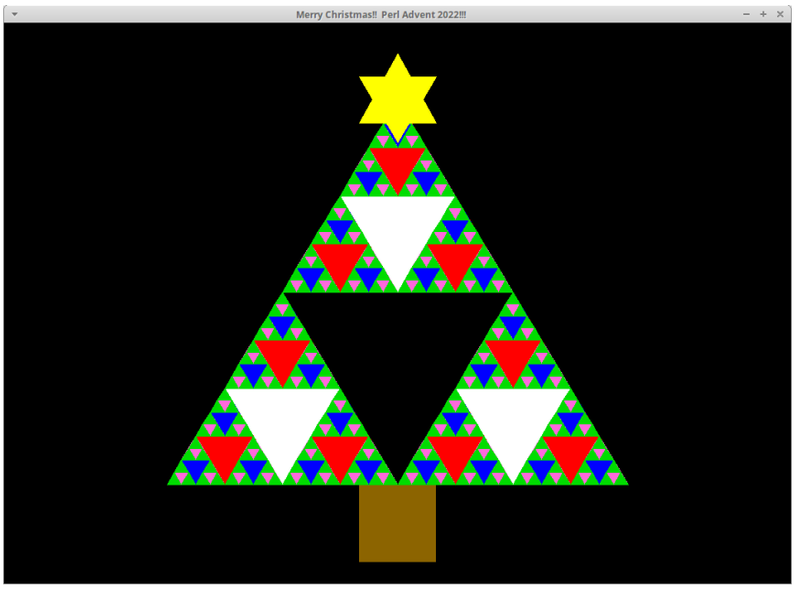
Fractal Christmas Tree

MathPerl::Fractal::ChristmasTree - 2022-12-01

Merry Christmas and Happy New Year! :-)

Our holiday-themed programming goal is to implement a fun little Perl application, which will display a Christmas tree using the Sierpinski fractal algorithm.



Let's begin by reviewing the code for generating Sierpinski Triangle fractals, stored in a file named Sierpinski.pm.

If you are an experienced Perl programmer, you will immediately notice the definition and utilization of data types such as `integer` and `number`, as well as data structures such as `integer::arrayref` and `integer::arrayref::arrayref` etc. Data types and data structures, along with other Perl programming strategies such as `CRITICS` as well as subroutine `$RETURN_TYPE` and `@ARG` named input arguments, are included for best practices and compatibility with the Perl compiler.

```
# [[[ PREPROCESSOR ]]]
# declare Perl-compatible data types & data structures
package void; 1;
package integer; 1;
package integer::arrayref; 1;
package integer::arrayref::arrayref; 1;
package number::arrayref; 1;
package number::arrayref::arrayref; 1;
package number::arrayref::arrayref::arrayref; 1;
package number::arrayref::arrayref::arrayref::arrayref; 1;

# [[[ HEADER ]]]
#use RPerl;  # replaced by PREPROCESSOR for simplicity
package MathPerl::Fractal::Sierpinski;
use strict;
use warnings;
use v5.14;  # required for /r return AKA non-destructive regex flag
our $VERSION = 0.008_000;

# [[[ CRITICS ]]]
# USER DEFAULT 1: allow numeric values & print operator
## no critic qw(ProhibitUselessNoCritic ProhibitMagicNumbers RequireCheckedSyscalls)
## no critic qw(RequireInterpolationOfMetachars)  # USER DEFAULT 2: allow single-quoted control characters & sigils
## no critic qw(ProhibitConstantPragma ProhibitMagicNumbers)  # USER DEFAULT 3: allow constants

# [[[ OO INHERITANCE ]]]
#use parent qw(MathPerl::Fractal);  # disable unnecessary inheritance for simplicity
#use MathPerl::Fractal;

# [[[ INCLUDES ]]]
use English;
use Data::Dumper;
$Data::Dumper::Deepcopy = 1;  # display human-readable numeric data, not $VAR1->[0] references

# [[[ SUBROUTINES ]]]

# recursively generate triangles, grouped by recursion level
sub sierpinski {
    { my void $RETURN_TYPE };
    (
        my number::arrayref::arrayref $triangle,
        my integer $recursions_remaining,
        my number::arrayref::arrayref::arrayref::arrayref $triangle_groups
    ) = @ARG;

    print 'in sierpinski(), received $recursions_remaining = ', $recursions_remaining, "\n";
    print 'in sierpinski(), received $triangle = ', (Dumper($triangle) =~ s/'//gr), "\n";

    if ($recursions_remaining > 0) {
        # shortcut variables, easier to read in midpoint calculations below
        my number::arrayref $point_a = $triangle->[0];
        my number::arrayref $point_b = $triangle->[1];
        my number::arrayref $point_c = $triangle->[2];

        # calculate midpoints between two coordinates [x1, y1] and [x2, y2] is [(x1+x2)/2, (y1+y2)/2]
        my number::arrayref $point_a_b =
            [(($point_a->[0] + $point_b->[0]) / 2),
             (($point_a->[1] + $point_b->[1]) / 2)];
        my number::arrayref $point_a_c =
            [(($point_a->[0] + $point_c->[0]) / 2),
             (($point_a->[1] + $point_c->[1]) / 2)];
        my number::arrayref $point_b_c =
            [(($point_b->[0] + $point_c->[0]) / 2),
             (($point_b->[1] + $point_c->[1]) / 2)];

        # construct 3 sub-triangles from orinal points and newly-calculated midpoints
        my number::arrayref::arrayref $subtriangle_a = [ $point_a,   $point_a_b, $point_a_c ];
        my number::arrayref::arrayref $subtriangle_b = [ $point_a_b, $point_b,   $point_b_c ];
        my number::arrayref::arrayref $subtriangle_c = [ $point_a_c, $point_b_c, $point_c   ];

        # $triangle_groups is zero-indexed like all other Perl arrays,
        # so we need to subtract one from $recursions_remaining before using as an index,
        # in order to avoid an undefined element at element 0;
        # also, we need to decrement $recursions_remaining before making recursive calls;
        # for both of these reasons, we can decrement now
        $recursions_remaining--;

        # store all triangles grouped by recursion level
        push @{$triangle_groups->[$recursions_remaining]}, $subtriangle_a;
        push @{$triangle_groups->[$recursions_remaining]}, $subtriangle_b;
        push @{$triangle_groups->[$recursions_remaining]}, $subtriangle_c;

        # recurse once for each sub-triangle
        sierpinski( $subtriangle_a, $recursions_remaining, $triangle_groups);
        sierpinski( $subtriangle_b, $recursions_remaining, $triangle_groups);
        sierpinski( $subtriangle_c, $recursions_remaining, $triangle_groups);
    }

    # return after maximum recursion level is reached (conditional block above not entered),
    # or all recursion calls have returned (conditional block above entered);
    # no return value, all generated data is stored directly in $triangle_groups
    return;
}

1;  # end of class
```

Let's see what happens when we call the `sierpinski()` subroutine, passing in only 1 level of recursion for simplicity...

The recursion directly populates `$retval` in reverse order, from highest index to lowest index, eventually ending at index 0 with no further recursion to be done, and all the values are returned back to the original subroutine call. Because of this reverse-index population, the hard-coded initial triangle is stored at the highest (not the lowest) index in `$retval`, as you can see in the Perl one-liner (two-liner?) below. The initial triangle's hard-coded definition is done during declaration for brevity, and the to-be-populated element is left as undef.

```
$ perl -e 'use MathPerl::Fractal::Sierpinski; my $retval = [undef, [[512, 100], [212, 600], [812, 600]]];\
    MathPerl::Fractal::Sierpinski::sierpinski($retval->[1], 1, $retval);'
```

First, `sierpinski()` will display the 3 `[ x, y ]` cartesian coordinates representing the 3 corners of our initial input triangle, received in the `my number::arrayref::arrayref $triangle` argument:

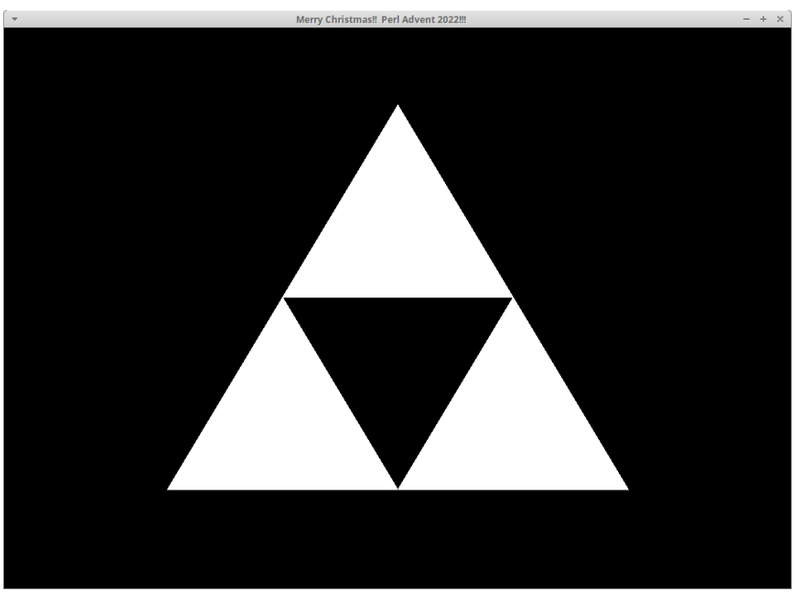
```
in sierpinski(), received $recursions_remaining = 1
in sierpinski(), received $triangle = $VAR1 = [
          [
            512,
            100
```

```
          ],
          [
            212,
            600
          ],
          [
            812,
            600
          ]
        ];
```

Then the Sierpinski algorithm creates 3 sub-triangles and makes a recursive call to `sierpinski()` for each sub-triangle:

```
in sierpinski(), received $recursions_remaining = 0
in sierpinski(), received $triangle = $VAR1 = [
          [
            512,
            100
          ],
          [
            362,
            350
          ],
          [
            662,
            350
          ]
        ];

in sierpinski(), received $recursions_remaining = 0
in sierpinski(), received $triangle = $VAR1 = [
          [
            362,
            350
          ],
          [
            212,
            600
          ],
          [
            512,
            600
          ]
        ];

in sierpinski(), received $recursions_remaining = 0
in sierpinski(), received $triangle = $VAR1 = [
          [
            662,
            350
          ],
          [
            512,
            600
          ],
          [
            812,
            600
          ]
        ];
```

If we were to render these 3 sub-triangles in white, it would look like a monochrome triforce:



After all recursion has completed, we are left with our final `number::arrayref::arrayref::arrayref::arrayref` data structure, which contains all generated triangles grouped by recursion level:

```
have $my_triangle_groups =
$VAR1 = [
          [
            [
              [
                512,
                100
              ],
              [
                362,
                350
              ],
              [
                662,
                350
              ]
            ],
            [
              [
                362,
                350
              ],
              [
                212,
                600
              ],
              [
                512,
                600
              ]
            ],
            [
              [
                662,
                350
              ],
              [
                512,
                600
              ],
              [
                812,
                600
              ]
            ]
          ],
          [
            [
              [
                512,
                100
              ],
              [
                212,
                600
              ],
              [
                812,
                600
              ]
            ]
          ]
        ];
```

Ultimately, it is the `$my_triangle_groups` data which will be rendered to comprise the main triangular body of the Christmas tree.

Next let's review the Perl code for generating the Christmas Tree data and rendering the Simple DirectMedia Layer (SDL) graphics, stored in a file named ChristmasTree.pm:

```
# [[[ HEADER ]]]
#use RPerl;  # disabled for simplicity; data types declared in Sierpinski.pm & inherited below
```

```perl
package MathPerl::Fractal::ChristmasTree;
use strict;
use warnings;
use v5.14;  # required for /r return AKA non-destructive regex flag
our $VERSION = 0.008_000;

# [[[ CRITICS ]]]
# USER DEFAULT 1: allow numeric values & print operator
## no critic qw(ProhibitUselessNoCritic ProhibitMagicNumbers RequireCheckedSyscalls)
## no critic qw(RequireInterpolationOfMetachars)  # USER DEFAULT 2: allow single-quoted control characters & sigils
## no critic qw(ProhibitConstantPragma ProhibitMagicNumbers)  # USER DEFAULT 3: allow constants

# [[[ OO INHERITANCE ]]]
use parent qw(MathPerl::Fractal::Sierpinski);
use MathPerl::Fractal::Sierpinski;

# [[[ INCLUDES ]]]
use English;
use Data::Dumper;
$Data::Dumper::Deepcopy = 1;  # display human-readable numeric data, not $VAR1->[0] references

# https://metacpan.org/dist/SDL/view/lib/pods/SDL/Event.pod
use SDL;
use SDLx::App;   # used for window creation & control
use SDL::Event;  # used for creating Event object
use SDL::Events; # used for Event queue handling functions
use Time::HiRes qw( gettimeofday usleep );  # used for time-based animation control

# [[[ SUBROUTINES ]]]

# display an animated Christmas tree!
# define hard-coded constant data; call sierpinski() to recursively generate fractal triangles;
# initialize SDL graphics; render static graphics; render dynamic graphics (animation)
sub generate_fractal__render_animation {
    { my void $RETURN_TYPE };

    # [ DATA FOR SIZES & SHAPES & COLORS;
    #   HARD-CODED 1024x768 RESOLUTION & 32-BIT COLOR DEPTH ]

    # initial triangle's 3 corners as [x, y] Euclidean coordinates
    my number::arrayref::arrayref $my_triangle_initial =
        [[ 512, 100],  # top point
         [ 212, 600],  # bottom left
         [ 812, 600]]; # bottom right

    my integer::arrayref $my_color_red     = [255, 000, 000, 255];
    my integer::arrayref $my_color_pink    = [255, 105, 220, 255];
    my integer::arrayref $my_color_orange  = [255, 150, 000, 255];
    my integer::arrayref $my_color_yellow  = [255, 255, 000, 255];
    my integer::arrayref $my_color_green   = [000, 220, 000, 255];
    my integer::arrayref $my_color_blue    = [000, 000, 255, 255];
    my integer::arrayref $my_color_purple  = [175, 000, 255, 255];
    my integer::arrayref $my_color_white   = [255, 255, 255, 255];
    my integer::arrayref $my_color_brown   = [140, 100, 000, 255];

    # colors as [r, g, b] triplets; number of colors is number of recursions
    my integer::arrayref::arrayref $my_triangle_colors =
        [ $my_color_green,  # green needs to be the color of the smallest, and thus most numerous, triangles
          $my_color_pink,
          $my_color_blue,
          $my_color_red,
          $my_color_white ];
    my integer $my_recursions_remaining = scalar(@{$my_triangle_colors});

    # rectangle in format [ x, y, width, height ]
    my number::arrayref::arrayref $my_rectangle_trunk =
        [462, 601, 100, 100];

    my number::arrayref::arrayref $my_triangle_star_up =
        [[ 512, 050],  # top point
         [ 462, 130],  # bottom left
         [ 562, 130]]; # bottom right
    my number::arrayref::arrayref $my_triangle_star_down =
        [[ 512, 155],  # bottom point
         [ 462,  70],  # top left
         [ 562,  70]]; # top right

    # colors for animated Christmas tree lights
    my integer::arrayref::arrayref $my_lights_colors = [$my_color_pink, $my_color_purple, $my_color_orange];

    # [ PREPARE & MAKE INITIAL RECURSIVE CALL ]

    # declare & initialize final array outside of the recursive subroutine for easy direct access by all recursive calls
    my number::arrayref::arrayref::arrayref::arrayref $my_triangle_groups = [];

    # initial triangle is in a triangle group by itself
    $my_triangle_groups->[$my_recursions_remaining] = [$my_triangle_initial];

    # initial call to recursive subroutine
    MathPerl::Fractal::Sierpinski::sierpinski($my_triangle_initial, $my_recursions_remaining, $my_triangle_groups);

    # regex to (g)lobally (s)earch for numbers incorrectly wrapped in 'single-quotes' by Dumper,
    # replace by // empty string, no lvalue $variable so directly (r)eturn modified string;
    # https://perldoc.perl.org/perlop#s%2FPATTERN%2FREPLACEMENT%2Fmsixpodualngcer
    print 'have $my_triangle_groups = ', "\n", (Dumper($my_triangle_groups) =~ s/'//gr);

    # [ INITIALIZE GRAPHICS ]

    # https://metacpan.org/dist/SDL/view/lib/pods/SDL/Events.pod
    my @SDL_EVENTS = qw(
        no_such_event
        SDL_ACTIVEEVENT
        SDL_KEYDOWN SDL_KEYUP
        SDL_MOUSEMOTION SDL_MOUSEBUTTONDOWN SDL_MOUSEBUTTONUP
        SDL_JOYAXISMOTION SDL_JOYBALLMOTION SDL_JOYHATMOTION SDL_JOYBUTTONDOWN SDL_JOYBUTTONUP
        SDL_QUIT
        SDL_SYSWMEVENT
        SDL_VIDEORESIZE SDL_VIDEOEXPOSE
        SDL_USEREVENT
        SDL_NUMEVENTS
    );  # constant data

    # SDL includes moved into [[[ INCLUDES ]]] section above

    # initialize SDL video & application & event;
    # we do not call $my_SDL_app->run() anywhere in this program, instead we use the while() run loop below
    SDL::init(SDL_INIT_VIDEO);
    my SDLx::App $my_SDL_app = SDLx::App->new(
        title  => 'Merry Christmas!!  Perl Advent 2022!!!',
        width  => 1024,                    # hard-coded 1024x768 resolution
        height => 768,
        depth  => 32,                      # hard-coded 32-bit color
        resizeable => 1                    # allow window resize; does not scale window contents
    );
    my $my_SDL_event = SDL::Event->new;

    # [ RENDER STATIC GRAPHICS ]

    # draw Christmas tree branches & snow tinsel & ornaments & lights;
    # iterate through triangle groups in reverse order, due to reverse population during recursion
    for (my $i = ((scalar @{$my_triangle_groups}) - 1); $i >= 0; $i--) {
        my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[$i];
        my integer::arrayref $my_color = $my_triangle_colors->[$i];

        for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
            my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];

            # https://metacpan.org/dist/SDL/view/lib/pods/SDLx/Surface.pod
            $my_SDL_app->draw_trigon_filled( $my_triangle, $my_color );

            # refresh window on every triangle for fun cascade drawing effect
            $my_SDL_app->update();
        }
    }

    # draw Christmas tree trunk & Star of Bethlehem
    $my_SDL_app->draw_rect( $my_rectangle_trunk, $my_color_brown );
    $my_SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
    $my_SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );
    $my_SDL_app->update();  # refresh window

    # [ RENDER DYNAMIC (ANIMATED) GRAPHICS ]

    # set initial index for accesssing Christmas tree lights colors
    my integer $my_lights_colors_index = 0;

    # set initial time for changing Christmas tree lights colors
    (my integer $seconds_start) = gettimeofday();
    #print 'have $seconds_start =  ', $seconds_start, "\n";

    # the main run loop, used instead of calling $my_SDL_app->run();
    # animate forever, until SDL_QUIT event is received in GUI window via <Alt-F4> keypress or window close mouse click,
    # or in CLI window via <Ctrl-C> keypress
    while(1)
    {
        # pump the event loop, gathering events from input devices
        SDL::Events::pump_events();

        # poll for currently pending events
        if(SDL::Events::poll_event($my_SDL_event))
        {
            print 'have @SDL_EVENTS[', $my_SDL_event->type(), '] = ', @SDL_EVENTS[$my_SDL_event->type()], "\n";

            # we only care about the SDL_QUIT event telling us to exit
            if ($my_SDL_event->type == SDL_QUIT) {
                print 'SDL_QUIT event received, exiting', "\n";
                exit;
            }
        }

        # get current time, for comparison with start time of current Christmas tree lights color
        (my integer $seconds_current) = gettimeofday();
    #    print 'have $seconds_current = ', $seconds_current, "\n";

        # twinkle Christmas tree lights every 1 second
        if (($seconds_current - $seconds_start) >= 1) {
            # reset start time to current time, for time cycle of next animation frame
            $seconds_start = $seconds_current;

            # iterate through lights colors
            my integer::arrayref $my_color = $my_lights_colors->[$my_lights_colors_index];
            print 'have $my_color = $my_lights_colors->[', $my_lights_colors_index, '] = ', Dumper($my_color);
```

```
            # wrap back to beginning of lights colors when end is reached
            $my_lights_colors_index++;
            if ($my_lights_colors_index > ((scalar @{$my_lights_colors}) - 1)) {
                $my_lights_colors_index = 0;
            }

            # only update second-smallest triangles, not the green of the Christmas tree itself
            my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[1];
            for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
                my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
                $my_SDL_app->draw_trigon_filled( $my_triangle, $my_color );
            }

            # redraw green of Christmas tree
            $my_triangle_group = $my_triangle_groups->[0];
            for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
                my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
                $my_SDL_app->draw_trigon_filled( $my_triangle, $my_triangle_colors->[0] );
            }

            # redraw Star of Bethlehem
            $my_SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
            $my_SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );

            # refresh window once for every Christmas tree lights color change, for synchronized lights
            $my_SDL_app->update();
        }

        # briefly pause between each while() loop iteration, to avoid overloading CPU;
        # ( 1_000_000 microseconds per second ) / ( 10_000 microseconds per iteration) = 100 iterations per second;
        # need at least 100 while loop iterations per second, in order to process all of the otherwise-ignored
        # SDL_MOUSEMOTION events which are caused by simply moving the mouse over top of the window
        usleep(10_000);
    }

    return;
}  # end of generate_fractal__render_animation()

1;  # end of class
```

The above Christmas tree code is pretty much the simplest 2-D graphics rendering system I could write using SDL, with the ability to be exited gracefully instead of having to type `Ctrl-Z` and then `$ killall -KILL perl`.

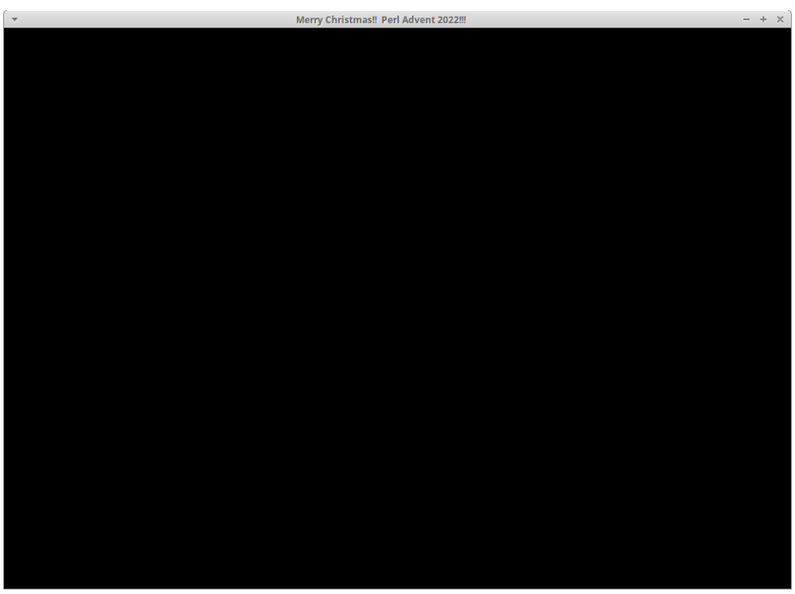
Last, we only need a few lines of driver code to run it all:

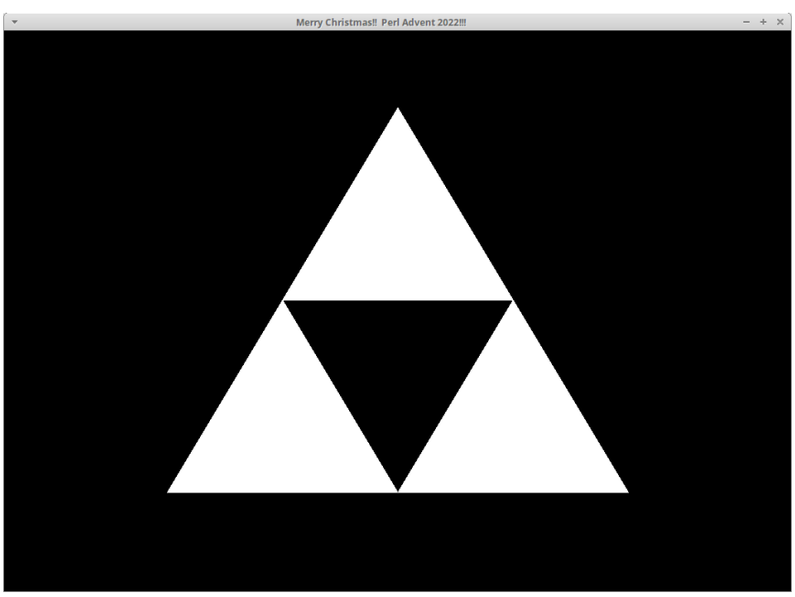
sierpinski_triangles_christmas.pl

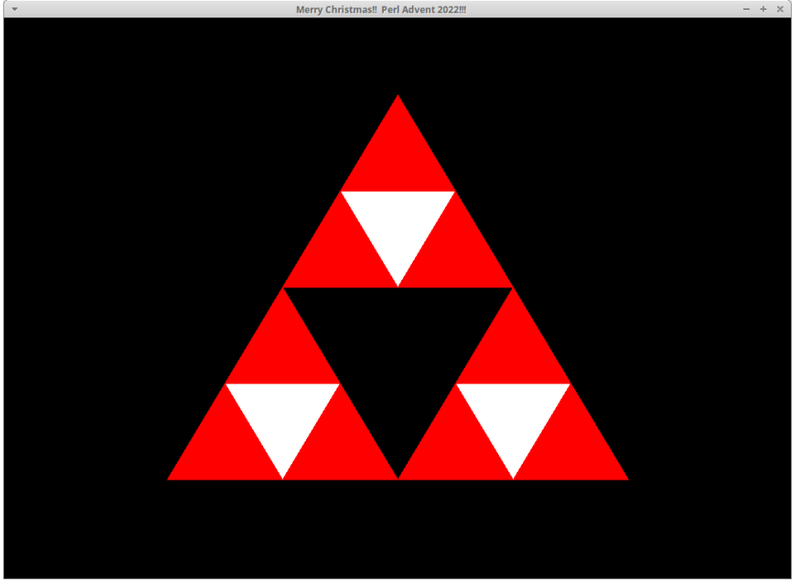
```
#!/usr/bin/env perl

# Fractal Christmas Tree
# Sierpinski triangles animated using SDL graphics

# [[[ HEADER ]]]
use strict;
use warnings;
our $VERSION = 0.008_000;

# [[[ INCLUDES ]]]
use MathPerl::Fractal::ChristmasTree;

# [[[ OPERATIONS ]]]
MathPerl::Fractal::ChristmasTree::generate_fractal__render_animation();
```

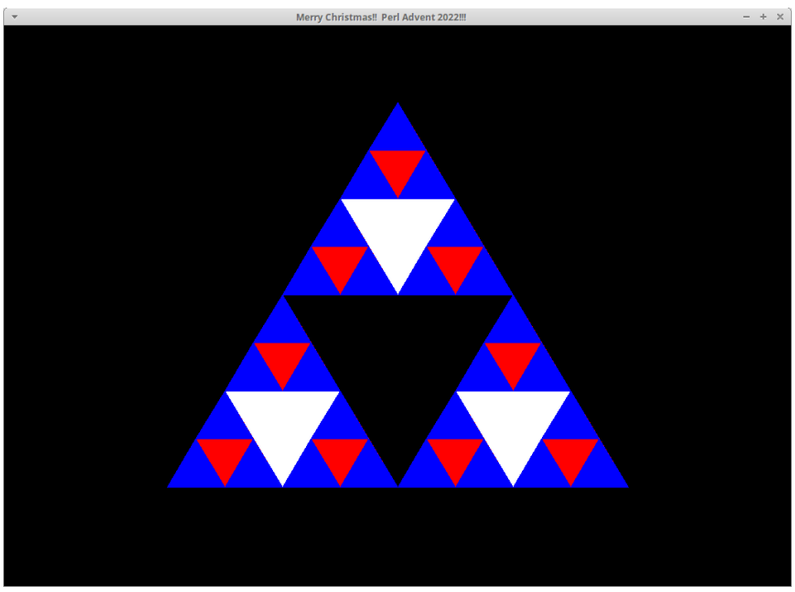
If you were to run the above program, which includes 5 colors for 5 levels of recursion, then you would see a rendered series of images similar to the following:

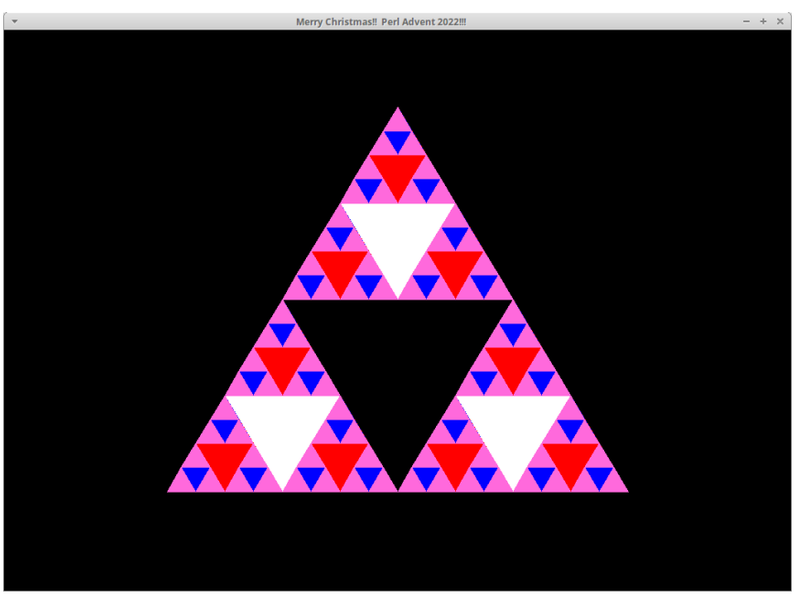
```
$ ./sierpinski_triangles_christmas.pl
```

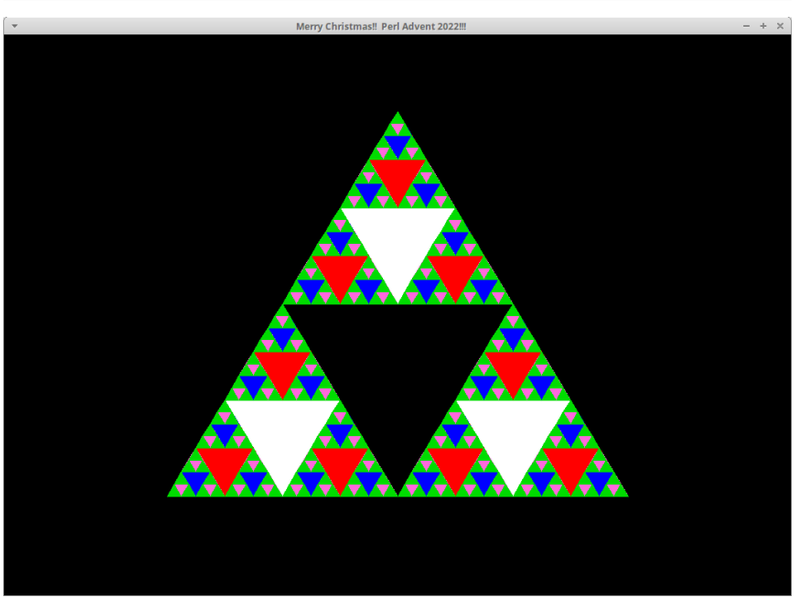


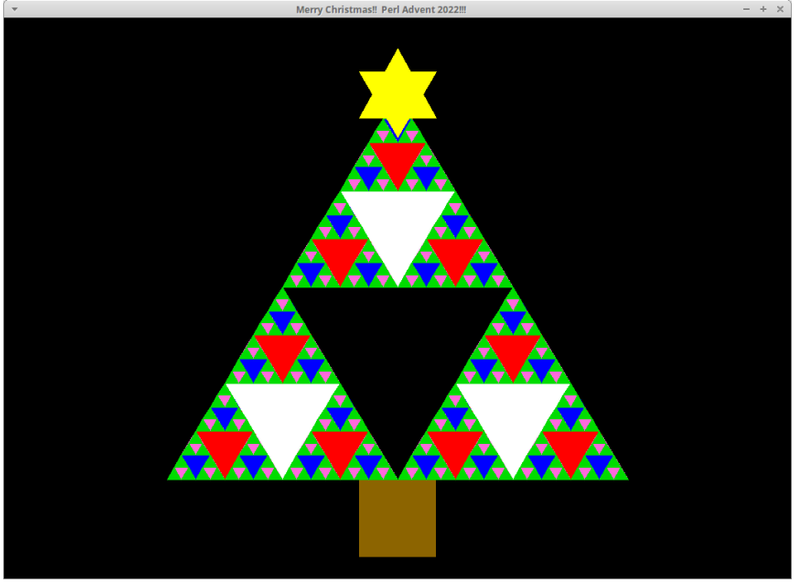


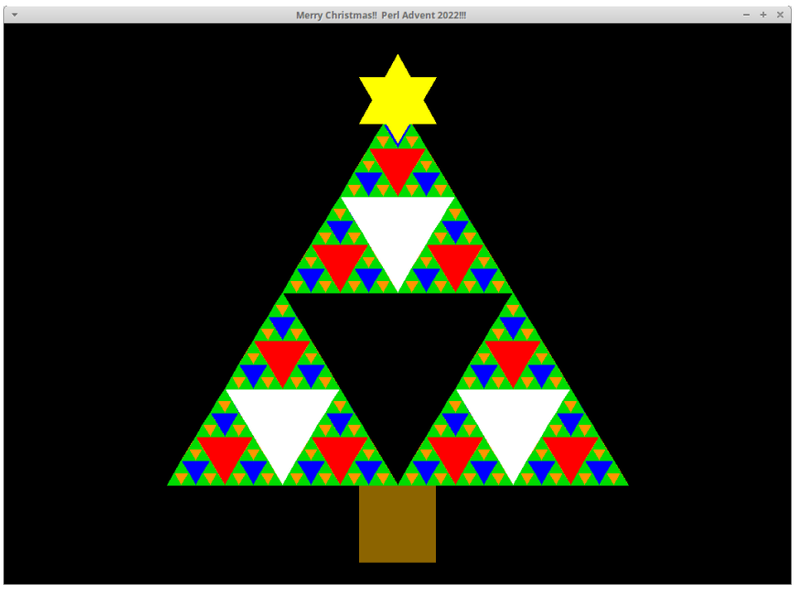

Merry Christmas!! Perl Advent 2022!!!

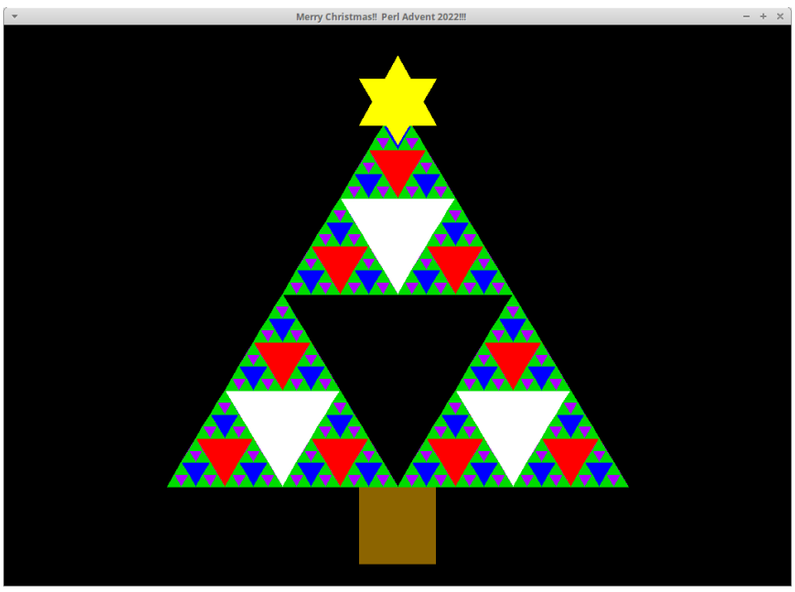

Merry Christmas!! Perl Advent 2022!!!


Merry Christmas!! Perl Advent 2022!!!


Merry Christmas!! Perl Advent 2022!!!







However, before you can run this program you will need to install the SDL dependencies:

```
$ cpanm -v SDL SDLx::App SDL::Event SDL::Events
```

Also, the easiest way to run this Fractal Christmas Tree program is to copy or download the monolithic code below (instead of all 3 files above), and then paste it into a single executable Perl file:

sierpinski_triangles_christmas_monolith.pl

```
#!/usr/bin/env perl

# Fractal Christmas Tree, Monolithic Single File
# Sierpinski triangles animated using SDL graphics

# [[[ PREPROCESSOR ]]]
# declare Perl-compatible data types & data structures
package void; 1;
package integer; 1;
package integer::arrayref; 1;
package integer::arrayref::arrayref; 1;
package number::arrayref; 1;
package number::arrayref::arrayref; 1;
package number::arrayref::arrayref::arrayref; 1;
package number::arrayref::arrayref::arrayref::arrayref; 1;

# [[[ HEADER ]]]
package main;
use strict;
use warnings;
use v5.14;  # required for /r return AKA non-destructive regex flag
our $VERSION = 0.008_000;

# [[[ CRITICS ]]]
# USER DEFAULT 1: allow numeric values & print operator
## no critic qw(ProhibitUselessNoCritic ProhibitMagicNumbers RequireCheckedSyscalls)
## no critic qw(RequireInterpolationOfMetachars)  # USER DEFAULT 2: allow single-quoted control characters & sigils
## no critic qw(ProhibitConstantPragma ProhibitMagicNumbers)  # USER DEFAULT 3: allow constants

# [[[ INCLUDES ]]]
use English;
use Data::Dumper;
$Data::Dumper::Deepcopy = 1;  # display human-readable numeric data, not $VAR1->[0] references

# https://metacpan.org/dist/SDL/view/lib/pods/SDL/Event.pod
use SDL;
use SDLx::App;   # used for window creation & control
use SDL::Event;  # used for creating Event object
use SDL::Events; # used for Event queue handling functions
```

```
use Time::HiRes qw( gettimeofday usleep );  # used for time-based animation control

# [[[ CONSTANTS ]]]

# [ DATA FOR SIZES & SHAPES & COLORS;
#   HARD-CODED 1024x768 RESOLUTION & 32-BIT COLOR DEPTH ]

# initial triangle's 3 corners as [x, y] Euclidean coordinates
my number::arrayref::arrayref $my_triangle_initial =
    [[ 512, 100],  # top point
     [ 212, 600],  # bottom left
     [ 812, 600]]; # bottom right

my integer::arrayref $my_color_red     = [255, 000, 000, 255];
my integer::arrayref $my_color_pink    = [255, 105, 220, 255];
my integer::arrayref $my_color_orange  = [255, 150, 000, 255];
my integer::arrayref $my_color_yellow  = [255, 255, 000, 255];
my integer::arrayref $my_color_green   = [000, 220, 000, 255];
my integer::arrayref $my_color_blue    = [000, 000, 255, 255];
my integer::arrayref $my_color_purple  = [175, 000, 255, 255];
my integer::arrayref $my_color_white   = [255, 255, 255, 255];
my integer::arrayref $my_color_brown   = [140, 100, 000, 255];

# colors as [r, g, b] triplets; number of colors is number of recursions
my integer::arrayref::arrayref $my_triangle_colors =
    [ $my_color_green,  # green needs to be the color of the smallest, and thus most numerous, triangles
      $my_color_pink,
      $my_color_blue,
      $my_color_red,
      $my_color_white ];
my integer $my_recursions_remaining = scalar(@{$my_triangle_colors});

# rectangle in format [ x, y, width, height ]
my number::arrayref::arrayref $my_rectangle_trunk =
    [462, 601, 100, 100];

my number::arrayref::arrayref $my_triangle_star_up =
    [[ 512, 050],  # top point
     [ 462, 130],  # bottom left
     [ 562, 130]]; # bottom right
my number::arrayref::arrayref $my_triangle_star_down =
    [[ 512, 155],  # bottom point
     [ 462,  70],  # top left
     [ 562,  70]]; # top right

# colors for animated Christmas tree lights
my integer::arrayref::arrayref $my_lights_colors = [$my_color_pink, $my_color_purple, $my_color_orange];

# https://metacpan.org/dist/SDL/view/lib/pods/SDL/Events.pod
my @SDL_EVENTS = qw(
    no_such_event
    SDL_ACTIVEEVENT
    SDL_KEYDOWN SDL_KEYUP
    SDL_MOUSEMOTION SDL_MOUSEBUTTONDOWN SDL_MOUSEBUTTONUP
    SDL_JOYAXISMOTION SDL_JOYBALLMOTION SDL_JOYHATMOTION SDL_JOYBUTTONDOWN SDL_JOYBUTTONUP
    SDL_QUIT
    SDL_SYSWMEVENT
    SDL_VIDEORESIZE SDL_VIDEOEXPOSE
    SDL_USEREVENT
    SDL_NUMEVENTS
);  # constant data

# [[[ OPERATIONS ]]]

# [ PREPARE & MAKE INITIAL RECURSIVE CALL ]

# declare & initialize final array outside of the recursive subroutine for easy direct access by all recursive calls
my number::arrayref::arrayref::arrayref::arrayref $my_triangle_groups = [];

# initial triangle is in a triangle group by itself
$my_triangle_groups->[$my_recursions_remaining] = [$my_triangle_initial];

# initial call to recursive subroutine
sierpinski($my_triangle_initial, $my_recursions_remaining, $my_triangle_groups);

# regex to (g)lobally (s)earch for numbers incorrectly wrapped in 'single-quotes' by Dumper,
# replace by // empty string, no lvalue $variable so directly (r)eturn modified string;
# https://perldoc.perl.org/perlop#s%2FPATTERN%2FREPLACEMENT%2Fmsixpodualngcer
print 'have $my_triangle_groups = ', "\n", (Dumper($my_triangle_groups) =~ s/'//gr);

# [ INITIALIZE GRAPHICS ]

# SDL includes moved into [[[ INCLUDES ]]] section above

# initialize SDL video & application & event;
# we do not call $my_SDL_app->run() anywhere in this program, instead we use the while() run loop below
SDL::init(SDL_INIT_VIDEO);
my SDLx::App $my_SDL_app = SDLx::App->new(
    title  => 'Merry Christmas!!  Perl Advent 2022!!!',
    width  => 1024,                    # hard-coded 1024x768 resolution
    height => 768,
    depth  => 32,                      # hard-coded 32-bit color
    resizeable => 1                    # allow window resize; does not scale window contents
);
my $my_SDL_event = SDL::Event->new;

# [ RENDER STATIC GRAPHICS ]

# draw Christmas tree branches & snow tinsel & ornaments & lights;
# iterate through triangle groups in reverse order, due to reverse population during recursion
for (my $i = ((scalar @{$my_triangle_groups}) - 1); $i >= 0; $i--) {
    my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[$i];
    my integer::arrayref $my_color = $my_triangle_colors->[$i];

    for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
        my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];

        # https://metacpan.org/dist/SDL/view/lib/pods/SDLx/Surface.pod
        $my_SDL_app->draw_trigon_filled( $my_triangle, $my_color );

        # refresh window on every triangle for fun cascade drawing effect
        $my_SDL_app->update();
    }
}

# draw Christmas tree trunk & Star of Bethlehem
$my_SDL_app->draw_rect( $my_rectangle_trunk, $my_color_brown );
$my_SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
$my_SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );
$my_SDL_app->update();  # refresh window

# [ RENDER DYNAMIC (ANIMATED) GRAPHICS ]

# set initial index for accesssing Christmas tree lights colors
my integer $my_lights_colors_index = 0;

# set initial time for changing Christmas tree lights colors
(my integer $seconds_start) = gettimeofday();
#print 'have $seconds_start = ', $seconds_start, "\n";

# the main run loop, used instead of calling $my_SDL_app->run();
# animate forever, until SDL_QUIT event is received in GUI window via <Alt-F4> keypress or window close mouse click,
# or in CLI window via <Ctrl-C> keypress
while(1)
{
    # pump the event loop, gathering events from input devices
    SDL::Events::pump_events();

    # poll for currently pending events
    if(SDL::Events::poll_event($my_SDL_event))
    {
        print 'have @SDL_EVENTS[', $my_SDL_event->type(), '] = ', @SDL_EVENTS[$my_SDL_event->type()], "\n";

        # we only care about the SDL_QUIT event telling us to exit
        if ($my_SDL_event->type == SDL_QUIT) {
            print 'SDL_QUIT event received, exiting', "\n";
            exit;
        }
    }

    # get current time, for comparison with start time of current Christmas tree lights color
    (my integer $seconds_current) = gettimeofday();
#    print 'have $seconds_current = ', $seconds_current, "\n";

    # twinkle Christmas tree lights every 1 second
    if (($seconds_current - $seconds_start) >= 1) {
        # reset start time to current time, for time cycle of next animation frame
        $seconds_start = $seconds_current;

        # iterate through lights colors
        my integer::arrayref $my_color = $my_lights_colors->[$my_lights_colors_index];
        print 'have $my_color = $my_lights_colors->[', $my_lights_colors_index, '] = ', Dumper($my_color);

        # wrap back to beginning of lights colors when end is reached
        $my_lights_colors_index++;
        if ($my_lights_colors_index > ((scalar @{$my_lights_colors}) - 1)) {
            $my_lights_colors_index = 0;
        }

        # only update second-smallest triangles, not the green of the Christmas tree itself
        my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[1];
        for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
            my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
            $my_SDL_app->draw_trigon_filled( $my_triangle, $my_color );
        }

        # redraw green of Christmas tree
        $my_triangle_group = $my_triangle_groups->[0];
        for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
            my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
            $my_SDL_app->draw_trigon_filled( $my_triangle, $my_triangle_colors->[0] );
        }

        # redraw Star of Bethlehem
        $my_SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
        $my_SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );

        # refresh window once for every Christmas tree lights color change, for synchronized lights
        $my_SDL_app->update();
    }

    # briefly pause between each while() loop iteration, to avoid overloading CPU;
```

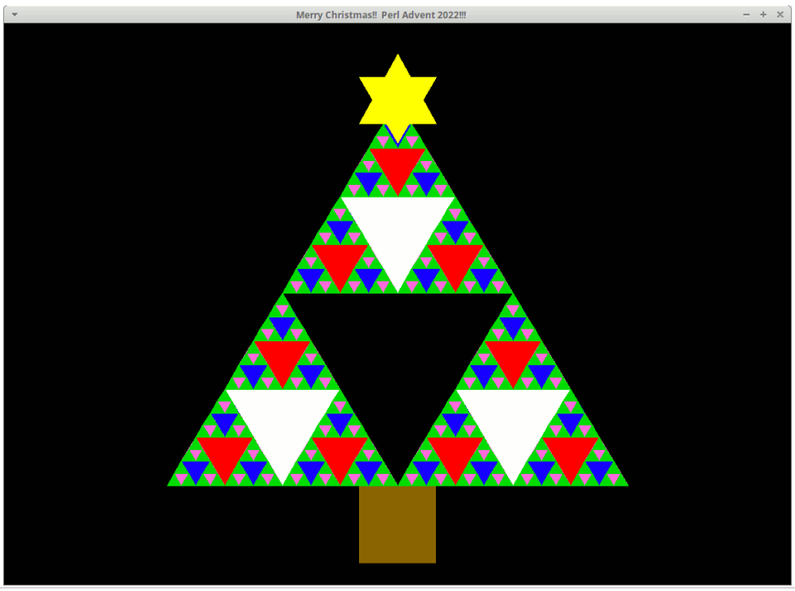
```
    # ( 1_000_000 microseconds per second ) / ( 10_000 microseconds per iteration) = 100 iterations per second;
    # need at least 100 while loop iterations per second, in order to process all of the otherwise-ignored
    # SDL_MOUSEMOTION events which are caused by simply moving the mouse over top of the window
    usleep(10_000);
}

# [[[ SUBROUTINES ]]]

# recursively generate triangles, grouped by recursion level
sub sierpinski {
    { my void $RETURN_TYPE };
    (
        my number::arrayref::arrayref $triangle,
        my integer $recursions_remaining,
        my number::arrayref::arrayref::arrayref::arrayref $triangle_groups
    ) = @ARG;

    print 'in sierpinski(), received $recursions_remaining = ', $recursions_remaining, "\n";
    print 'in sierpinski(), received $triangle = ', (Dumper($triangle) =~ s/'//gr), "\n";

    if ($recursions_remaining > 0) {
        # shortcut variables, easier to read in midpoint calculations below
        my number::arrayref $point_a = $triangle->[0];
        my number::arrayref $point_b = $triangle->[1];
        my number::arrayref $point_c = $triangle->[2];

        # calculate midpoints between two coordinates [x1, y1] and [x2, y2] is [(x1+x2)/2, (y1+y2)/2]
        my number::arrayref $point_a_b =
            [(($point_a->[0] + $point_b->[0]) / 2),
             (($point_a->[1] + $point_b->[1]) / 2)];
        my number::arrayref $point_a_c =
            [(($point_a->[0] + $point_c->[0]) / 2),
             (($point_a->[1] + $point_c->[1]) / 2)];
        my number::arrayref $point_b_c =
            [(($point_b->[0] + $point_c->[0]) / 2),
             (($point_b->[1] + $point_c->[1]) / 2)];

        # construct 3 sub-triangles from orinal points and newly-calculated midpoints
        my number::arrayref::arrayref $subtriangle_a = [ $point_a,   $point_a_b, $point_a_c ];
        my number::arrayref::arrayref $subtriangle_b = [ $point_a_b, $point_b,   $point_b_c ];
        my number::arrayref::arrayref $subtriangle_c = [ $point_a_c, $point_b_c, $point_c   ];

        # $triangle_groups is zero-indexed like all other Perl arrays,
        # so we need to subtract one from $recursions_remaining before using as an index,
        # in order to avoid an undefined element at element 0;
        # also, we need to decrement $recursions_remaining before making recursive calls;
        # for both of these reasons, we can decrement now
        $recursions_remaining--;

        # store all triangles grouped by recursion level
        push @{$triangle_groups->[$recursions_remaining]}, $subtriangle_a;
        push @{$triangle_groups->[$recursions_remaining]}, $subtriangle_b;
        push @{$triangle_groups->[$recursions_remaining]}, $subtriangle_c;

        # recurse once for each sub-triangle
        sierpinski( $subtriangle_a, $recursions_remaining, $triangle_groups);
        sierpinski( $subtriangle_b, $recursions_remaining, $triangle_groups);
        sierpinski( $subtriangle_c, $recursions_remaining, $triangle_groups);
    }

    # return after maximum recursion level is reached (conditional block above not entered),
    # or all recursion calls have returned (conditional block above entered);
    # no return value, all generated data is stored directly in $triangle_groups
    return;
}

1;  # end of package 'main'
```

If you review the graphics rendering code above, you will see the `while(1)` main run loop which twinkles the Christmas tree lights, displaying an animated color change once every second.

Run it yourself and bask in the Perl yuletide glory of your very own Sierpinski triangle fractal Christmas tree!

```
$ ./sierpinski_triangles_christmas_monolith.pl
```

Merry Christmas to all, and to all a good night! :-)



This article contributed by: Will 'the Chill' Braswell <william.braswell@autoparallel.com>