Engineering of Computer Vision Algorithms Using Evolutionary Algorithms

Marc Ebner

Eberhard Karls Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Abt. Rechnerarchitektur, Sand 1, 72076 Tübingen
marc.ebner@wsii.uni-tuebingen.de
http://www.ra.cs.uni-tuebingen.de/mitarb/ebner/welcome.html

Abstract. Computer vision algorithms are currently developed by looking up the available operators from the literature and then arranging those operators such that the desired task is performed. This is often a tedious process which also involves testing the algorithm with different lighting conditions or at different sites. We have developed a system for the automatic generation of computer vision algorithms at interactive frame rates using GPU accelerated image processing. The user simply tells the system which object should be detected in an image sequence. Simulated evolution, in particular Genetic Programming, is used to automatically generate and test alternative computer vision algorithms. Only the best algorithms survive and eventually provide a solution to the user's image processing task.

1 Introduction

Software development of computer vision algorithms is usually quite difficult. In several other fields, e.g. developing graphical user interfaces, the customer is able to specify how the product should look and how it should react to the user's input. In most cases, developing the software is rather straight forward were it not for communication problems between the customer and the software development company. The difficulty lies in understanding the customer and finding out what the customer actually wants. However, in the field of Computer Vision, for many difficult problems it is not known how the problem may be solved at all.

Suppose that the task is to program a software which recognizes different objects. The customer would be able to provide images which show the object which should be recognized. The task of the software engineer would be to write a piece of software which recognizes these objects in an image sequence taken by a video camera. The software engineer would then develop the required software in his lab and take the software and equipment to the site where it should be used. Quite often, the software developed in the lab may behave different when installed outside the lab.

This may be due to different lighting conditions. A computer vision algorithm usually depends on the given environment. Different algorithms may be needed when there is little light available compared to when there is a lot of light available and consequently there is little noise in the data. Development of computer vision software is usually a tedious process with many iterations of testing and modification.

In the field of evolutionary computation [1], where simulated evolution is used to find optimal parameters for a given problem, methods have been developed to automatically evolve computer programs. This field is called Genetic Programming (GP) [2, 3]. Currently, it is not possible to evolve large scale software such as a word processor through Genetic Programming. However, Genetic Programming has been used very successfully to evolve variable sized structures for problems such as analog and digital circuit design [4], antenna design [5], robotics or design of optical lenses [6].

With this contribution, we will show how the software development of computer vision algorithms may be automated through the use of Genetic Programming. Until recently, it was very difficult to evolve computer vision algorithms due to the enormous computational resources which are required. With the advent of powerful programmable graphics hardware it is now possible to accelerate this process such that computer vision algorithms can be evolved interactively by the user. This considerable reduces development times of computer vision algorithms.

The paper is structured as follows. First we will have a look at the field of evolutionary computer vision in Section 2. Section 3 shows how Genetic programming may be used evolve computer vision algorithms. GPU accelerated image processing is described in Section 4. A case study of an experimental system which is used to evolve computer vision algorithms is described in Section 5. Section 6 gives some conclusions.

2 Evolutionary Computer Vision

Evolutionary algorithms can be used to search for a solution which is not immediately apparent to those skilled in the art or to improve upon an existing solution. They work with a population of possible solutions. Each individual of the population represents a possible solution to the given problem. The solution is coded into the genetic material of the individuals. Starting from the parent population a new population of individuals is created using Darwin's principle "survival of the fittest". According to this principle, only those individuals are selected which are better than their peers at solving the given problem (usually above average individuals are selected). The selection is usually performed probabilistically. Above average individuals have a higher probability of getting selected than individuals which only perform below the population average. The selected individuals will breed offspring. Those offspring are usually not identical to their parents. Genetic operators such as crossover and mutation are used to recombine and change the genetic material of the parents. This process continues until a sufficient number of offspring have been created. After this, the cycle repeats for another generation of individuals.

In the field of evolutionary computer vision, evolutionary algorithms are used to search for optimal solutions for a variety of computer vision problems. Early work on evolutionary computer vision was started by Lohmann in the 1990s. He showed how an method can be evolved which computes the Euler number of an image using an Evolution Strategy [7].

Initially, evolutionary algorithms were used to evolve low-level operators such as edge detectors [8], feature detectors [9, 10], or interest point detectors [11]. They were also used to extract geometric primitives [12] or to recognize targets [13]. Evolutionary algorithms may of course also be used to evolve optimal operators for the task at hand [14]. Poli noted in 1996 that Genetic Programming would be particularly useful for image analysis [15]. Johnson et al. have used Genetic Programming successfully to evolve visual routines which detect the position of the hand in a silhouette of a person [16].

Evolutionary computer vision has become a very active research area in the last couple of years. Current work still focuses on the evolution of low-level detectors [17]. However, research questions such as object recognition [18] or camera calibration [19] are also addressed. Due to the enormous computational requirement, most experiments in evolutionary computer vision are performed off-line. Experiments in evolutionary computer vision can take from one day to several days or even weeks depending on the difficulty of the problem. Once an appropriate solution has been found, it may of course be used in real time. Recently, it has become increasingly apparent that the graphical processing unit (GPU) can be used to speed up general processing done by the central processing unit (CPU). Computer vision algorithms are particularly amenable to GPU acceleration due to the similarity between the computations required for image processing and those performed by the GPU when rendering an image.

3 Genetic Programming for Automated Software Evolution

Evolutionary Algorithms are quite frequently used for parameter optimization. However, for automatic software induction we need to apply Genetic Programming, an evolutionary method which is used to evolve variable sized structures in general and computer programs in particular [2, 3]. Genetic Programming can either be used to evolve a computer algorithm from scratch or to improve upon an existing solution. The genetic operators are used to arrange the program instructions contained in the individual such that over the course of evolution the individual performs its intended function.

A computer vision algorithm usually consists of a sequence of image processing operators which are known from the literature. These operators are applied to an input image or to a sequence of images. If one wants to solve a new computer vision problem one has to decide in what order and with which parameters the operators should be applied. For many difficult problems, this is an experimental

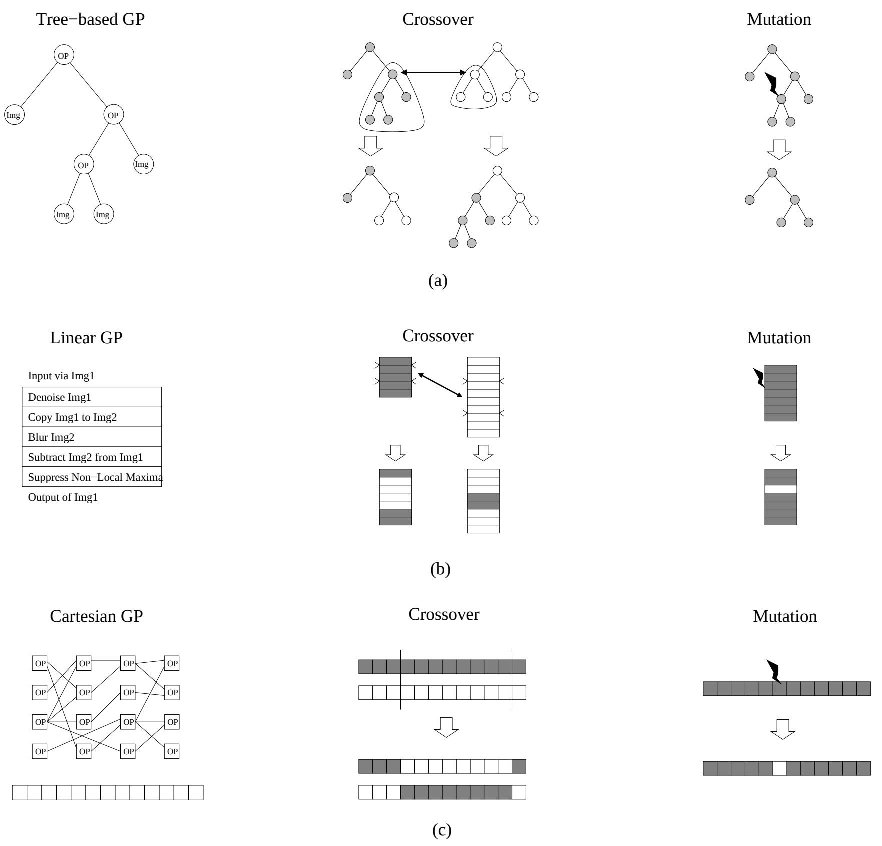

Fig. 1. (a) Tree-based Genetic Programming (b) Linear Genetic Programming (c) Cartesian Genetic Programming

process. Genetic Programming can be used to automate this process. Here, Genetic Programming is used to automatically search the space of possible image processing operators. The instructions of the computer program correspond to image processing operators.

Current research on Genetic Programming focuses on three main paradigms: tree-based GP, linear GP and Cartesian GP. The three paradigms are illustrated in Figure 1. Tree-based Genetic Programming works with a representation where the individuals are represented by trees. The internal nodes of the tree are the instructions of the program [20]. The external nodes hold the input to the program. If this representation is used for evolutionary computer vision, the external nodes represent the input image and the nodes operate on the entire image. The genetic operators crossover and mutation are used to manipulate the structure of the individual. When the crossover operator is applied, two individuals are selected from the population and two randomly selected sub-trees are exchanged between the two parent individuals. When the mutation operator is applied to create an offspring, a node is selected at random and is replaced with a newly generated sub-tree.

Linear GP uses a linear sequence of instructions [21]. The instructions operate on a set of registers. The input to the program is supplied via the registers. The instructions are used to modify the content of the registers. The output of the program is read out from the output registers. Again, genetic operators such as crossover and mutation are used to manipulate the linear sequence of instructions. If this representation is used for evolutionary computer vision, then the registers represent the entire image or a pixel from the image.

Cartesian Genetic Programming works with a representation where the operators or functions are arranged into a $n \times m$ matrix [22]. The input is provided on the left hand side of the matrix. The operator located in a given column can process the data which is available from any previous column. The output is read out from one or more output nodes on the right hand side of the matrix.

4 GPU Accelerated Image Processing

Evolution of computer vision algorithms at interactive rates is only possible with hardware acceleration. Current PCs are equipped with powerful graphics hardware which can be used to accelerate the image processing operations. The graphics hardware is even used to perform computations which are completely unrelated to computer graphics. It has successfully been used to implement algorithms such as sorting, searching, solving differential equations, matrix multiplications or computing the fast Fourier transform. Owens et al. [23] give a detailed survey on general-purpose computation on graphics hardware. Different applications such as the simulation of reaction-diffusion equations, fluid dynamics, image segmentation, ray tracing or computing echoes of sound sources have all been implemented on the GPU.

Several different packages are available which facilitate the development of GPU accelerated algorithms. Buck et al. [24] have developed a system for general-purpose computation on programmable graphics hardware by using the GPU as a streaming coprocessor. The Compute Unified Device Architecture (CUDA) package is available from Nvidia [25]. With CUDA it is possible to use the GPU as a massively parallel computing device. It is programmed using a C like language.

Image processing operators can be readily mapped to the programming paradigm which is used when rendering images. Hence, several researchers have used the GPU to implement computer vision algorithms. Fung et al. [26] implemented a projective image registration algorithm on the GPU. A hierarchical correlation based stereo algorithm was implemented by Yang and Pollefeys [27, 28]. Fung and Mann [29] showed how simple image operations such as blurring, down-sampling and computing derivatives and even a real-time projective camera motion tracking routine can be mapped to the GPU programming paradigm. Fung et al. [30] have developed a computer vision software OpenVIDIA which can be used to develop computer vision algorithms on the GPU.

We now have a look at how the GPU can be used to evolve computer vision algorithms at interactive rates. Current graphics hardware is highly optimized for rendering triangles [31]. In computer graphics, a triangle is defined by its three vertices in three-dimensional space. Additional information such as normal vector, material properties or texture coordinates are usually stored with each vertex. The graphics hardware then maps this triangle onto a two-dimensional plane with discrete pixels. This usually occurs in several stages. The three-dimensional coordinates are first transformed into eye coordinates, i.e. relative to the camera. The coordinates lying within the view frustum are then further transformed into a unit cube. This has the advantage that clipping becomes easier. The rasterizer maps all coordinates within the unit cube to the discrete raster positions of the output image.

The information which is required to color the pixel within a triangle are obtained by interpolating data from the vertices. Originally, the steps taken by the GPU to render a triangle was fixed. It could not be modified by the user. This changed in 1999, when programmable stages were introduced into the graphics pipeline [23]. With modern graphics hardware it is now possible to execute small programs, called vertex and pixel shaders, which perform various computations either per vertex or per pixel in order to determine the color or shader of a vertex or pixel. The code of the vertex shader is usually used to transform the coordinates of the vertex into the unit cube, the clip space. At this stage, all data which is required by the pixel shader is set up. This data includes the normal vector or texture coordinates both of which are specified per vertex. The rasterizer interpolates this data. It is then available for each pixel of the triangle. The code of the pixel shader is basically used to compute the color of each pixel using the interpolated data for every pixel of the triangle.

The shader programs (vertex and pixel shaders) are set up to process four-dimensional 32-bit floating-point vectors. These vectors are four-dimensional because four-dimensional homogeneous coordinates are used to process three-dimensional Cartesian coordinates. These vectors are also used to store the color components red, green and blue together with a transparency value.

When pixel and vertex shaders were first introduced, they had to be programmed using a special kind of assembly language. The commands from this assembly language were mostly dedicated to performing various computations which are frequently needed during the lighting computation when coloring a pixel. A drawback of this approach was that the code was difficult to port to a different graphics card. Later, high-level C-like languages appeared, e.g. Cg [32], developed by Nvidia, and the Open Graphics Library Shading Language (OpenGLSL) [33]. With these C-like languages it is now considerably easier to write pixel and vertex shaders which can be executed on a variety of different graphics cards. The code written for the vertex and pixel shaders is compiled by the graphics driver wherever it is executed.

We will be using the OpenGL Shading Language to program the vertex and pixel shaders. Image processing operations are mapped to the GPU by sending four vertices to the GPU. These vertices constitute a quad which represents our

Texture Declaration

```
uniform sampler2D textureColor;
float sx=1.0/width; float sy=1.0/height;
```

Blur Shader

```
gl_FragColor.rgb=texture2D(textureColor,gl_TexCoord[0].st,3);
```

Gradient Shader

```
vec2 dx[4]={vec2(-sx,.0),vec2(.0,sy),vec2(sx,.0),vec2(.0,-sy)};
vec4 color,delta;
float gradient=0.0;
color=texture2D(textureColor,gl_TexCoord[0].st,0);
for (int i=0;i<4;i++) {
  delta=color-texture2D(textureColor,gl_TexCoord[0].st+dx[i],0);
  gradient+=dot(delta,delta);
}
gl_FragColor.rgb=2*sqrt(gradient);
```

Laplacian Shader

```
vec4 color;
vec2 dx[4]={vec2(-sx,.0),vec2(.0,sy),vec2(sx,.0),vec2(.0,-sy)};
color=4*texture2D(textureColor,gl_TexCoord[0].st,0);
for (int i=0;i<4;i++)
  color-=texture2D(textureColor,gl_TexCoord[0].st+dx[i],0);
gl_FragColor=5*color;
```

Fig. 2. Shader code which was used to create the output images shown in Figure 3. This demonstrates how easily computer vision operators can be implemented using the OpenGL Shading Language.

image. The pixel shader is used to perform the image processing operation in parallel for all pixels of the input image. The input image is stored in a texture which is made available to the pixel shader through texture operations. Figure 2 shows the OpenGLSL code for three different image operators (Blur, Gradient, Laplacian). The output of these three pixel shaders for a sample image is shown in Figure 3.

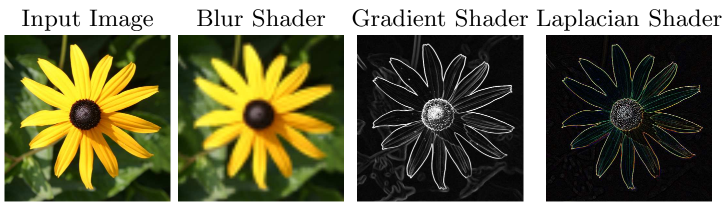

Fig. 3. Output images generated using the four pixel shaders shown in Figure 2.

The blur shader uses the mip mapping mechanism of OpenGLSL to compute a blurred input image. The mip mapping mechanism is usually used to down-sample textures. The blur shader can be implemented with a single line of code. The gradient shader reads adjacent pixels of the texture, computes the differences and sums up the squared differences over all three color channels. The Laplacian shader reads out the center and surrounding pixels. It then computes the differences between these two. The output of the Laplacian falls within the range [-1,1]. It is therefore mapped to [0,1] for display. These three examples show how easy it is to use the GPU for computer vision applications.

5 Evolving Computer Vision Algorithms Interactively

We have created an object recognition vision system which allows the user to automatically evolve computer vision algorithms at interactive rates. The system is equipped with a video camera from which input images are gathered. Alternatively, the system can also process images from video sequences. The user specifies the object which should be recognized using the mouse pointer. The user keeps pressing the mouse button as long as the object is located underneath the mouse pointer. This is the teaching input used by the system.

Our system works with a parent population of μ individuals. Initially, these individuals are generated entirely at random. The output of the three best individuals is always shown on the screen as shown in Figure 4. The task of the evolved computer programs is to locate the object, which was specified by the user, as closely as possible. The pixel with the largest response, where the RGB colors are interpreted as a 24-bit number, is taken as the position of the object. If several pixels have a response of 0xFFFFFF, then the center of gravity is computed.

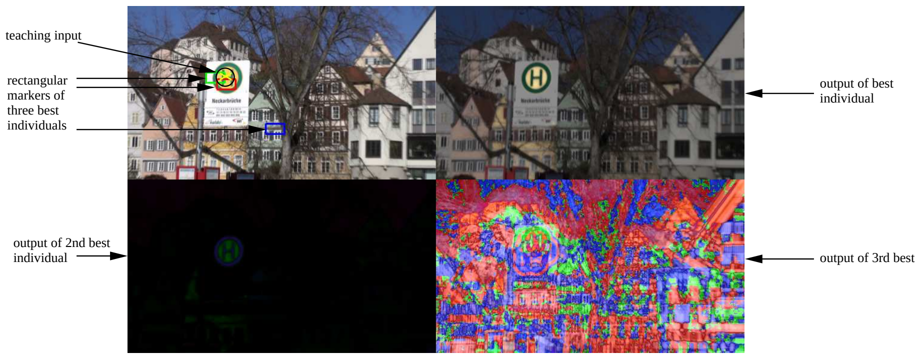
teaching input
rectangular markers of three best individuals
output of best individual
output of 2nd best individual
output of 3rd best

Fig. 4. System overview. The input image is shown in the upper left hand corner. The remaining three images show the output of the best three individuals of the population. The teaching input (specified by the user using the mouse) is the yellow marker. The three rectangular markers (shown in red, green, and blue) show the position located by the best three individual.

The Cartesian Genetic Programming paradigm [22] is used as a representation for the individuals. Each individual consists of a set of connected nodes. For each node, we have to specify the function which is computed by this node. We also have to specify how a given node is connected to previous nodes. This set of connected nodes is not evolved directly. Instead, one works with a linear representation which can be readily mapped to the set of connected nodes. The first number specifies the function computed by the first node. The second number specifies the first argument which can be used be this node and the third number specifies the second argument. The fourth number specifies the function of the second node and so on. The functions which are available to the individuals are taken directly from the OpenGL shading language. The representation is fully described by Ebner [34].

Starting from the parent individuals, we create offspring by using the crossover operator with a probability of 70%. In other words, 70% of the offspring are created by recombining the genetic material of parent individuals. The remaining 30% of the offspring are created by simply reproducing a parent individual. For a mutation, a randomly chosen byte is either decreased by one or increased by one or the entire string is mutated with a mutation probability of $2/l$ where l is the length of the string in bits, i.e. on average, we will have two mutations per offspring. In addition to those offspring which are generated from the parent population, the same number of individuals are also generated at random. This allows for a continuous influx of new genetic material.

Individuals are evaluated by computing the distance between the position, which is specified by the user, and the position which is returned by the individual. This is our error measure or fitness function. Since our input is dynamically changing, we also re-evaluate the parent individuals. We then sort the μ parents and λ offspring according to fitness.

The best μ individuals are selected as parents for the next generation among both parents and offspring. This is a so called $(\mu + \lambda)$ Evolution Strategy. Note that we are working with a redundant representation. Two individuals which differ in their genetic representation can actually compute the same function. That's why only one individual for every fitness value is considered for selection. This is an effective method of diversity maintenance in our context. We then repeat this process of reproduction, variation and selection for every input image.

Our system, consisting of an Intel Core 2 CPU running at 2.13GHz and a GeForce 9600GT/PCI/SEE2 graphics card, achieves a frame rate of 4 Hz while evaluating 23 individuals, each processing an input image of size 320x240, for each generation. The speedup is 17 if two high level operators followed by a 2×2 matrix of simpler operators are used compared to an all software implementation of the same algorithm. The speedup increases to 24.6 if four high level operators are used.

This system was tested on several different image sequences (shown in Figure 5). Object detectors which have been evolved so far, include ducks in a pond, traffic signs as well as an interest point on a building. An object detector which is able to locate the object with a reasonably small error was evolved in less than 120 generations, i.e. within 30 seconds with a frame rate of 4Hz. Manually writing these detectors would have taken considerably longer. As the graphics

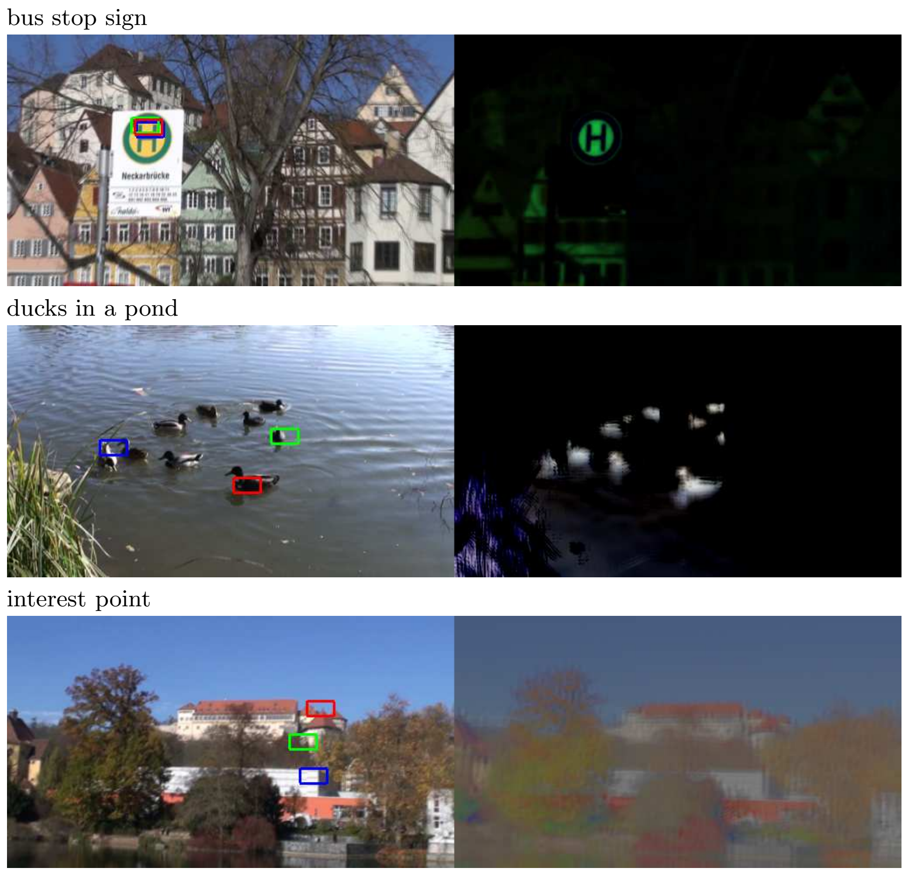

Fig. 5. Results obtained with three evolved object detectors. The three rectangular markers (in red, green and blue) show the position of the object located by the three best individuals of the population. The best output (shown in red) corresponds closely to the object or part of the image which should be located.

hardware gets more powerful, we will be able to evolve increasingly complex detectors.

6 Conclusions

With current advances in computer graphics hardware it is now possible to automatically generate computer vision algorithms at interactive rates. This reduces development times considerable and also allows laymen, not having any previous experience with the programming of computer vision algorithms, to automatically generate such algorithms. This is a step towards the programming of computers by telling them what the user wants and without explicitly telling the computer exactly what to do.

References

1. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin (2007)
2. Koza, J.R.: Genetic Programming. On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, MA (1992)
3. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming - An Introduction: On The Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann Publishers, San Francisco, CA (1998)
4. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III. Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers, San Francisco, CA (1999)
5. Linden, D.S.: Innovative antenna design using genetic algorithms. In Bentley, P.J., Corne, D.W., eds.: Creative Evolutionary Systems, San Francisco, CA, Morgan Kaufmann (2002) 487–510
6. Koza, J.R., Al-Sakran, S.H., Jones, L.W.: Automated re-invention of six patented optical lens systems using genetic programming. In: Proc. of the 2005 Conf. on Genetic and Evolutionary Computation, Washington, DC, New York, NY, ACM (2005) 1953–1960
7. Lohmann, R.: Bionische Verfahren zur Entwicklung visueller Systeme. PhD thesis, Technische Universität Berlin, Verfahrenstechnik und Energietechnik (1991)
8. Harris, C., Buxton, B.: Evolving edge detectors with genetic programming. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming, Proc. of the 1st Annual Conf., Stanford University, Cambridge, MA, The MIT Press (1996) 309–314
9. Rizki, M.M., Tamburino, L.A., Zmuda, M.A.: Evolving multi-resolution feature-detectors. In Fogel, D.B., Atmar, W., eds.: Proc. of the 2nd American Conf. on Evolutionary Programming, Evolutionary Programming Society (1993) 108–118
10. Andre, D.: Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In Kinnear, Jr., K.E., ed.: Advances in Genetic Programming, Cambridge, MA, The MIT Press (1994) 477–494
11. Ebner, M.: On the evolution of interest operators using genetic programming. In Poli, R., Langdon, W.B., Schoenauer, M., Fogarty, T., Banzhaf, W., eds.: Late Breaking Papers at EuroGP'98: the 1st European Workshop on Genetic Programming, Paris, France, The University of Birmingham, UK (1998) 6–10
12. Roth, G., Levine, M.D.: Geometric primitive extraction using a genetic algorithm. IEEE Trans. on Pattern Analysis and Machine Intelligence **16**(9) (1994) 901–905
13. Katz, A.J., Thrift, P.R.: Generating image filters for target recognition by genetic learning. IEEE Trans. on Pattern Analysis and Machine Int. **16**(9) (1994) 906–910
14. Ebner, M., Zell, A.: Evolving a task specific image operator. In Poli, R., Voigt, H.M., Cagnoni, S., Corne, D., Smith, G.D., Fogarty, T.C., eds.: Joint Proc. of the 1st Europ. Workshops on Evolutionary Image Analysis, Signal Processing and Telecommunications, Göteborg, Sweden, Berlin, Springer-Verlag (1999) 74–89
15. Poli, R.: Genetic programming for image analysis. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996, Proc. of the 1st Annual Conf., Stanford University, Cambridge, MA, The MIT Press (1996) 363–368
16. Johnson, M.P., Maes, P., Darrell, T.: Evolving visual routines. In Brooks, R.A., Maes, P., eds.: Artificial Life IV, Proc. of the 4th Int. Workshop on the Synthesis and Simulation of Living Systems, Cambridge, MA, The MIT Press (1994) 198–209

17. Trujillo, L., Olague, G.: Synthesis of interest point detectors through genetic programming. In: Proc. of the Genetic and Evolutionary Computation Conf., Seattle, WA, ACM (2006) 887–894
18. Treptow, A., Zell, A.: Combining adaboost learning and evolutionary search to select features for real-time object detection. In: Proc. of the IEEE Congress on Evolutionary Computation, Portland, OR. Volume 2., IEEE (2004) 2107–2113
19. Heinemann, P., Streichert, F., Sehnke, F., Zell, A.: Automatic calibration of camera to world mapping in robocup using evolutionary algorithms. In: Proc. of the IEEE Int. Congress on Evolutionary Computation, San Francisco, CA, IEEE (2006) 1316–1323
20. Koza, J.R.: Artificial life: Spontaneous emergence of self-replicating and evolutionary self-improving computer programs. In Langton, C.G., ed.: Artificial Life III: SFI Studies in the Sciences of Complexity Proc. Vol. XVII, Reading, MA, Addison-Wesley (1994) 225–262
21. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K.E., ed.: Advances in Genetic Programming, Cambridge, MA, The MIT Press (1994) 311–331
22. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Banzhaf, W., et al., eds.: Proc. of the Genetic and Evolutionary Computation Conf., San Francisco, CA, Morgan Kaufmann (1999) 1135–1142
23. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, State of the Art Reports. (2005) 21–51
24. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware. In: Int. Conf. on Comp. Graphics and Interactive Techniques (ACM SIGGRAPH). (2004) 777–786
25. NVIDIA: NVIDIA CUDA. Compute Unified Device Architecture. V1.1. (2007)
26. Fung, J., Tang, F., Mann, S.: Mediated reality using computer graphics hardware for computer vision. In: Proc. of the 6th Int. Symposium on Wearable Computers, ACM (2002) 83–89
27. Yang, R., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. In: Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition, IEEE (2003) 211–218
28. Yang, R., Pollefeys, M.: A versatile stereo implementation on commodity graphics hardware. Real-Time Imaging **11**(1) (2005) 7–18
29. Fung, J., Mann, S.: Computer vision signal processing on graphics processing units. In: Proc. of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, 2004. Volume 5., IEEE (2004) 93–96
30. Fung, J., Mann, S., Aimone, C.: OpenVIDIA: Parallel GPU computer vision. In: Int. Multimedia Conference. Proc. of the 13th annual ACM Int. Conf. on Multimedia, Singapore. Volume 5., ACM (2005) 849–852
31. Akenine-Möller, T., Haines, E.: Real-Time Rendering. 2nd edn. A K Peters, Natick, MA (2002)
32. Fernando, R., Kilgard, M.J.: The Cg Tutorial. The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley, Boston, MA (2003)
33. Rost, R.J.: OpenGL Shading Language. 2nd edn. Addison-Wesley, Upper Saddle River, NJ (2006)
34. Ebner, M.: A real-time evolutionary object recognition system. In Vanneschi,L., Gustafson,S., Moraglio,A., Falco,I.D., Ebner,M., eds.: Genetic Programming: Proc. of the 12th Europ. Conf., Tübingen, Germany, Berlin, Springer (2009) 268–279