A Real-Time Evolutionary Object Recognition System

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. We have created a real-time evolutionary object recognition system. Genetic Programming is used to automatically search the space of possible computer vision programs guided through user interaction. The user selects the object to be extracted with the mouse pointer and follows it over multiple frames of a video sequence. Several different alternative algorithms are evaluated in the background for each input image. Real-time performance is achieved through the use of the GPU for image processing operations.

1 Motivation

Current vision systems are usually not adaptive to their environment. Algorithms which have been developed in the laboratory often break when the vision system is moved to a different environment. The only component of a vision system which is currently adaptive is the automatic white balance of the camera. Recently, Ebner [1] proposed building an adaptive on-line evolutionary visual system. Such a system would adapt itself to the current environment. Evolutionary algorithms would be used to search for an algorithm which would always perform optimally for the given environmental conditions.

Suppose we are given some algorithm which extracts or detects a person in an image. The algorithm would perform flawlessly provided that there is enough light available to illuminate the person in the image. During sunset, the algorithm may have to be modified to extract or detect the same person. At night when there is likely going to be a lot of noise in the image data, additional modifications have to be made to the algorithm to perform the same task it did during daylight.

Ebner [1] suggested to run multiple, slightly modified algorithms in the background in addition to the main algorithm. The main algorithm would take the input image, compute an output and present this output to the user. The system would also evaluate the modified algorithms using the same input image. The algorithm with the best performance would become the main algorithm for the next input image. Clearly, such a system would require enormous computational powers.

A computer vision algorithm usually applies several image processing operators to an input image. Such algorithms often struggle to maintain real-time performance. How will it then be possible to run multiple algorithms in parallel? The rise of powerful graphics processing units provides a solution to this problem.

Below, we will show how we created an experimental real-time evolutionary vision system by exploiting the power of the graphics processing unit. The paper is structured as follows. We first provide a brief review of related research in the field of evolutionary computer vision in Section 2. In Section 3, we describe how our real-time evolutionary vision system works. Details on how this system is mapped to the graphics hardware is presented in Section 4. The system is evaluated on several image sequences in Section 5. Conclusions are provided in Section 6.

2 Evolutionary Computer Vision

In evolutionary computer vision, evolutionary algorithms are often used to search for a solution which is not immediately apparent to those skilled in the art or to improve upon an existing evolution. Early work on evolutionary computer vision started in the early 1990s. Lohmann, a pioneer in the field, advocated the idea and showed how an Evolution Strategy may be used to find an algorithm which computes the Euler number of an image [2]. In relatively early work, evolutionary algorithms were mostly used to evolve low-level operators, e.g. edge detectors [3], feature detectors [4], or interest point detectors [5]. They were also used for target recognition [6].

However, early on, it was clear that in principle these techniques could be used to create fully adaptive operators which would be optimal or near optimal for the task at hand [7]. It was also clear that Genetic Programming would be particularly useful for image analysis [8]. Johnson et al. were successful at using genetic programming to evolve visual routines [9]. Today, evolutionary computer vision has become a very active research area. Current work ranges from the evolution of low-level detectors [10], to object recognition [11,12] or camera calibration [13]. A taxonomic tutorial and introduction into the field is given by Cagnoni [14].

Due to the enormous computational requirements, experiments in evolutionary computer vision are usually performed off-line (see Mussi and Cagnoni [15] for a notable exception). Once an appropriate algorithm has been evolved, it may of course be used in real time. Genetic algorithms [16] or Evolution Strategies [17] are mostly used to improve already existing algorithms, i.e. they are used for parameter optimization while Genetic Programming [18,19] is used to evolve an algorithm from scratch. Since we are interested in the most general scheme for the evolution of algorithms, we will be using Genetic Programming to search the space of possible solutions.

3 A Real-Time Evolutionary Object Recognition System

In searching for an optimal computer vision algorithm, one has to answer the question in what order and with which parameters should the known operators from the literature be applied to the input image to achieve the desired output. The instructions of the evolved computer program correspond to the operators from the computer vision literature. Possible representations which could be used to evolve image processing algorithms include tree-based GP, linear GP or Cartesian GP. For our experiments, we will be using Cartesian Genetic Programming [20] because this GP variant is readily mapped to the graphics hardware as we will describe below.

Similar to the Cartesian GP paradigm, we will be working with a $(n_x \times n_y)$ matrix of image processing operators as shown in Figure 1. In addition to this matrix, we will be using a vector of n_1 input images (column 0) which will serve as input to the operators stored in the matrix. The output computed by one operator located at position (x, y) in the matrix can be computed by combining the output computed by any of the cells located at column $x-1$. The processed image data is finally available in column n_x. We denote this representation as a $n_1 + n_x \times n_y$ representation. Many algorithms known from the literature can be fit into this scheme.

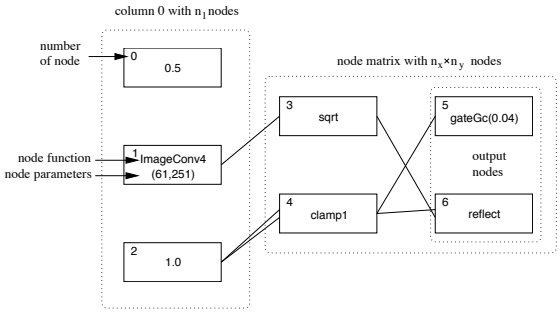

Fig. 1. Cartesian Genetic Programming representation of an individual.

Table 3 shows the list of operators which are available for column 0. The first three operators, `0.0`, `0.5` and `1.0` simply return a gray image where all colors are set to the color `vec3(0.0,0.0,0.0)`, `vec3(0.5,0.5,0.5)`, `vec3(1.0,1.0,1.0)` respectively. The operator `Image` outputs the input image. Argument 1 is used to offset the image by mapping the one byte argument to an offset vector as shown in Figure 2(a). A byte value of 0 is mapped to no offset. The vector moves clockwise and radially outward as the byte value increases. The maximum distance from the center is 10 pixels. The second byte argument is used to set the appropriate scale (see Figure 2(b) and (c)). The range $[0, 255]$ is mapped to the scale $[0, 4]$. A scale of 0 is just the original input image. A scale of 1 denotes a down-sampled

(a) (b) (c)

Fig. 2. (a) one byte is used to specify an image offset. The offset is determined by moving a vector clockwise and radially outward as the byte value increases. (b) image at scale 0 (c) image at scale 4.

version of the original image where a 2×2 area of pixels is averaged for every image pixel.

The operators `DX` and `DY` compute the derivative in the x- and y-direction respectively. The operator `Lap` computes the Laplacian and the operator `Grad` computes the gradient magnitude. For all these operators, the two byte arguments again denote offset and scale of the input image which is used for the computation. The operator `ImageLogDX` outputs the logarithm of the derivative in the x-direction. This is a so called color constant descriptor [21] which only depends on the reflectance on the patch but not on the illuminant. Such descriptors are particularly useful for object detection when the lighting conditions change.

The operator `ImageIdw`, can be used to compute a gray scale image from the input image. The output of this operator $o_i = w_r c_r + w_g c_g + w_b c_b$ with $i \in \{r, g, b\}$ is computed using RGB weights $w_i \in \{-1, 0, 1, 2\}$ The RGB weights are stored in the second byte argument. The first byte argument is used to store the offset and the scale of the image as before. Similarly, the operator `ImageChrom` can be used to compute chromaticities. For this operator, the output is computed using $o_i = w_i c_i / (c_r + c_g + c_b)$.

The operators `ImageGC1`, `ImageGC4`, `ImageGC16` provide the input image convolved with a filter matrix which is stored in the two byte arguments. A 3×3 filter matrix is used with two bits per weight $w_i \in \{-1, 0, 1, 2\}$ with $i \in \{1, ..., 8\}$. The center element is not included. Let N be the neighborhood of the current pixel, then the output is computed as

$$o_i = \sum_{(x,y) \in N} w_i c_i. \tag{1}$$

The operator `ImageGC1` places the neighboring pixels directly around the current pixel. The operator `ImageGC4` increases the distance of the neighboring pixels from the current pixel radially outward by a distance of 4. The operator `ImageGC16` increases this distance to 16 pixels. An additional convolution operator `ImageGCd` is provided which uses a variable sized distance of the pixels from the current pixel. The first byte argument is used to specify the weights $w_i \in \{-0.5, 0.5\}$ with $i \in \{1, ..., 8\}$. The second byte argument is used to specify the distance of the pixels within the range $[0, 64]$.

Fig. 3. Operators available for column 0.

Function of Operator	Name	Args	Arg 1	Arg 2
zero	`0.0`	0	unused	unused
half	`0.5`	0	unused	unused
one	`1.0`	0	unused	unused
identity	`Image`	2	offset	scale
horizontal derivative	`DX`	2	offset	scale
vertical derivative	`DY`	2	offset	scale
Laplacian	`Lap`	2	offset	scale
gradient	`Grad`	2	offset	scale
color constant desc.	`ImageLogDX`	2	offset	scale
gray scale image	`ImageIdw`	2	offset/scale	RGB weights
chromaticities	`ImageChrom`	2	offset/scale	RGB weights
convolution (dist. 1)	`ImageGC1`	2	conv. weights	conv. weights
convolution (dist. 4)	`ImageGC4`	2	conv. weights	conv. weights
convolution (dist. 16)	`ImageGC16`	2	conv. weights	conv. weights
convolution	`ImageGCd`	2	conv. weights	distance
segmentation	`ImageSeg`	2	levels	scale

The operator `ImageSeg` is used to segment or discretize the image into a discrete number of regions. The first byte argument specifies the number of allowed pixel values and the second byte argument specifies the scale of the image which is used for the computation.

A $n_x \times n_y$ matrix of operators is used to combine the image data which has been made available in column 0. The list of operators which can be used in this $n_x \times n_y$ matrix is shown in Table 4. Many of these operators are taken directly from the specification of the OpenGL Shading Language (OpenGLSL) [22]. The data is fed through the matrix from left to right. First, column 1 is evaluated, then column 2 and so on. Each node of the matrix either takes one or two arguments. The first byte argument specifies from which node in the previous column the input value `v1` is read out. A modulo operation is used to map the byte argument to the range $[0, n_x]$. Similarly, the second byte argument specifies the input node used for `v2`. Some operations only take a single argument. In this case, the second byte argument is ignored. Some operators use the second byte argument as a constant `c2`. This constant is computed using `c2=b/255`.

The functions performed by the operators are shown in the last column of Table 4 using the syntax of the OpenGL shading language. Functions include standard operations such as addition, subtraction, multiplication, division as well as specialized functions for image processing which extract one of the channels or uses a color channel as a gate function. Other functions, which were included in the function set, are functions which are mostly used for computer graphics applications. Those functions were included simply because they are readily available from the OpenGLSL function set.

The overall output of the detector can be computed once the last column has been evaluated. To compute the overall output, we average the output of all

Fig. 4. Operators available for the nodes of the $n_x \times n_y$ matrix.

function of operator	name	args	computed function
pass through	`id`	1	v1
absolute value	`abs`	1	abs(v1)
scalar product	`dot`	1	vec3(dot(v1,v1))
square root	`sqrt`	1	sqrt(v1)
normalize	`norm`	1	normalize(v1)
clamp	`clamp(0,1)`	1	clamp(v1,0.0,1.0)
step function	`step(0)`	1	step(vec3(0),v1)
step function	`step(0.5)`	1	step(vec3(0.5),v1)
smooth step function	`smstep(0,1)`	1	smoothstep(vec3(0),vec3(1),v1)
red channel	`red`	1	vec3(v1.r,0,0)
green channel	`green`	1	vec3(0,v1.g,0)
blue channel	`blue`	1	vec3(0,0,v1.b)
channel average	`avg`	1	vec3((v1.r+v1.g+v1.b)/3)
channel minimum	`min`	1	vec3(min(min(v1.r,v1.g),v1.b))
channel maximum	`max`	1	vec3(max(max(v1.r,v1.g),v1.b))
mark minimum comp.	`equalMin`	1	equal(v1,vec3(min(min(v1.r,v1.g),v1.b)))
mark maximum comp.	`equalMax`	1	equal(v1,vec3(max(max(v1.r,v1.g),v1.b)))
use red channel as gate	`gateR`	1	?(v1.r>0) vec3(v1.g):vec3(a.b)
use green channel as gate	`gateG`	1	?(v1.g>0) vec3(v1.r):vec3(a.b)
use blue channel as gate	`gateB`	1	?(v1.b>0) vec3(v1.r):vec3(a.g)
use red channel as gate	`gateRc`	2	?(v1.r>c2) vec3(v1.g):vec3(v1.b)
use green channel as gate	`gateGc`	2	?(v1.g>c2) vec3(v1.r):vec3(v1.b)
use blue channel as gate	`gateBc`	2	?(v1.b>c2) vec3(v1.r):vec3(v1.g)
step function	`step`	2	step(vec3(c2),v1)
addition	`+`	2	v1+v2
subtraction	`-`	2	v1-v2
multiplication	`*`	2	v1*v2
division	`/`	2	v1/v2
minimum	`min`	2	min(v1,v2)
maximum	`max`	2	max(v1,v2)
clamp	`clamp0`	2	clamp(v1,v2,vec3(1))
clamp	`clamp1`	2	clamp(v1,vec3(0),v2)
mix	`mix`	2	mix(v1,v2,0.5)
step	`step`	2	step(v1,v2)
less than	`lessThan`	2	vec3(lessThan(v1,v2))
greater than	`greaterThan`	2	vec3(greaterThan(v1,v2))
dot	`dot`	2	vec3(dot(v1,v2))
cross	`cross`	2	cross(v1,v2)
reflect	`reflect`	2	reflect(v1,v2)
refract	`refract`	2	refract(v1,v2,0.1)

operators in column n_x. In other words, we output the average behavior of the detectors in the last column. Because of this, modifications to a single element towards the right hand side of the matrix have a relatively small effect on the overall output.

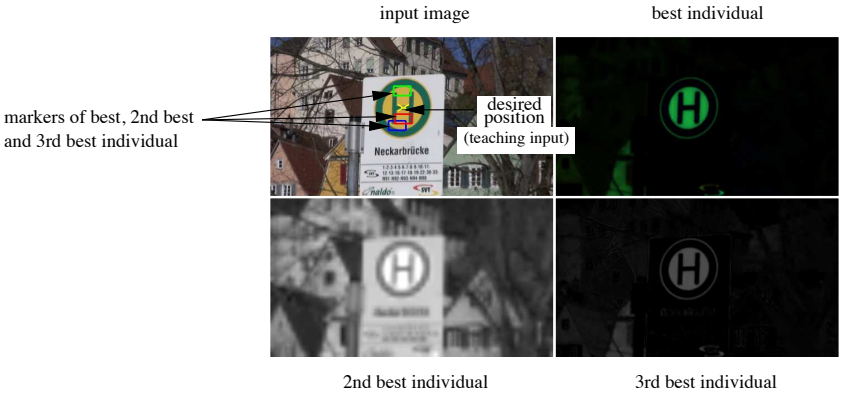

Fig. 5. System overview. The user manually specifies the position of the object which should be extracted in the upper left hand sub-window. The output of the three best individuals is shown using markers (overlayed on the input image). The images which are computed by the three best individuals are also shown.

Obviously, it would also be possible to use a different method to compute the overall output. For instance, we could also multiply the output of the detectors located in the last column. In this case, all detectors would have to have a high output for the overall detector to respond at all. However, such an object detector would most likely be highly fragile. All detector outputs have to be non-zero for the overall detector to output anything at all.

Experiments in the field of evolutionary computer vision are usually very expensive with respect to the computational requirements. Each individual of the population represents a possible algorithmic solution for a given problem. Each solution has to be evaluated on the input image or sets of input images. We will be working with an input stream of images which is continually being processed by the system. Due to clever use of the graphics processing unit, our system is able to evaluate multiple individuals for every input image.

Our task will be to locate certain objects in the input stream. The user is able to tell the system which object should be extracted by moving the mouse pointer over the object and then pressing the left mouse button (see Figure 5). The position of the mouse pointer is then used for computing the fitness of the individuals as long as the button is pressed. The detected object position as well as the processed image of the three best individuals of the population is always output onto the screen. Once the user releases the mouse button, the images are still being processed, however, evolution is halted and the output of the three best individuals is continued to be shown.

Figure 1 shows the entire setup for a sample individual. It is straightforward to map this representation to a linear bit string genome by concatenating all the parameters. The parameters of column 0 are stored first, followed by the parameters for the $n_x \times n_y$ matrix read out from top to bottom and from left to right.

4 GPU Accelerated Image Processing

We have used the graphics processing unit (GPU) to accelerate the image processing operators. Fung et al. [23] noticed very early on that image processing tasks can be mapped to the graphics hardware. Today, graphics hardware is used to accelerate a variety of tasks from the simulation of reaction-diffusion equations to fluid dynamics or image segmentation [24]. Nvidia has developed the Compute Unified Device Architecture (CUDA) [25]. This architecture allows the programmer to use the GPU as a massively parallel computing device. The CUDA architecture is highly suited for accelerating image processing operations. However, we have chosen to use the OpenGL shading language for our implementation. OpenGLSL has the advantage of providing easy access to scale spaces through mip mapping. Currently, it is not known whether a CUDA implementation would provide a significant speedup over the approach followed here.

Our choice of operators and representation which were described in the previous section were largely fixed by the syntax of the OpenGL shading language. OpenGLSL is usually used to render realistic computer graphics in real-time. It is highly optimized to render triangles and planar polygons. For added realism, textures can be applied to these triangles and polygons. So called pixel shaders can be used to individually program the operation which is performed by the GPU whenever the rasterizer renders a single pixel.

The pixel shaders can be programmed using a C-like language, the OpenGL shading language. Each pixel shader has access to multiple textures. We implemented the above representation by rendering a single rectangle which has the same size as the input image and which consists of four vertices. The rasterizer executes the code of the pixel shader for every pixel of the rectangle. The current input image is provided to the pixel shader as a texture. The operations listed in Table 3 are implemented by reading out the texture. For instance, the following OpenGLSL code reads out the input image with an offset of (3,4) and at a scale of 2.

```
vec3 c=texture2D(texture0,gl_TexCoord[0].st+vec2(3,4),2).rgb;
```

The three component vector `c` then holds the down-sampled data from the input image.

All high level operations such as edge detection, computation of the Laplacian, convolution or segmentation are implemented through the OpenGLSL. The result of these operations can then be combined through binary operations such as multiplication, computation of maximum values or through gate functions. Obviously, in computer vision one also wants to apply multiple image processing operators in sequence. For instance, one would like to apply a convolution and then an edge detector to the convolved image. However, it is currently not possible to implement such operations using the OpenGLSL with a single pass through the graphics pipeline. One would have to use multiple passes through the graphics pipeline and the image data would have to be exchanged between the GPU and the CPU for each pass. This transfer between GPU and CPU would

be a severe bottleneck. In the future, the OpenGLSL may allow write operations to textures. If this were possible, we would be able to compute multiple image processing operations with a single pass through the pipeline. Unfortunately, at present, this is not a possibility.

5 Experiments

Our system works with a parent population of μ individuals. We basically use a $(\mu + \lambda)$ Evolution Strategy to evolve our pixel shader programs. All parents are re-evaluated for each input image because the input image is continuously changing. New parents are selected among the parents and the offspring by sorting them in ascending order according to their fitness and then selecting the best μ parents. Since the representation that we use is highly redundant, the μ best parents will all be identical after several iterations. That's why we sort parents and offspring according to fitness and then select the μ best parents which all have different fitness values. This provides a simple form of diversity maintenance.

In addition, half of the offspring for every iteration are generated from scratch using the random number generator. This also provides a steady influx of alternative algorithms and prevents the system from converging to a single point inside the search space. The other half of the offspring are generated by recombining two parent individuals with a crossover probability of $p_{\text{cross}} = 0.7$.

All offspring are mutated by selecting one of two mutation operations at random. The first mutation operation is a standard GA-like mutation operation with a mutation probability of $p_{\text{mut}} = \frac{2}{l}$ where l is the size of the genome in bits. The second mutation operation selects one byte of the individual at random and increases or decreases that byte at random. This allows for small incremental or decremental changes to arguments which specify offsets or scale. Alternatively, a gray code could also have been used to achieve the same effect.

The task of the individuals is to detect objects in an image sequence which are specified interactively by a human operator by moving the mouse over the object and holding the mouse button pressed as long as the evolved object detector is not good enough. Our training set consists of those input images where the mouse button is pressed and our testing test consists of all other images. No image is used twice during an evolutionary run.

The evolved object detector outputs an entire image. The detected object position is determined by locating the pixel with the highest value. Each pixel with RGB components $[r, g, b]$ is interpreted as a 24-bit number. If several pixels have the value $FFFFFF$, we compute the center of gravity of these pixel positions. Fitness is computed based on the difference (measured in pixels) between the detected object position and the object position which is manually determined by a human operator.

We tested our system on several different image sequences. Each detector receives as input only a single image. Naturally, the shape of an object can vary from one image to the next. Its colors will also vary slightly from one image to

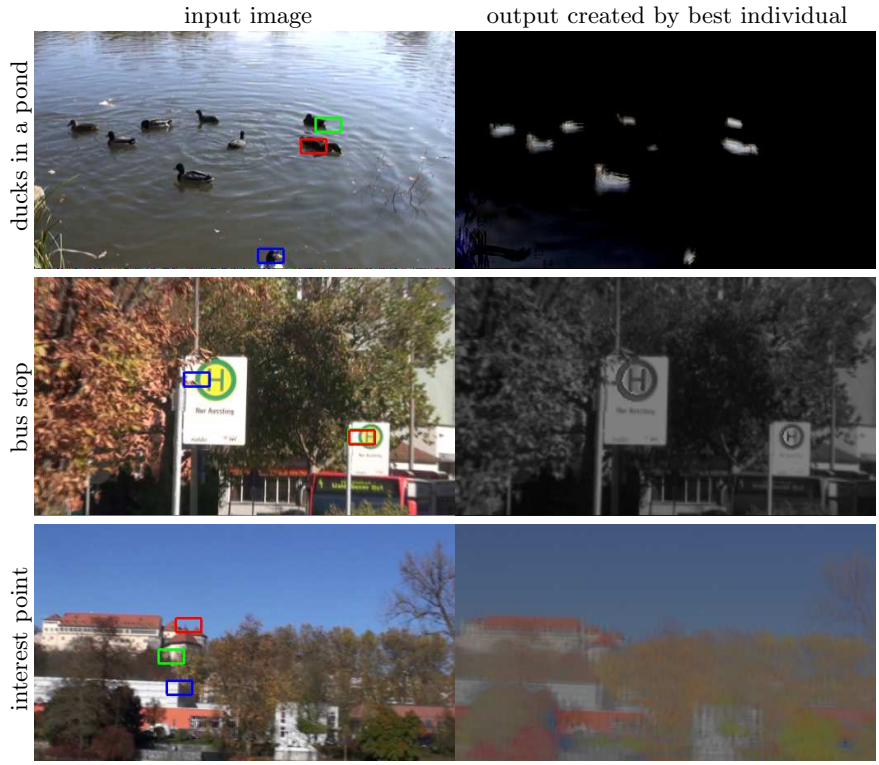

Fig. 6. Output of three evolved object detectors. The located output of the three best evolved individuals are marked in red, green, and blue respectively. In all three cases, we were able to evolve an individual which is able to detect the object or part of the image which should be located.

the next (this is also the case for a stationary camera and a stationary object due to noise in the sensor). A successful detector must therefore become robust against small distortions or noise in the data.

We were able to evolve detectors which detect ducks in a pond, interest points on a building or traffic signs. The object was usually located with a reasonably small error after relatively short time. Manually writing detectors which perform the same task would have taken considerably more time than was required to evolve the detectors.

In some cases, the object color provided a strong cue on where the object is located. However, we were also able to show that it is possible to evolve detectors which were not based on color. Figure 6 shows three different image sequences which were used to test our system. The task for the first sequence was to detect ducks in a pond. The task for the second image sequence was to detect the sign of a bus stop. The task for the third sequence was to detect an interest point located on a building. In many cases, we were surprised on the robustness of the

detectors. Quite often, the detectors were able to tolerate medium changes in appearance and/or the scale of the object.

Our system is able to achieve a frame rate of 4.5 Hz while evaluating 23 alternative individuals in the background and also visualizing the results of the three best individuals. This frame rate is achieved with a $2+2\times2$ representation on an Intel Core 2 CPU running at 2.13GHz and a GeForce 9600GT/PCI/SEE2. In other words, more than one hundred individuals are evaluated per second. The image sequences had a size of 320x240.

6 Conclusions

We have shown how a real-time evolutionary system can be built based on the OpenGL shading language which evaluates multiple alternative algorithms in the background for every input image. Our system is based on a Cartesian Genetic Programming representation. High-level image processing operations are used as input nodes. The output of these operations is then recombined using elementary functions which are available from the OpenGL shading language. At present, it is not possible to apply multiple image processing operations such as edge detection or a convolution in sequence without transferring the output of one operation back to the CPU. Our representation is streamlined to match the architecture of the GPU hardware and thereby fully exploiting the power of the GPU. With future GPU hardware, it could be possible that write operations to texture are also allowed. This would increase the power of the present approach considerably.

References

1. Ebner, M.: An adaptive on-line evolutionary visual system. In Hart, E., Paechter, B., Willies, J., eds.: Workshop on Pervasive Adaptation, Venice, Italy, IEEE (2008) (in press)
2. Lohmann, R.: Bionische Verfahren zur Entwicklung visueller Systeme. PhD thesis, Technische Universität Berlin, Verfahrenstechnik und Energietechnik (1991)
3. 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 1996, Proc. of the 1st Annual Conf., Stanford University, Cambridge, MA, The MIT Press (1996) 309–314
4. 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
5. 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 Europ. Workshop on Genetic Programming, Paris, France, The University of Birmingham, UK (1998) 6–10
6. Katz, A.J., Thrift, P.R.: Generating image filters for target recognition by genetic learning. IEEE Transactions on Pattern Analysis and Machine Intelligence **16**(9) (1994) 906–910

7. 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 Proceedings of the 1st Europ. Workshops on Evolutionary Image Analysis, Signal Processing and Telecommunications, Göteborg, Sweden, Berlin, Springer-Verlag (1999) 74–89
8. 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
9. 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
10. 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
11. Krawiec, K., Bhanu, B.: Visual learning by evolutionary and coevolutionary feature synthesis. IEEE Transactions on Evolutionary Computation **11**(5) (2007) 635–650
12. 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
13. 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 International Congress on Evolutionary Computation, San Francisco, CA, IEEE (2006) 1316–1323
14. Cagnoni, S.: Evolutionary computer vision: a taxonomic tutorial. In: 8th Int. Conf. on Hybrid Intelligent Systems, Los Alamitos, CA, IEEE Comp. Society (2008) 1–6
15. Mussi, L., Cagnoni, S.: Artificial creatures for object tracking and segmentation. In: Applications of Evolutionary Computing. Proc. EvoWorkshops 2008, Naples, Italy, Berlin, Springer (2008) 255–264
16. Mitchell, M.: An Introduction to Genetic Algorithms. The MIT Press, Cambridge, MA (1996)
17. Rechenberg, I.: Evolutionsstrategie '94. frommann-holzboog, Stuttgart (1994)
18. Koza, J.R.: Genetic Programming. On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge, MA (1992)
19. 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)
20. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proc. of the Genetic and Evolutionary Computation Conf., San Francisco, CA, Morgan Kaufmann (1999) 1135–1142
21. Ebner, M.: Color Constancy. John Wiley & Sons, England (2007)
22. Rost, R.J.: OpenGL Shading Language. 2nd ed., Addison-Wesley, Upper Saddle River, NJ (2006)
23. 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
24. 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
25. NVIDIA: Compute Unified Device Architecture. Programming Guide V.1.1. (2007)