator for interest point detection on 3D meshes"*, The Vision Computer, 27:963, © Springer-Verlag 2011, DOI: 10.1007/s00371-011-0610-y

[2] Princeton Shape Benchmark (2016) *Princeton Shape Retrieval and Analysis Group* [ONLINE] Available: `http://shape.cs.princeton.edu/benchmark/index.cgi?fbclid=IwAR3Tr55zzuaBFJhgzOUOthLNT6ZSX-L51Fx57akqDAJDwdEFdjKP4AmNGM4` [Accessed 04 January 2019]

[3] Ryan Holmes (2016) *OFF Files* [ONLINE] Available at: `http://www.holmes3d.net/graphics/offfiles/` [Accessed 10 December 2018].

[4] *GnuWin32* (2011) [ONLINE] Available: `http://gnuwin32.sourceforge.net/` [Accessed 10 December 2018]

[5] Harris, C., Stephens, M.(1988). A combined corner and edge detection. In Proceedings of The Fourth Alvey Vision Conference, pp. 147–151.