Interactive Evolution of Camouflage

Technical Report IEC-2021-1

October 2, 2021 version 1

Craig Reynolds

cwr@red3d.com
https://www.red3d.com/cwr/

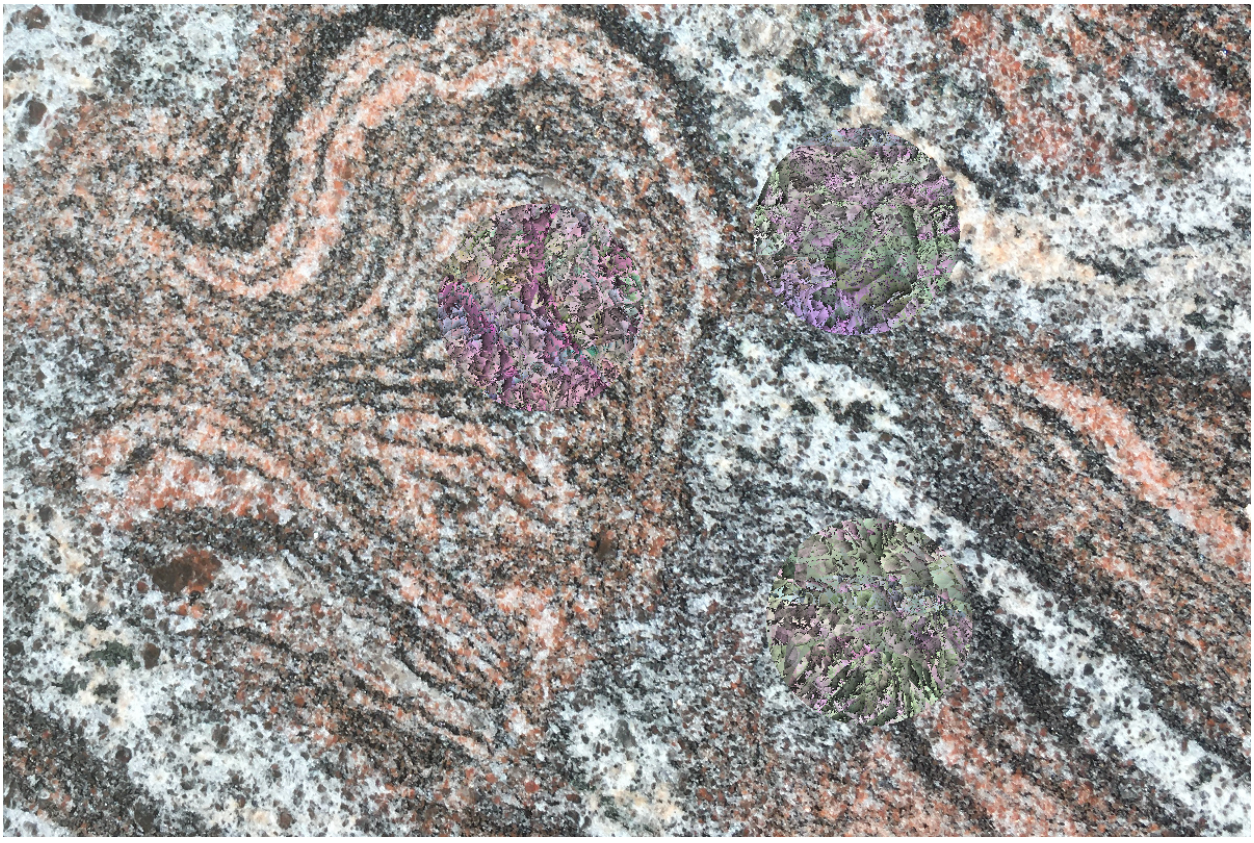


Fig. 1: photograph of a polished granite counter-top in bright sun, overlaid with three synthetic “prey,” circular samples of evolved camouflage texture.

Abstract

A simple computer simulation of the evolution of camouflage in nature is described. Several examples are given. The model is based on a game-like concept with a human "player" in the loop to act as a predator. This is a reimplementation of a technique first implemented in 2009 [???]. The basic concepts are the same between the earlier work and this new implementation. The point of this work was to write a clean modern software infrastructure, to verify the old results could be reproduced, and to then move on new topics in this general area of research.

Background

This report describes experiments in generating 2d camouflage patterns in response to 2d background textures. In these experiments, the backgrounds are all photographs of the natural world, taken with a mobile phone.

After a brief description of the software used, results of interactive "runs" will be presented, followed by some conclusions.

Description of software

These experiments use three primary software components. First is a library for procedural texture synthesis called *TexSyn*. Second is a general purpose engine for optimization via evolutionary computation, called *LazyPredator*. And third is a simple interactive application that uses those components to make a game-like simulation where camouflage evolves in response to selection pressure from a predator played by the human.

App

That interactive app is defined by a c++ class called *EvoCamoGame* and is invoked from the unix style command line (maxOS *Terminal* app) with an executable called `evo_camo_game`.

Procedural texture synthesis

TexSyn is a library for procedural texture synthesis. It defines objects representing an abstract model of two dimensional color textures. Common digital images are defined by a rectangular array of pixel values each representing a color. *TexSyn*'s textures are purely procedural: defined by code. These textures represent an infinite 2d plane. For any given location on that plane the texture can return a color.

In c++ terms: these textures are represented by instances of type *Texture,* a base class. *Texture* defines a "virtual member function" called *getColor()* which maps a *Vec2* position into a *Color* value. (*Vec2* is defined by *x*, *y* Cartesian coordinates as unrestricted floating point numbers. *Color* is defined by *red*, *green*, and *blue* components, as unrestricted floating point numbers, where monitor gamut is the positive unit cube.) *TexSyn* defines about fifty specialized types derived from *Texture*. They each "override" the *getColor()* function. A few define a texture from only numeric parameters, such as *Uniform,* which has the same color everywhere on the plane. Most of *TexSyn*'s classes are *texture operators* which take one or more *Textures* as parameters. Complex *TexSyn* textures are nested expressions — "trees" — which describe combinations of many texture operators. Typically these trees have between 100 and 200 operators (plus various numeric values at the leaves) but there is no explicit upper bound on size.

[??? should I give an example of code and image here, or save that for a later more detailed section ???]

The development of *TexSyn* since late 2019 has been documented in this development log. The open source code for *TexSyn* is in a publicly available repository at GitHub.

Optimization

In these experiments, an *optimization* process is used to find textures that meet certain criteria. Generally this means to start from some sort of initial guess, often random, and incrementally improving it to be closer to a goal. In this work the initial guess is a "random texture"—perhaps better to say an arbitrary texture, sampled from a very large distribution of possible textures. This universe of textures is defined by nested expressions (trees) of *TexSyn* operators (generally smaller than a given maximum size).

There are many kinds of analytic numeric optimization processes used across various fields, with a long history going back to *Newton's method*. Widely known are the gradient descent techniques used in very high dimensional space for modern *deep learning*.

A much more ancient method for optimization is evolution in the natural world. While not strictly "optimization" from a mathematical perspective, the process of evolution explores spaces of vast dimensionality to produce well adapted and complex living organisms. In nature this exploration is based primarily on genetic variation from crossover and mutation of DNA—and the evolutionary developmental processes ("evo-devo") that lead to the morphology of living creatures. Resulting organisms compete for survival and resources. Those who thrive have increased "reproductive success" and so pass on more of their genes to their offspring.

Complex natural organism such as animals have nervous systems that learn and reason from experience. Stepping back, it is possible to see intelligence (during an animal's lifetime) and evolution (across many generations of animals) as related phenomena. They are both processes of *self organization*. This is the perspective of Leslie Valiant's engaging 2013 book *Probably Approximately Correct (Nature's Algorithms for Learning and Prospering in a Complex World)*.

Similarly, we can draw a rough equivalence between computer optimization algorithms inspired by learning and those inspired by evolution.

Evolutionary computation

Many types of nature-inspired algorithmic techniques have been applied to difficult computational optimization problems. One category is known as *evolutionary computation* or *evolutionary algorithms*. Within this category are a class of *population based* models. Best known is the *genetic algorithm*. The work described in this document uses a closely related technique called *genetic programming*.

In both cases a *population* of *individuals* (candidate solutions) is maintained. Any population size can be used but somewhere between 100 and 1000 are typical. The evolutionary computation proceeds by testing the *fitness* of individuals, using this metric to choose parents, who reproduce by crossover and mutation of their representation ("genes") to form new individuals ("offspring"). Older, or lower fitness individuals are replaced by these offspring. This updating process can happen in parallel to all individuals in a population (*generational*) or it can proceed individual by individual (*steady state*).

Evolutionary computation is directed by a *fitness* metric which can take many forms. (Fitness is analogous to loss or error metrics used in other optimization techniques.) It can be *absolute* by assigning a number to a given individual. Or it can be *relative*, being defined by comparing two or more individuals, essentially sorting them by their relative fitness. This allows using fitness measures that are difficult to quantify, but can be determined through competitions. Sometimes fitness is *procedural*, defined by software, or it can be *interactive*, with a human in the loop.

Genetic programming

This work uses the *genetic programing* variant of *evolutionary computation*. In *genetic algorithms* the genome is represented by a sequence of numbers, a *vector*. In genetic programming the genome is a *tree*, normally interpreted as a nested expression—a "program"—in some domain-specific language.

The *TexSyn* library for procedural texture synthesis defines an interface with the *LazyPredator* system for genetic programming. This allows textures to be discovered according to how well they match a given fitness requirement. TexSyn can use LazyPredator and the interface to make a population of textures. Then a *run* incrementally tries to improve the population according to a fitness function passed as a parameter to the run. For applications based on relative fitness, instead of a fitness function, a *tournament function* is provided to the run. (So far in LazyPredator, tournaments always consist of three individuals, which are drawn uniformly from the population.)

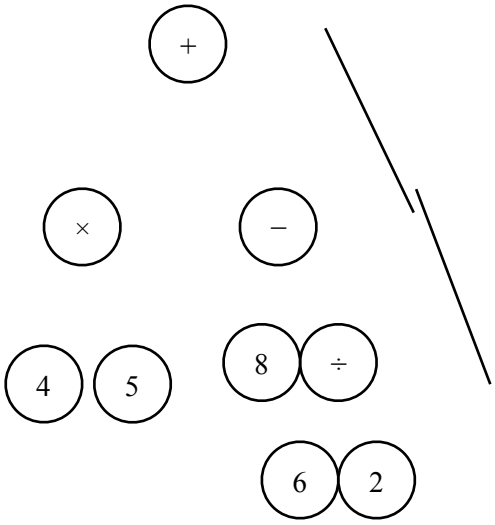
In general, as the run progresses, higher fitness textures survive, lower fitness textures die, and are replaced by new offspring textures, which are formed by *crossover* and *mutation*. In genetic programming, crossover is a sort of random, syntax-aware, cut-and-paste: a portion (subtree) of one parent is replaced by a portion (subtree) of the other parent. Mutation involves adding noise to numeric constants at the leaves of GP trees.

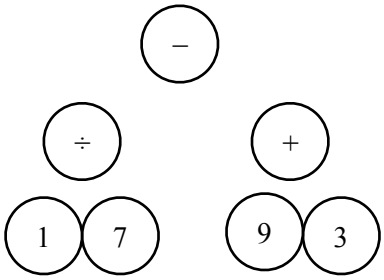
[fig GP crossover and mutation?]

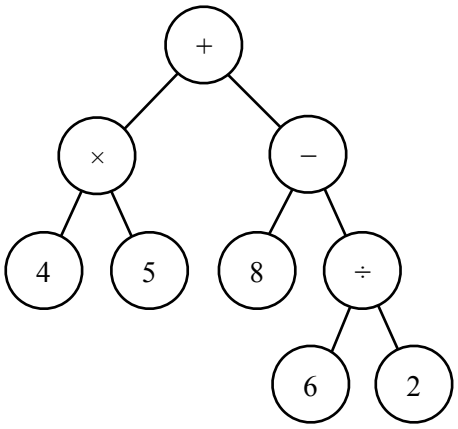
OpenCV

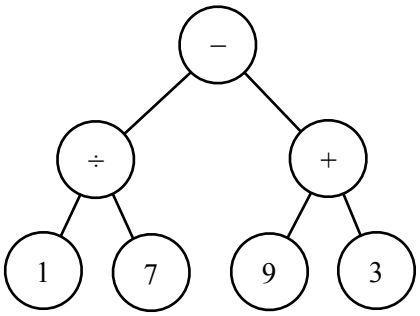
…

Draft for GP crossover diagram:









old:

This is an alpha test version of software for “interactive evolution of camouflage” as described in my 2011 paper: Interactive Evolution of Camouflage. This 2021 app evo_camo_game is built from new components: the TexSyn library for procedural texture synthesis, and the LazyPredator library for evolutionary optimization via genetic programming. For more information see the development blog for TexSyn.

Very quick overview

This program is a crude simulation of the evolution of camouflage in nature. There is a predator-prey system. Software for texture optimization plays the part of an evolving population of camouflaged “prey.” The human user serves as a predator hunting its prey with vision. This can be seen as a sort of minimalist “game” or a human based computation. The app displays a window with a random portion taken from a given set of photographs. Over that background are drawn three randomly positions disks of synthetic texture—three “prey.” The human user/player/predator then decides which of the three textures/prey is most conspicuous or least well camouflaged, indicating their selection by clicking/tapping on that prey. The window will go blank then display the next step. Runs typically consist of 1000 such steps or more. They can be stopped at any time. Results are currently saved during as image files, and texture “source code” in text files, as described below.