2.8 Physics-based Vision: Active Contours (Snakes)

Physics-based Vision is a branch of Computer Vision that became very fashionable around 1987. The basic idea of Physics-based Vision is to pose a vision problem as a physics problem. The resulting algorithms are very intuitive in contrast to the 'magic', non-intuitive selection of parameters for many other algorithms in Computer Vision. The best example of Physics-based algorithms are Active Contours, usually called 'Snakes'. Snakes are very popular because they are easy to use and reasonably fast.

2.8.1 Snakes - Examples and Physical Model

Snake behavior - a first intuition

Before we start to explain the physical intuition behind Snakes and the algorithms that allow to compute them efficiently, we should first look at two examples of Snakes. Figure

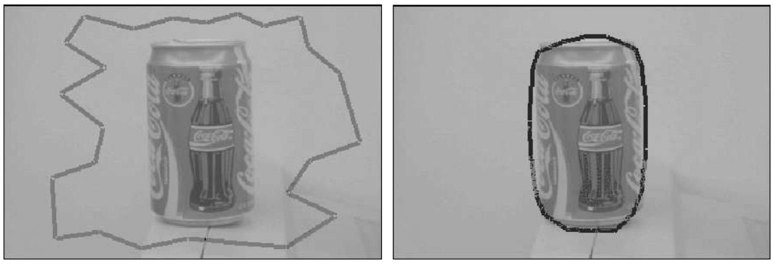

Figure 1: Initial and final position of a contour finding snake.

1 shows a snake that finds object contours. To initialize the snake, the user draws an approximate boundary of the object. This initial configuration is shown in the left image of Figure 1.The snake then behaves roughly like a rubber band that snaps onto the object boundaries if released (only much slower). This behavior of the snake is caused by an 'external energy' term that in this case determines that the snake feels attracted to object boundaries. The rubber-band metaphor is somewhat misleading because the snake would snap to the boundaries even when started from 'within' the object.

However, if you look carefully at the image on the right, you will see that it is not entirely true that the snake aligns itself with the object boundary : at the top of the can, the snake does not delineate the can contours but bulges a little bit. This behavior of the snake is caused by the second energy term, the so-called internal energy. The internal energy gives the snake the physical properties of a willow rod[1]. If you try to wrap a willow branch around a coke can, it will refuse to bend into sharp corners but will instead bulge like the snake in the image on the right-hand side.

The second example (figure 2) shows a snake that is able to track moving objects. The input was a random dot 'movie' of a sliding square. The images in the top row are samples

[1]Physicists prefer to think of thin plates instead.

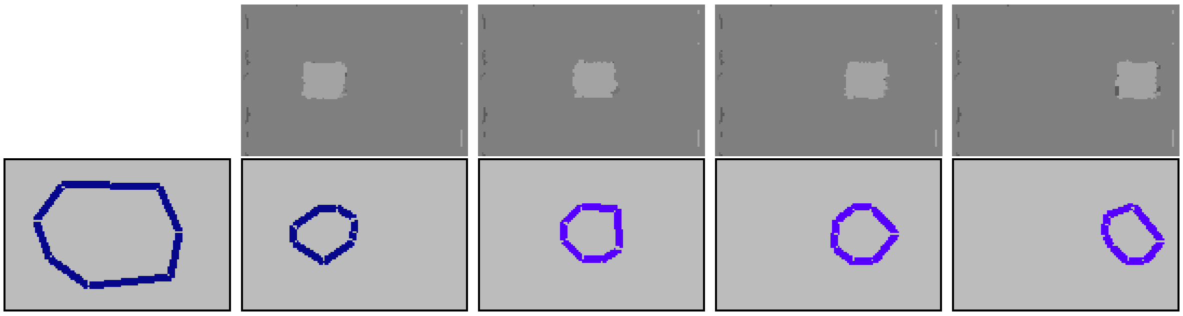

Figure 2: A snake that tracks motion in random dot 'movies'.

of the approximate position of the square during the 'movie'..[2] The second row shows the initial snake in the first image and the positions of the snake following the sliding square. The snake is tracking the moving square because the external energy was defined such that the snake feels attracted to boundaries of motion fields. This external force would shape the snake into a square, were it not for the internal energy that makes the snake bulge instead.[3]

The Physics behind Snakes

Phrased in a more technical way, Snakes are energy-minimizing splines that simulate the behavior of closed springs rolling downhill on a hilly terrain. This leads us directly to the continuous model for Snakes.

Consider a 2-D curve V parameterized by the arclength s:

$$V(s) = (x(s), y(s)), \quad for\ s \in [0,1]$$

We wish to minimize the energy E given by:

$$E = \int_0^1 E_{int} + E_{ext} ds$$

where E_{int} and E_{ext} represent the energy associated with the "internal" and "external" forces that act on the snake. Internal forces are the ones that impose constraints on smoothness, while external forces arise from the data. These terms are usually weighted such that the emphasis on smoothness is appropriate for the application, but the weighting factor λ is hidden in one of the two terms and does not appear explicitly.

The external energy also depends on the application but is usually rather straightforward to model. For example, we obtain an energy function for the contour finding snake by assuming that object contours correspond to sudden intensity changes in the image. How pronounced such intensity changes are is measured by the magnitude of the intensity gradient. Therefore, we can model the external energy as $E_{ext} = -|\nabla I|$, which corresponds to a hillside with valleys at the location of edges. Figure 4 shows the image of a cat, the corresponding

[2] Technically, the images are samples from the series of disparity maps in x direction that were computed between every pair of successive frames of the random dot movie.

[3] The snake also would approximate the shape of the square better if it had more nodes.

image of the (squared) gradient magnitudes and and edge image resulting from thresholding the gradient magnitude.

The internal energy is modeled in terms of the derivatives of the curve. A spring-like,

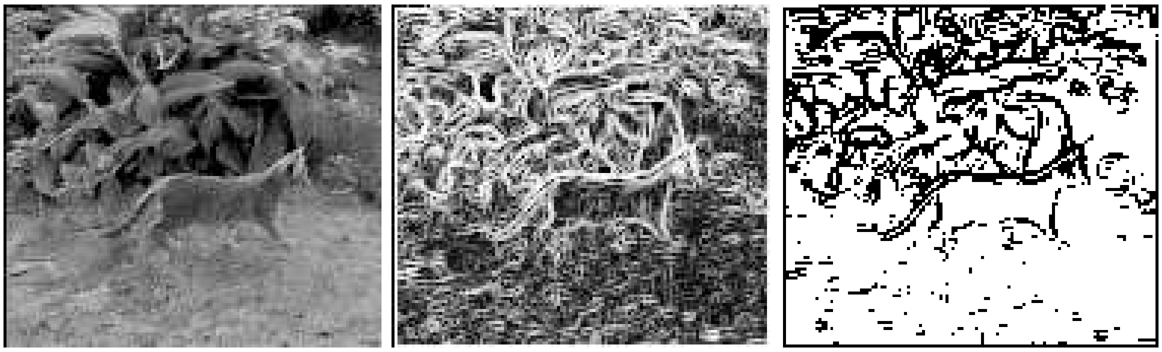

Figure 3: An image of a cat, the corresponding squared gradient magnitudes (enhanced) and the edge image.

contracting behavior corresponds to the minimization of first derivatives, while the thin-plate behavior that avoids sharp bends is modeled as the minimization of second derivatives. Therefore, the internal energy is

$$E_{int} = \alpha\,|V_s(s)|^2 + \beta\,|V_{ss}(s)|^2$$

where

$$V_s = \frac{\partial}{\partial s}V(s) \quad V_s = \frac{\partial^2}{\partial s^2}V(s)$$

The framework of snakes is actually more general, and would allow us to have coefficients $\alpha(s)$ and $\beta(s)$ that are dependent on s. But this is never used in practice and we will not consider it here any further.

Interpretation of the Parameters

Parameters α and β respectively control the sensitivity with respect to the first and the second derivative, which physicists call the *membrane* term, and the *thin plate* term, respectively. We can distinguish the following cases (see Fig. 4):

(a) $[\alpha = \beta = 0]$ The internal force is zero, and the curve does not have to be continuous ($V \in C^0$). In other words, the snake can even have sharp angles.

(b) $[\alpha > 0, \beta = 0]$ This imposes that V_s be bounded, and thus that $V \in C^1$, i.e. continuous and once differentiable.

(c) $[\beta > 0]$ This forces V_{ss} to be bounded, which implies that $V \in C^2$, i.e. 2 times differentiable and once continuously differentiable.

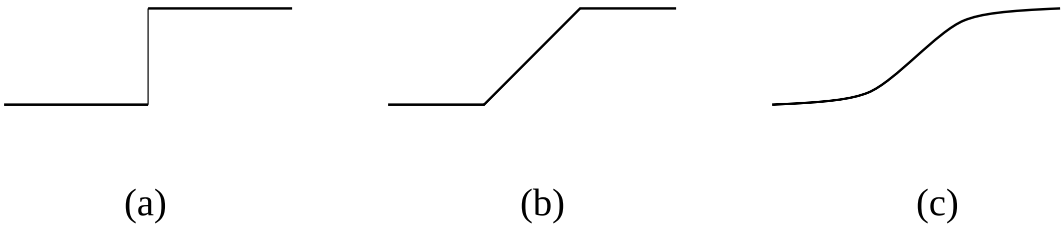

Figure 4: Examples of (a) discontinuous, (b) continuous, and (c) continuously differentiable curves.

Relative values of α, β control the snake's "stiffness".

At this point, we are left with an energy minimization problem:

$$\text{Find } V \text{ that minimizes } \int_0^1 E_{int} + E_{ext}\, ds$$

2.8.2 Computing Snakes

In the continuous case, this minimization problem can be solved using the Calculus of Variations. The solution has to satisfy the following set of equations:

$$X_t = (A - \gamma\, I)^{-1}\,(\gamma\, X_{t-1} - \frac{\partial}{\partial x} E_{ext}(old))$$

$$Y_t = (A - \gamma\, I)^{-1}\,(\gamma\, Y_{t-1} - \frac{\partial}{\partial y} E_{ext}(old))$$

where γ is the step size, and A is a pentadiagonal matrix formed from α, β. These equations are solved by an iterative method. However, this approach for computing Snakes has several problems, and today, Snakes are usually computed using a dynamic programming technique known as the Viterbi Algorithm. We will start the discussion of this algorithm by a short introduction to Dynamic Programming in general.

Dynamic Programming

The main idea behind dynamic programming is to compute the solution of a problem by combining solutions of subproblems. What makes Dynamic Programming different from Divide-and-Conquer is that many of the subproblems overlap or occur multiple times. The main speed-up is then gained by computing solutions to these subproblems only once and reusing them wherever they occur. A very simple example of this is the computation of a Fibonacci number. Fibonacci numbers are defined recursively by

$$F_0 = 0; \quad F_1 = 1; \quad F_i = F_{i-1} + F_{i-2}$$

Obviously, all subterms $F_0, ...F_{i-2}$ are computed multiple times if F_i is implemented as a recursive function. However, if you compute the Fibonacci numbers bottom-up (in the order i= 2,3,4..) and store all results in a table[4], no redundant computations are made.

[4] In this particular example, you only need to remember the last two values, but for other problems, you usually have to store a whole table of values.

Usually, dynamic programming is applied to complex optimization problems. We will demonstrate the power of dynamic programming with the example of graph coloring. If you do not have background in Graph Theory, you may not understand some of the expressions, but the basic idea should come across anyway.

Solving the Graph Coloring Problem with Dynamic Programming

The particular variant of the Graph Coloring Problem that we are considering here asks in how many ways a given graph can be 4-colored. The general problem is NP-hard and the computation of its solution takes $O(4^n)$. More formally,
Let G(N,E) a graph with node set N and edge set $E \in N \times N$. *How many 4-colorings* $C : N \rightarrow \{Red, Gree, Blue, Yellow\}$ *exist such that* $\forall(n_1, n_2) \in E : C(n_1) \neq C(n_2)$.

In the general case, you cannot do much better than enumerating all combinations of color assignments for all nodes and to count the number of combinations for which no two neighbored nodes have the same color. However, there are some special cases for which the complexity of the computation can be reduced considerably. An example of such a graph is shown in figure 5. The key observation here is that the nodes 7 and 8 form the only

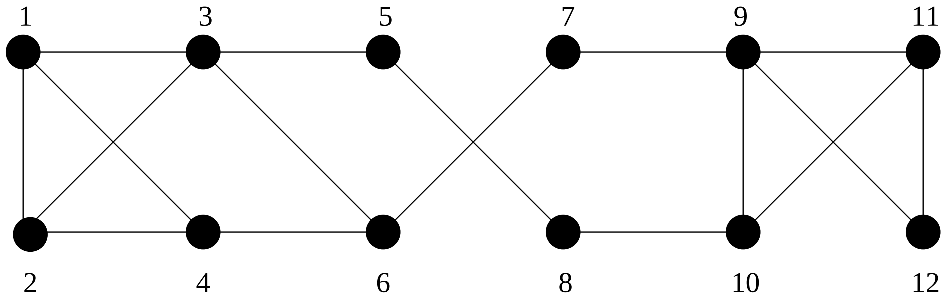

Figure 5: An example graph.

connection between the subgraph on the left and the subgraph on the right. This allows us to reduce the problem to the solution of the problem for the left subgraph and for the right subgraph as follows:

$$\sum_{C(n_7)=Red}^{Yellow} \sum_{C(n_8)=Red}^{Yellow} |(colorings\ of\ n_1, ..n_6)| * |(colorings\ of\ n_9, ..n_{12})|$$

That is, for each of the 16 different ways to color node 7 and 8, we compute the number of possibilities to color the graph by computing the number of ways to color each subgraph, multiplying these two numbers and adding up the results for all 16 different colorings.

In order to appreciate the complexity reduction, we can look at the problem in terms of a search tree. In order to solve the general problem, we would have to enumerate all possible color combinations in a systematic manner. This leads to a search tree of the following form (see Fig. 6). The tree has n levels, and at expansion level k, the graph nodes 1..k-1 are colored. Now assume we have expanded the tree until level 7, which means that nodes 1-6 are already colored. At level 7, there are 6^4 search nodes and in brute force search, we would have to expand each of these nodes into the complete tree of all different combinations of

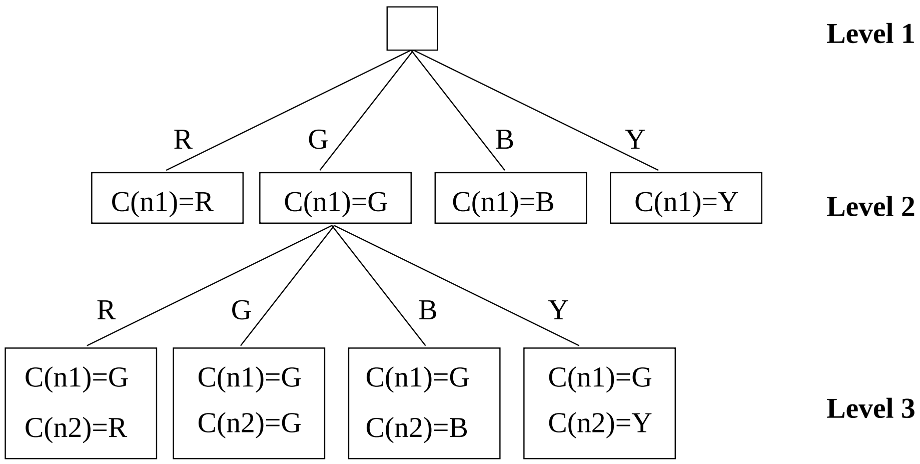

Figure 6: The search graph for the 4-coloring problem of the above graph.

ways to color the nodes 7-12. However, we have already noticed that the colorings of the nodes 9-13 depends only on the colors of nodes 7 and 8. This means that we can put the 6^4 nodes into 16 equivalence classes and expand only 16 search trees for the subgraph on the right instead of more than 1000.

We have chosen this rather complicated example from graph theory because it is actually quite similar to the Viterbi algorithm. The Viterbi algorithm operates on discretized Snakes, i.e. we chop the snake into small pieces and regard the positions where we cut the Snake as nodes. The Snake pieces between the nodes are then approximated by simple functions like polynoms. This approximation of a function by a chain of short pieces of simple functions is called a *spline*. For all nodes of the spline holds that their contribution to the overall energy of the Snake only depends on their position relative to their immediate predecessor and successor nodes in the chain. This is similar to the graph coloring problem where the color of each node was constrained by the color of its immediate neighbors and we can apply a similar trick. The following paragraph will clarify how the Viterbi algorithm works.

2.8.3 Viterbi algorithm

We already mentioned that we have to discretize the Snake before we can apply the Viterbi algorithm. A discretized Snake is a spline with vertex positions $V = v_1, v_2, ...v_n$. We also have to discretize the energy terms associated with a particular snake. The discretization of the external energy depends on the particular energy term, but the internal energy is always the same and can be discretized as follows:

$$E_{int}(v_i) = \alpha\,|v_i - v_{i-1}|^2 + \beta\,|v_{i+1} - 2\,v_i + v_{i-1}|^2$$

where v_i denotes the position of the ith vertex of the snake. Our problem reduces to finding the set of vertex positions V that minimizes

$$\sum_{v_i \in V} E_{int}(v_i) + E_{ext}(v_i)$$

In order to make the presentation of the algorithm easier, we will now make the assumption that the snake is open and that $\beta = 0$. We will relax these two assumptions later on. We also assume that the snake vertex can only move to m nearby locations in order to make the minimization problem tractable.

First, observe that since the internal energy is a local property of neighboring vertices, we can decompose it into a sum of local terms:

$$E(v_1, \ldots, v_n) = E_1(v_1, v_2) + E_2(v_2, v_3) + \ldots + E_{n-1}(v_{n-1}, v_n)$$

where $E_i(v_{i-1}, v_i) = E_{ext}(v_i) + E_{int}(v_{i-1}, v_i)$. In particular, this means that each vertex position v_i influences the total energy only through the terms E_{i-1} and E_i. The Viterbi algorithm capitalizes on the property that the influence of the vertex positions is so decoupled.

Now, we introduce the intermediate variables s_i (i=2..n) defined by

$$s_2(v_2) = \min_{v_1} E_1(v_1, v_2)$$

$$s_3(v_3) = \min_{v_2}(s_2(v_2) + E_2(v_2, v_3))$$

$$s_4(v_4) = \min_{v_3}(s_3(v_3) + E_3(v_3, v_4))$$

$$\vdots$$

$$s_n(v_n) = \min_{v_{n-1}}(s_{n-1}(v_{n-1}) + E_{n-1}(v_{n-1}, v_n))$$

Each $s_k(v_k)$ contains the lowest total energy for the first k-1 vertices of the snake for a given value of v_k. Thus, the minimum energy E of the whole snake is equal to $\min_{v_n}(s_n(v_n))$.

The globally best position for the snake is therefore computed by first computing all the s_k, which means that we determine at each node the optimal position of its predecessor for each possible location of the node under consideration. When we have computed s_n, we can find the optimal position for v_n by minimizing the expression $s_n(v_n)$. Once we now the position of the last node, we look up the optimal position for the second last node, and so on until we have determined the optimal position for all the nodes.

To further improve the efficiency, we consider typically only $m = 9$ possible positions for each vertex when searching for the local minimum (see Fig. 7). In order to allow the snake to move more than 1 pixel per vertex, we iterate the above procedure using the previous result as the new starting position, until the snake has converged to a quasi-globally optimal position. Snakes are not performing a truly global optimization, but they find the optimal position of the snake within the search window and the search window is repositioned after each iteration.

Relaxing the assumptions

If $\beta > 0$, then the decomposition of the energy function into local terms becomes

$$E(v_1, \ldots, v_n) = E_1(v_1, v_2, v_3) + E_2(v_2, v_3, v_4) + \ldots + E_{n-2}(v_{n-2}, v_{n-1}, v_n)$$

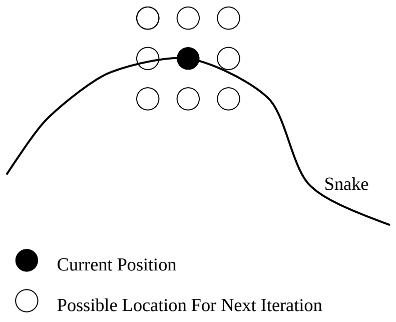

Figure 7: Minimization within a 9-pixel window.

Therefore, we compute the optimal position of the predecessing node for all possible positions of the node under consideration *and* the following node:

$$s_2(v_2, v_3) = \min_{v_1} E_1(v_1, v_2, v_3)$$

$$s_3(v_3, v_4) = \min_{v_2}(s_2(v_2, v_3) + E_2(v_2, v_3, v_4))$$

$$\vdots$$

$$s_{n-1}(v_{n-1}, v_n) = \min_{v_{n-2}}(s_{n-2}(v_{n-2}, v_{n-1}) + E_{n-2}(v_{n-2}, v_{n-1}, v_n))$$

where $E_{i-1}(v_{i-1}, v_i, v_{i+1}) = E_{ext}(v_i) + E_{int}(v_{i-1}, v_i, v_{i+1})$. Note that we have to compute a table of m^2 values at each node instead of only m values per node for the simpler snakes with $\beta = 0$.
The adaptation of the algorithm to a closed snake is part of the homework assignment.

2.8.4 Advantages and Disadvantages of Snakes

The most salient advantage of Snakes besides their highly intuitive behavior is their efficiency. Snakes are are fast for three reasons:

1. Snakes are 1-dimensional which means that we can reduce 2-dimensional optimization problems to 1-dimensional optimization problems.

2. Snakes optimize locally.

3. It is possible to apply dynamic programming techniques, which reduces the complexity from $O(m^n)$ for naive search to $O(m^2 * n)$ for Snakes with $\beta = 0$ and to $O(m^3 * n)$ for Snakes with $\beta > 0$.

In the original paper, Kass and Terzopolous also mention that snakes make the combination of low-level and high-level reasoning processes easy, because they can be implemented as an interactive technique where the user (or a high-level reasoning system) can provide feedback during the optimization process by adding additional energy terms. In this respect, Snakes have not quite lived up to their promises. However, it is a real advantage of the dynamic programming formulation of Snakes that it is easy to impose hard constraints like 'the snake must not have knots' or 'vertices cannot come closer than distance d', which would be impossible in the Calculus of Variations framework.

The only disadvantage of Snakes is that the Viterbi algorithm trades space for time and that it can be quite space consuming.

2.8.5 Applications and Extensions

We have already seen in the introduction that Snakes can be used to find contours. Kass and Terzopolous show in their paper how they used contour finding snakes to track lips. This is especially impressive because the lips do not only move but also change their shape. Motion algorithms that rely on correspondence have often difficulties with deformations. Edge-attracted snakes can even be used to locate illusory contours as shown in Figure 8. They are also widely used in bio-medical applications for outlining organs or tumors in X-rays and magnetic resonance tomograms (MRTs).

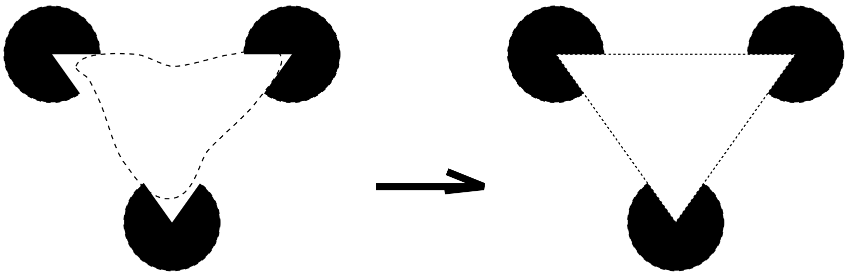

Figure 8: A snake for perception of illusory contours

When snakes are applied in technical domains, e.g. locating planes and houses on satellite imagery, it is desirable that the snakes can form corners. One could set β to zero, but then the snakes cannot delineate curved objects adequately anymore. A better solution is to use other types of splines instead of the simple quadratic splines.

Another variation of the basic idea are balloons which are more or less the 3-dimensional equivalent of snakes and are used to locate the 3D surface of objects, for example from range data.