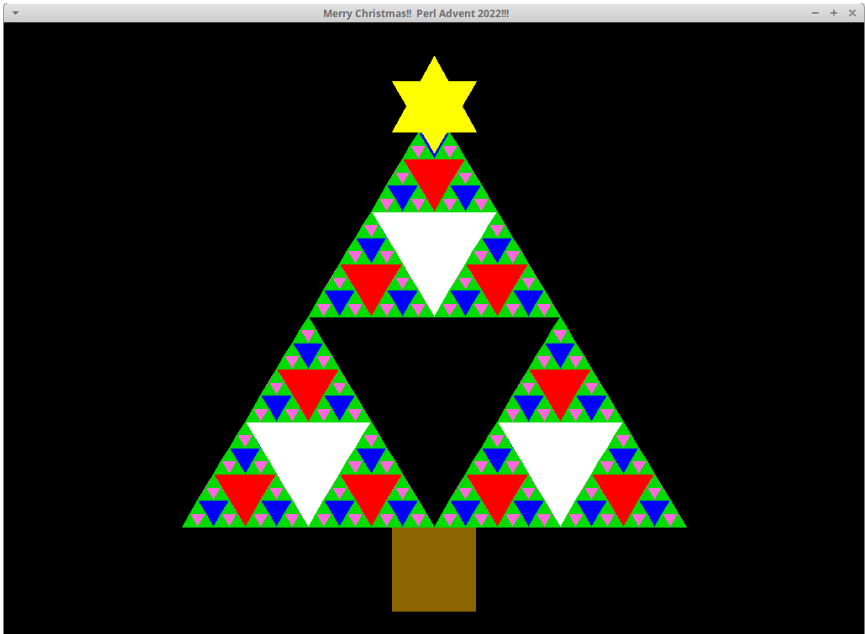


Fractal Christmas Tree

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`.

```
1:  # [[[ PREPROCESSOR ]]]
2:  # declare Perl-compatible data types & data structures
3:  package void; 1;
4:  package integer; 1;
5:  package integer::arrayref; 1;
6:  package integer::arrayref::arrayref; 1;
7:  package number::arrayref; 1;
8:  package number::arrayref::arrayref; 1;
9:  package number::arrayref::arrayref::arrayref; 1;
10: package number::arrayref::arrayref::arrayref::arrayref; 1;
11:
12: # [[[ HEADER ]]]
13: #use RPerl; # replaced by PREPROCESSOR for simplicity
14: package MathPerl::Fractal::Sierpinski;
15: use strict;
16: use warnings;
17: use v5.14; # required for /r return AKA non-destructive regex flag
18: our $VERSION = 0.008_000;
19:
20: # [[[ CRITICS ]]]
21: # USER DEFAULT 1: allow numeric values & print operator
22: ## no critic qw(ProhibitUselessNoCritic ProhibitMagicNumbers RequireCheckedSyscalls)
23: ## no critic qw(RequireInterpolationOfMetachars) # USER DEFAULT 2: allow single-quoted control characters & sigils
24: ## no critic qw(ProhibitConstantPragma ProhibitMagicNumbers) # USER DEFAULT 3: allow constants
25:
26: # [[[ OO INHERITANCE ]]]
27: #use parent qw(MathPerl::Fractal); # disable unnecessary inheritance for simplicity
28: #use MathPerl::Fractal;
29:
30: # [[[ INCLUDES ]]]
31: use English;
32: use Data::Dumper;
33: $Data::Dumper::Deepcopy = 1; # display human-readable numeric data, not $VAR1->{0} references
34:
35: # [[[ SUBROUTINES ]]]
36:
37: # recursively generate triangles, grouped by recursion level
38: sub sierpinski {
39:     { my void $RETURN_TYPE };
40:     (
41:         my number::arrayref::arrayref $triangle,
42:         my integer $recursions_remaining,
43:         my number::arrayref::arrayref::arrayref::arrayref $triangle_groups
44:     ) = @ARG;
45:
46:     print "in sierpinski(), received $recursions_remaining = ", $recursions_remaining, "\n";
47:     print "in sierpinski(), received $triangle = ", (Dumper($triangle) =~ s//gr), "\n";
48:
49:     if ($recursions_remaining > 0) {
50:         # shortcut variables, easier to read in midpoint calculations below
51:         my number::arrayref $point_a = $triangle->[0];
52:         my number::arrayref $point_b = $triangle->[1];
53:         my number::arrayref $point_c = $triangle->[2];
54:
55:         # calculate midpoints between two coordinates [x1, y1] and [x2, y2] is [(x1+x2)/2, (y1+y2)/2]
56:         my number::arrayref $point_a_b =
57:             [
58:                 (((point_a->[0] + point_b->[0]) / 2),
59:                 (((point_a->[1] + point_b->[1]) / 2)]);
60:         my number::arrayref $point_a_c =
61:             [
62:                 (((point_a->[0] + point_c->[0]) / 2),
63:                 (((point_a->[1] + point_c->[1]) / 2)]);
64:         my number::arrayref $point_b_c =
65:             [
66:                 (((point_b->[0] + point_c->[0]) / 2),
67:                 (((point_b->[1] + point_c->[1]) / 2)]);
68:
69:         # construct 3 sub-triangles from orinal points and newly-calculated midpoints
70:         my number::arrayref::arrayref $subtriangle_a = [ $point_a, $point_a_b, $point_a_c ];
71:         my number::arrayref::arrayref $subtriangle_b = [ $point_a_b, $point_b, $point_b_c ];
72:         my number::arrayref::arrayref $subtriangle_c = [ $point_a_c, $point_b_c, $point_c ];
73:
74:         # $triangle_groups is zero-indexed like all other Perl arrays,
75:         # so we need to subtract one from $recursions_remaining before using as an index,
76:         # in order to avoid an undefined element at element 0;
77:         # also, we need to decrement $recursions_remaining before making recursive calls;
78:         # for both of these reasons, we can decrement now
79:         $recursions_remaining--;
80:
81:         # store all triangles grouped by recursion level
82:         push @$triangle_groups->[$recursions_remaining], $subtriangle_a;
83:         push @$triangle_groups->[$recursions_remaining], $subtriangle_b;
84:         push @$triangle_groups->[$recursions_remaining], $subtriangle_c;
85:
86:         # recurse once for each sub-triangle
87:         sierpinski( $subtriangle_a, $recursions_remaining, $triangle_groups);
88:         sierpinski( $subtriangle_b, $recursions_remaining, $triangle_groups);
89:         sierpinski( $subtriangle_c, $recursions_remaining, $triangle_groups);
90:     }
91:
92:     # return after maximum recursion level is reached (conditional block above not entered),
93:     # or all recursion calls have returned (conditional block above entered);
94:     # no return value, all generated data is stored directly in $triangle_groups
95:     return;
96: }
```

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 undeclared.

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

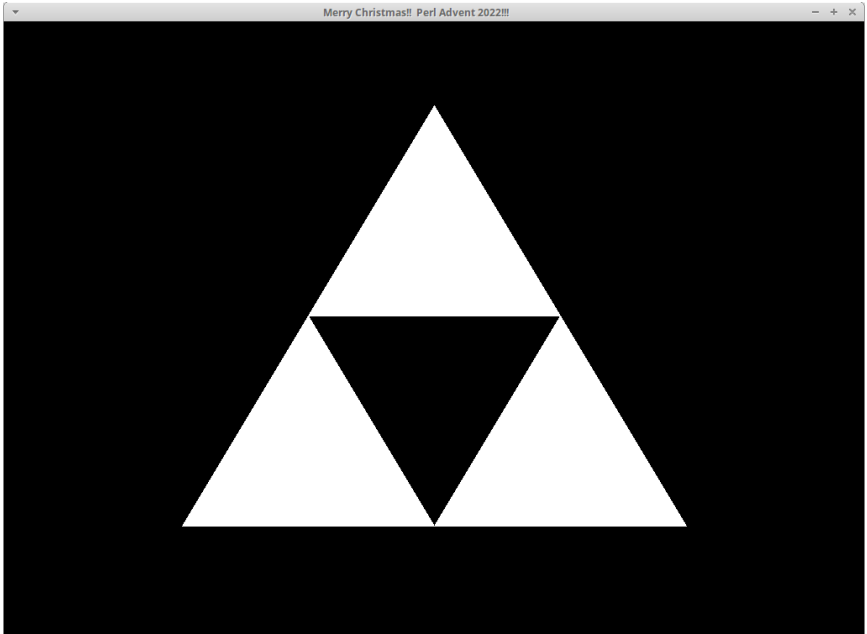
```
1:  in sierpinski(), received $recursions_remaining = 1
2:  in sierpinski(), received $triangle = $VAR1 = [
3:      [
4:          512,
5:          100,
```

```
6:         ],
7:         [
8:             212,
9:             600
10:        ],
11:        [
12:            812,
13:            600
14:        ]
15:    ];
```

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

```
1:  in sierpinski(), received $recursions_remaining = 0
2:  in sierpinski(), received $triangle = $VAR1 = [
3:      [
4:          512,
5:          100
6:      ],
7:      [
8:          362,
9:          350
10:     ],
11:     [
12:         662,
13:         350
14:     ]
15: ];
16:
17:  in sierpinski(), received $recursions_remaining = 0
18:  in sierpinski(), received $triangle = $VAR1 = [
19:      [
20:          362,
21:          350
22:      ],
23:      [
24:          212,
25:          600
26:      ],
27:      [
28:          512,
29:          600
30:      ]
31: ];
32:
33:  in sierpinski(), received $recursions_remaining = 0
34:  in sierpinski(), received $triangle = $VAR1 = [
35:      [
36:          662,
37:          350
38:      ],
39:      [
40:          512,
41:          600
42:      ],
43:      [
44:          812,
45:          600
46:      ]
47: ];
```

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:

```
1:  have $my_triangle_groups =
2:  $VAR1 = [
3:      [
4:          [
5:              [
6:                  512,
7:                  100
8:              ],
9:              [
10:                 362,
11:                 350
12:             ],
13:             [
14:                 662,
15:                 350
16:             ]
17:         ],
18:         [
19:             [
20:                 362,
21:                 350
22:             ],
23:             [
24:                 212,
25:                 600
26:             ],
27:             [
28:                 512,
29:                 600
30:             ]
31:         ],
32:         [
33:             [
34:                 662,
35:                 350
36:             ],
37:             [
38:                 512,
39:                 600
40:             ],
41:             [
42:                 812,
43:                 600
44:             ]
45:         ]
46:     ],
47:     [
48:         [
49:             [
50:                 512,
51:                 100
52:             ],
53:             [
54:                 212,
55:                 600
56:             ],
57:             [
58:                 812,
59:                 600
60:             ]
61:         ]
62:     ]
63: ];
```

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`:

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

```

3: package MathPerl::Fractal::ChristmasTree;
4: use strict;
5: use warnings;
6: use v5.14; # required for /r return AKA non-destructive regex flag
7: our $VERSION = 0.008_000;
8:
9: # [[[ CRITICS ]]]
10: # USER DEFAULT 1: allow numeric values & print operator
11: ## no critic qw(ProhibitUselessNoCritic ProhibitMagicNumbers RequireCheckedSyscalls)
12: ## no critic qw(RequireInterpolationOfMetachars) # USER DEFAULT 2: allow single-quoted control characters & sigils
13: ## no critic qw(ProhibitConstantPragma ProhibitMagicNumbers) # USER DEFAULT 3: allow constants
14:
15: # [[[ 00 INHERITANCE ]]]
16: use parent qw(MathPerl::Fractal::Sierpinski);
17: use MathPerl::Fractal::Sierpinski;
18:
19: # [[[ INCLUDES ]]]
20: use English;
21: use Data::Dumper;
22: $Data::Dumper::Deepcopy = 1; # display human-readable numeric data, not $VAR1->[0] references
23:
24: # https://metacpan.org/dist/SDL/view/lib/pods/SDL/Event.pod
25: use SDL;
26: use SDLx::App; # used for window creation & control
27: use SDL::Event; # used for creating Event object
28: use SDL::Events; # used for Event queue handling functions
29: use Time::HiRes qw( gettimeofday usleep ); # used for time-based animation control
30:
31: # [[[ SUBROUTINES ]]]
32:
33: # display an animated Christmas tree!
34: # define hard-coded constant data; call sierpinski() to recursively generate fractal triangles;
35: # initialize SDL graphics; render static graphics; render dynamic graphics (animation)
36: sub generate_fractal_render_animation {
37:     { my void $RETURN_TYPE };
38:
39:     # [ DATA FOR SIZES & SHAPES & COLORS;
40:     #   HARD-CODED 1024x768 RESOLUTION & 32-BIT COLOR DEPTH ]
41:
42:     # initial triangle's 3 corners as [x, y] Euclidean coordinates
43:     my number::arrayref::arrayref $my_triangle_initial =
44:         [[ 512, 100], # top point
45:          [ 212, 600], # bottom left
46:          [ 812, 600]]; # bottom right
47:
48:     my integer::arrayref $my_color_red      = [255, 000, 000, 255];
49:     my integer::arrayref $my_color_pink     = [255, 105, 220, 255];
50:     my integer::arrayref $my_color_orange  = [255, 150, 000, 255];
51:     my integer::arrayref $my_color_yellow  = [255, 255, 000, 255];
52:     my integer::arrayref $my_color_green   = [000, 220, 000, 255];
53:     my integer::arrayref $my_color_blue    = [000, 000, 255, 255];
54:     my integer::arrayref $my_color_purple  = [175, 000, 255, 255];
55:     my integer::arrayref $my_color_white   = [255, 255, 255, 255];
56:     my integer::arrayref $my_color_brown   = [140, 100, 000, 255];
57:
58:     # colors as [r, g, b] triplets; number of colors is number of recursions
59:     my integer::arrayref::arrayref $my_triangle_colors =
60:         [ $my_color_green, # green needs to be the color of the smallest, and thus most numerous, triangles
61:           $my_color_pink,
62:           $my_color_blue,
63:           $my_color_red,
64:           $my_color_white ];
65:     my integer $my_recursions_remaining = scalar(@{$my_triangle_colors});
66:
67:     # rectangle in format [ x, y, width, height ]
68:     my number::arrayref::arrayref $my_rectangle_trunk =
69:         [462, 601, 100, 100];
70:
71:     my number::arrayref::arrayref $my_triangle_star_up =
72:         [[ 512, 050], # top point
73:          [ 462, 130], # bottom left
74:          [ 562, 130]]; # bottom right
75:     my number::arrayref::arrayref $my_triangle_star_down =
76:         [[ 512, 155], # bottom point
77:          [ 462, 70], # top left
78:          [ 562, 70]]; # top right
79:
80:     # colors for animated Christmas tree lights
81:     my integer::arrayref::arrayref $my_lights_colors = [$my_color_pink, $my_color_purple, $my_color_orange];
82:
83:     # [ PREPARE & MAKE INITIAL RECURSIVE CALL ]
84:
85:     # declare & initialize final array outside of the recursive subroutine for easy direct access by all recursive calls
86:     my number::arrayref::arrayref::arrayref $my_triangle_groups = [];
87:
88:     # initial triangle is in a triangle group by itself
89:     $my_triangle_groups->[ $my_recursions_remaining ] = [$my_triangle_initial];
90:
91:     # initial call to recursive subroutine
92:     MathPerl::Fractal::Sierpinski::sierpinski($my_triangle_initial, $my_recursions_remaining, $my_triangle_groups);
93:
94:     # regex to (globally) search for numbers incorrectly wrapped in 'single-quotes' by Dumper,
95:     # replace by // empty string, no lvalue $variable so directly (r)eturn modified string;
96:     # https://perldoc.perl.org/perlops%2FPATTERN%2FREPLACEMENT%2Fmsixpodualngcr
97:     print 'have $my_triangle_groups = ', "\n", (Dumper($my_triangle_groups) =~ s/'//gr);
98:
99:     # [ INITIALIZE GRAPHICS ]
100:
101:     # https://metacpan.org/dist/SDL/view/lib/pods/SDL/Events.pod
102:     my @SDL_EVENTS = qw(
103:         no_such_event
104:         SDL_ACTIVEEVENT
105:         SDL_KEYDOWN SDL_KEYUP
106:         SDL_MOUSEMOTION SDL_MOUSEBUTTONDOWN SDL_MOUSEBUTTONUP
107:         SDL_JOYAXISMOTION SDL_JOYBALLMOTION SDL_JOYHATMOTION SDL_JOYBUTTONDOWN SDL_JOYBUTTONUP
108:         SDL_QUIT
109:         SDL_SYSWMEVENT
110:         SDL_VIDEORESIZE SDL_VIDEORESIZE
111:         SDL_USEREVENT
112:         SDL_NUMEVENTS
113:     ); # constant data
114:
115:     # SDL includes moved into [[[ INCLUDES ]]] section above
116:
117:     # initialize SDL video & application & event;
118:     # we do not call $my SDL_app->run() anywhere in this program, instead we use the while() run loop below
119:     SDL::init(SDL_INIT_VIDEO);
120:     my SDLx::App $my SDL_app = SDLx::App->new(
121:         title => 'Merry Christmas!! Perl Advent 2022!!!!',
122:         width => 1024, # hard-coded 1024x768 resolution
123:         height => 768,
124:         depth => 32, # hard-coded 32-bit color
125:         resizable => 1 # allow window resize; does not scale window contents
126:     );
127:     my $my SDL_event = SDL::Event->new;
128:
129:     # [ RENDER STATIC GRAPHICS ]
130:
131:     # draw Christmas tree branches & snow tinsel & ornaments & lights;
132:     # iterate through triangle groups in reverse order, due to reverse population during recursion
133:     for (my $i = ((scalar @{$my_triangle_groups}) - 1); $i >= 0; $i--) {
134:         my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[$i];
135:         my integer::arrayref $my_color = $my_triangle_colors->[$i];
136:
137:         for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
138:             my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
139:
140:             # https://metacpan.org/dist/SDL/view/lib/pods/SDLx/Surface.pod
141:             $my SDL_app->draw_trigon_filled( $my_triangle, $my_color );
142:
143:             # refresh window on every triangle for fun cascade drawing effect
144:             $my SDL_app->update();
145:         }
146:     }
147:
148:     # draw Christmas tree trunk & Star of Bethlehem
149:     $my SDL_app->draw_rect( $my_rectangle_trunk, $my_color_brown );
150:     $my SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
151:     $my SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );
152:     $my SDL_app->update(); # refresh window
153:
154:     # [ RENDER DYNAMIC (ANIMATED) GRAPHICS ]
155:
156:     # set initial index for accessing Christmas tree lights colors
157:     my integer $my_lights_colors_index = 0;
158:
159:     # set initial time for changing Christmas tree lights colors
160:     (my integer $seconds_start) = gettimeofday();
161:     #print 'have $seconds_start = ', $seconds_start, "\n";
162:
163:     # the main run loop, used instead of calling $my SDL_app->run();
164:     # animate forever, until SDL_QUIT event is received in GUI window via <Alt-F4> keypress or window close mouse click,
165:     # or in CLI window via <Ctrl-C> keypress
166:     while(1)
167:     {
168:         # pump the event loop, gathering events from input devices
169:         SDL::Events::pump_events();
170:
171:         # poll for currently pending events
172:         if(SDL::Events::poll_event($my SDL_event))
173:         {
174:             print 'have @SDL_EVENTS[' , $my SDL_event->type(), ' ] = ', @SDL_EVENTS[$my SDL_event->type()], "\n";
175:
176:             # we only care about the SDL_QUIT event telling us to exit
177:             if ($my SDL_event->type == SDL_QUIT) {
178:                 print 'SDL_QUIT event received, exiting', "\n";
179:                 exit;
180:             }
181:         }
182:
183:         # get current time, for comparison with start time of current Christmas tree lights color
184:         (my integer $seconds_current) = gettimeofday();
185:         # print 'have $seconds_current = ', $seconds_current, "\n";
186:
187:         # twinkle Christmas tree lights every 1 second
188:         if (($seconds_current - $seconds_start) >= 1) {
189:             # reset start time to current time, for time cycle of next animation frame
190:             $seconds_start = $seconds_current;
191:
192:             # iterate through lights colors
193:             my integer::arrayref $my_color = $my_lights_colors->[$my_lights_colors_index];
194:             print 'have $my_color = $my_lights_colors->[' , $my_lights_colors_index, ' ] = ', Dumper($my_color);

```

```
195:
196:     # wrap back to beginning of lights colors when end is reached
197:     $my_lights_colors_index++;
198:     if ($my_lights_colors_index > ((scalar @{$my_lights_colors}) - 1)) {
199:         $my_lights_colors_index = 0;
200:     }
201:
202:     # only update second-smallest triangles, not the green of the Christmas tree itself
203:     my number::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[1];
204:     for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
205:         my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
206:         $my SDL_app->draw_trigon_filled( $my_triangle, $my_color );
207:     }
208:
209:     # redraw green of Christmas tree
210:     $my_triangle_group = $my_triangle_groups->[0];
211:     for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
212:         my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
213:         $my SDL_app->draw_trigon_filled( $my_triangle, $my_triangle_colors->[0] );
214:     }
215:
216:     # redraw Star of Bethlehem
217:     $my SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
218:     $my SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );
219:
220:     # refresh window once for every Christmas tree lights color change, for synchronized lights
221:     $my SDL_app->update();
222: }
223:
224: # briefly pause between each while() loop iteration, to avoid overloading CPU;
225: # ( 1_000_000 microseconds per second ) / ( 10_000 microseconds per iteration ) = 100 iterations per second;
226: # need at least 100 while loop iterations per second, in order to process all of the otherwise-ignored
227: # SDL_MOUSEMOTION events which are caused by simply moving the mouse over top of the window
228: usleep(10_000);
229: }
230:
231: return;
232: } # end of generate_fractal__render_animation()
233:
234: 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`.

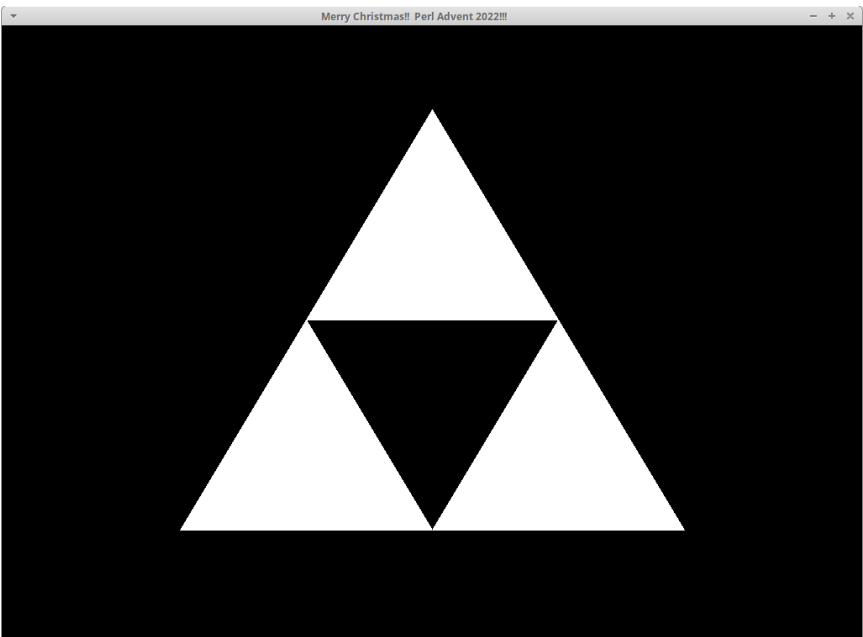
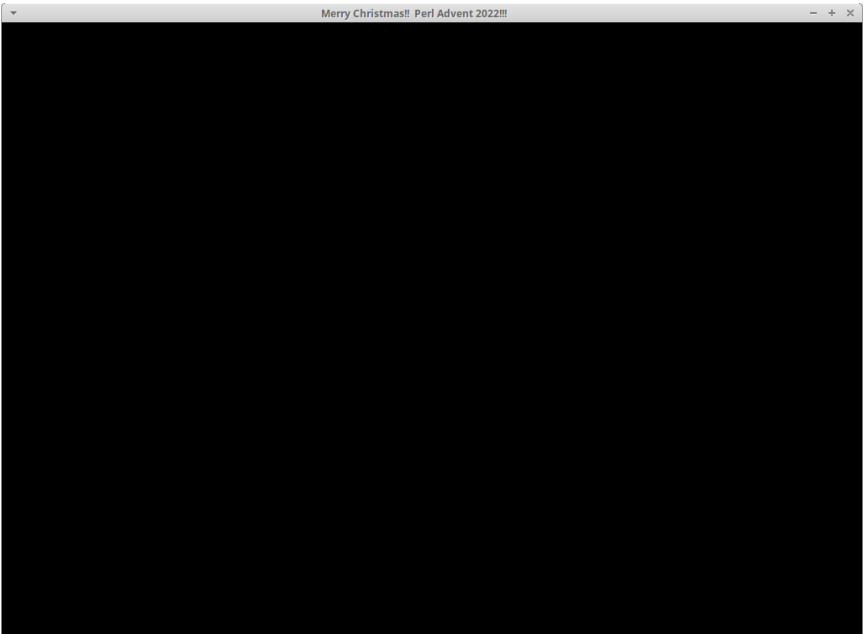
Last, we only need a few lines of driver code to run it all:

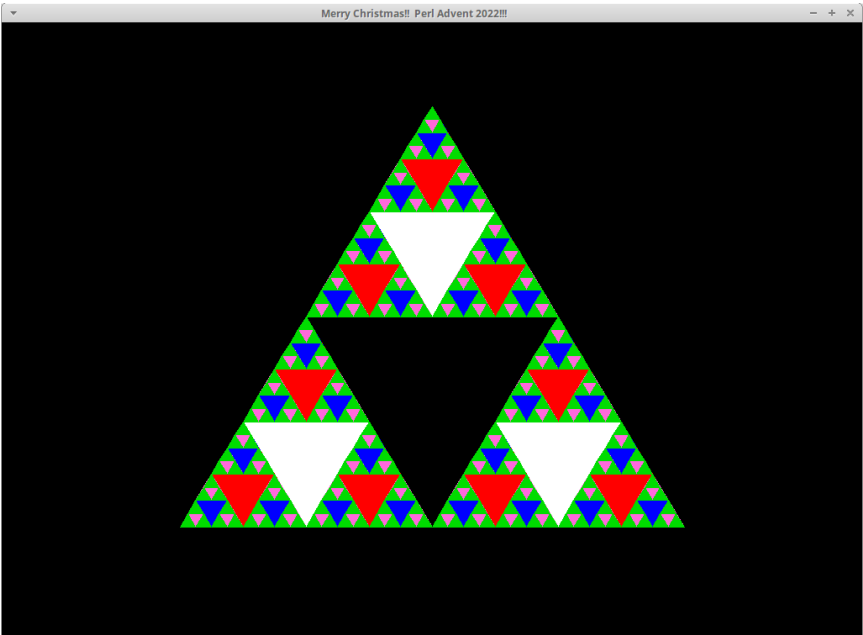
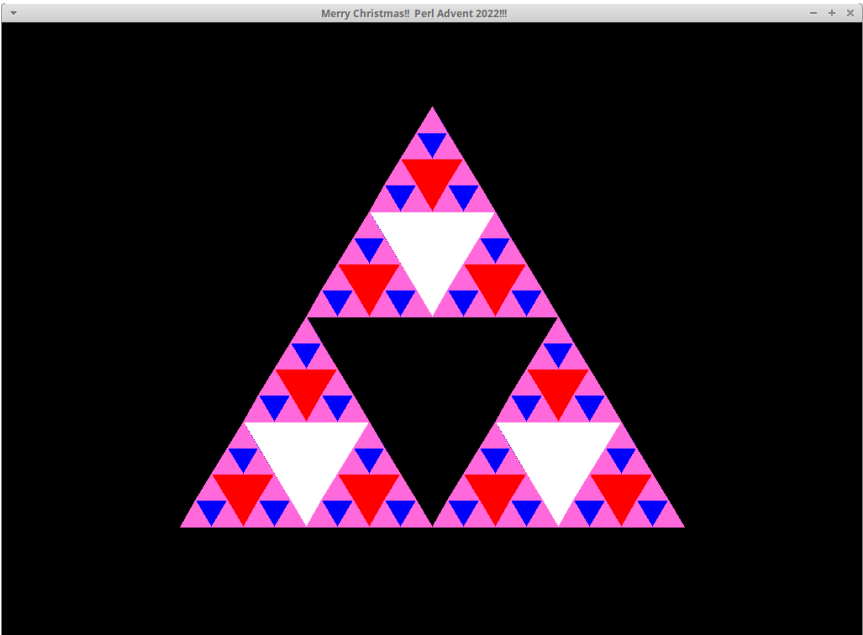
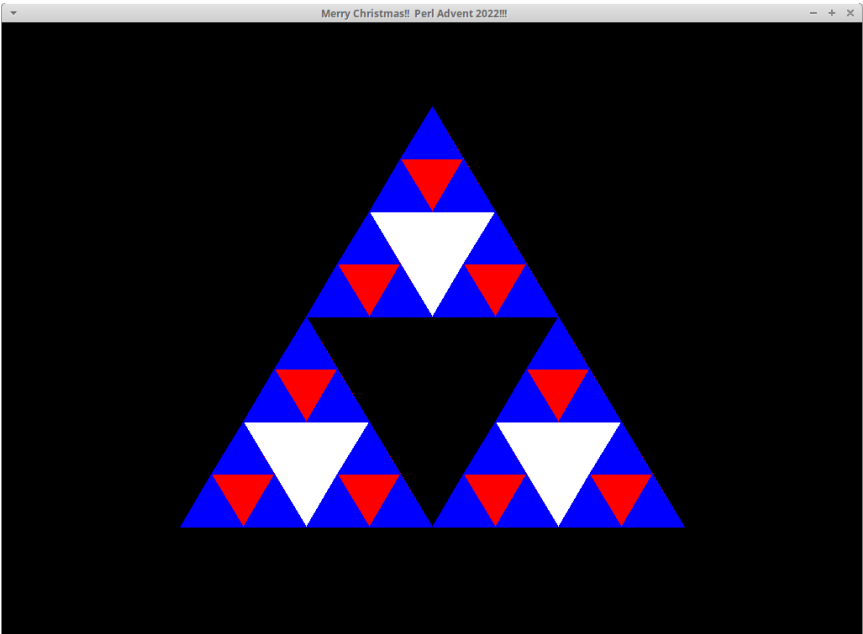
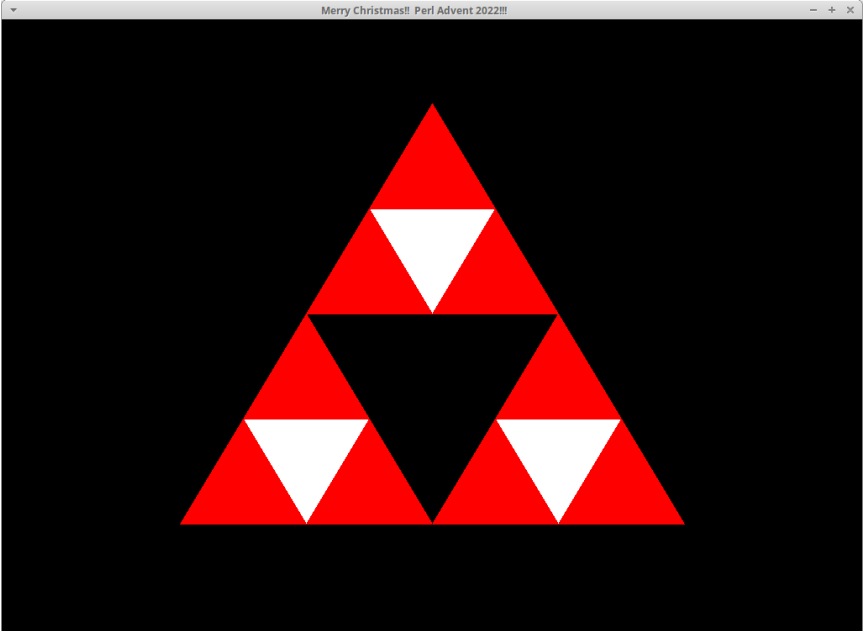
sierpinski\_triangles\_christmas.pl

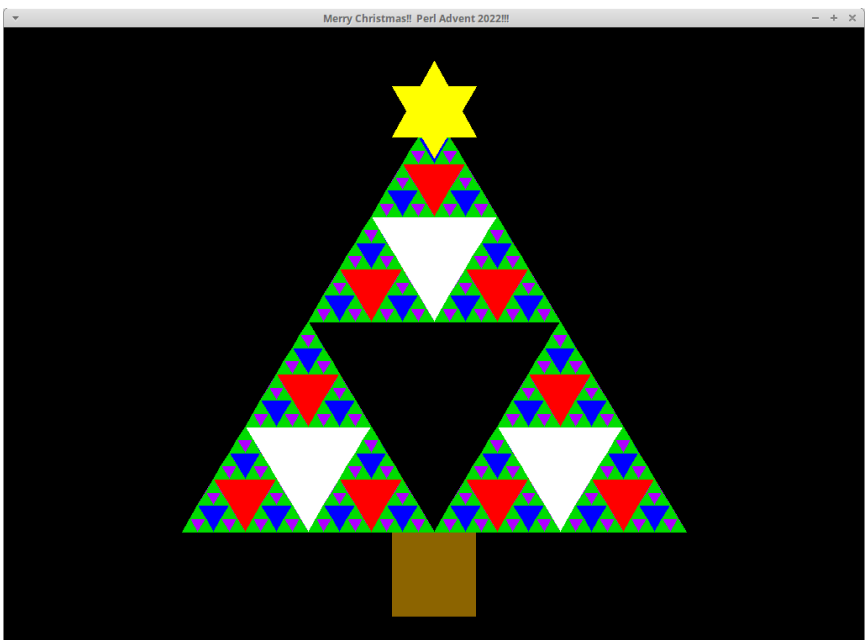
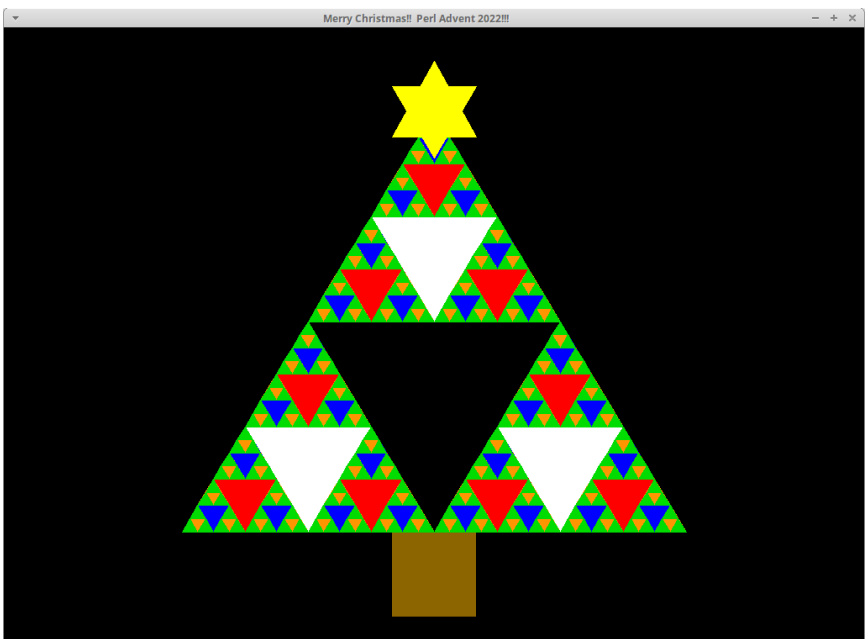
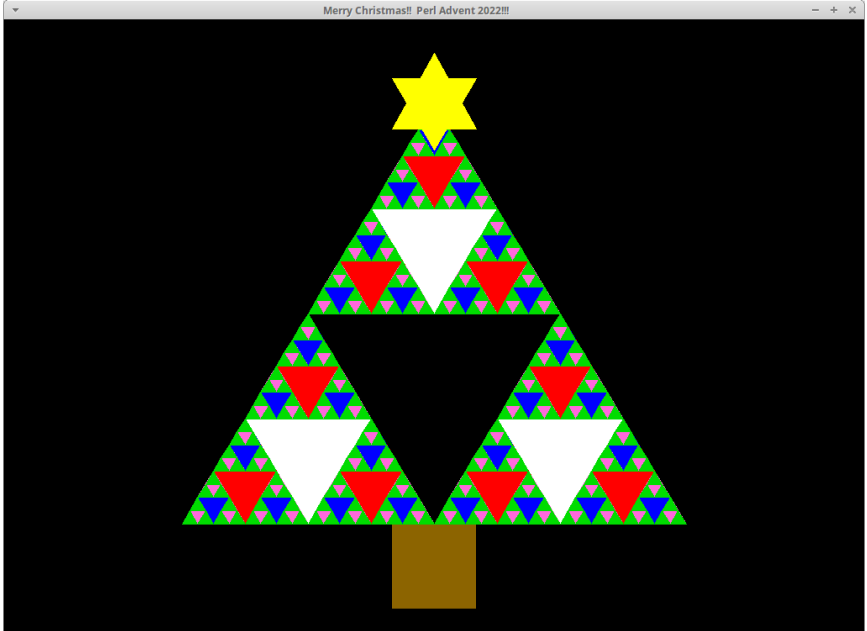
```
1: #!/usr/bin/env perl
2:
3: # Fractal Christmas Tree
4: # Sierpinski triangles animated using SDL graphics
5:
6: # [[[ HEADER ]]]
7: use strict;
8: use warnings;
9: our $VERSION = 0.008_000;
10:
11: # [[[ INCLUDES ]]]
12: use MathPerl::Fractal::ChristmasTree;
13:
14: # [[[ OPERATIONS ]]]
15: MathPerl::Fractal::ChristmasTree::generate_fractal__render_animation();
```

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:

```
1: $ ./sierpinski_triangles_christmas.pl
```







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

```
1: $ 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](#)

```
1: #!/usr/bin/env perl
2:
3: # Fractal Christmas Tree, Monolithic Single File
4: # Sierpinski triangles animated using SDL graphics
5:
6: # [[[ PREPROCESSOR ]]]
7: # declare Perl-compatible data types & data structures
8: package void; 1;
9: package integer; 1;
10: package integer::arrayref; 1;
11: package integer::arrayref::arrayref; 1;
12: package number; 1;
13: package number::arrayref; 1;
14: package number::arrayref::arrayref; 1;
15: package number::arrayref::arrayref::arrayref; 1;
16:
17: # [[[ HEADER ]]]
18: package main;
19: use strict;
20: use warnings;
21: use v5.14; # required for /r return AKA non-destructive regex flag
22: our $VERSION = 0.008_000;
23:
24: # [[[ CRITICS ]]]
25: # USER DEFAULT 1: allow numeric values & print operator
26: ## no critic qw(ProhibitUselessNoCritic ProhibitMagicNumbers RequireCheckedSyscalls)
27: ## no critic qw(RequireInterpolationOfMetachars) # USER DEFAULT 2: allow single-quoted control characters & sigils
28: ## no critic qw(ProhibitConstantPragma ProhibitMagicNumbers) # USER DEFAULT 3: allow constants
29:
30: # [[[ INCLUDES ]]]
31: use English;
32: use Data::Dumper;
33: $Data::Dumper::Deepcopy = 1; # display human-readable numeric data, not $VAR1->[0] references
34:
35: # https://metacpan.org/dist/SDL/view/lib/pods/SDL/Event.pod
36: use SDL;
37: use SDLx::App; # used for window creation & control
38: use SDL::Event; # used for creating Event object
39: use SDL::Events; # used for Event queue handling functions
```

```
use Time::HiRes qw( gettimeofday sleep ); # used for time-based animation control
40:
41: # [[[ CONSTANTS ]]]
42:
43:
44: # [ DATA FOR SIZES & SHAPES & COLORS;
45: #   HARD-CODED 1024x768 RESOLUTION & 32-BIT COLOR DEPTH ]
46:
47: # initial triangle's 3 corners as [x, y] Euclidean coordinates
48: my number::arrayref::arrayref $my_triangle_initial =
49:   [[ 512, 100], # top point
50:    [ 212, 600], # bottom left
51:    [ 812, 600]]; # bottom right
52:
53: my integer::arrayref $my_color_red      = [255, 000, 000, 255];
54: my integer::arrayref $my_color_pink     = [255, 105, 220, 255];
55: my integer::arrayref $my_color_orange   = [255, 150, 000, 255];
56: my integer::arrayref $my_color_yellow   = [255, 255, 000, 255];
57: my integer::arrayref $my_color_green    = [000, 220, 000, 255];
58: my integer::arrayref $my_color_blue     = [000, 000, 255, 255];
59: my integer::arrayref $my_color_purple   = [175, 000, 255, 255];
60: my integer::arrayref $my_color_white    = [255, 255, 255, 255];
61: my integer::arrayref $my_color_brown    = [140, 100, 000, 255];
62:
63: # colors as [r, g, b] triplets; number of colors is number of recursions
64: my integer::arrayref::arrayref $my_triangle_colors =
65:   [ $my_color_green, # green needs to be the color of the smallest, and thus most numerous, triangles
66:     $my_color_pink,
67:     $my_color_blue,
68:     $my_color_red,
69:     $my_color_white ];
70: my integer $my_recursions_remaining = scalar(@{$my_triangle_colors});
71:
72: # rectangle in format [ x, y, width, height ]
73: my number::arrayref::arrayref $my_rectangle_trunk =
74:   [462, 601, 100, 100];
75:
76: my number::arrayref::arrayref $my_triangle_star_up =
77:   [[ 512, 050], # top point
78:    [ 462, 130], # bottom left
79:    [ 562, 130]]; # bottom right
80: my number::arrayref::arrayref $my_triangle_star_down =
81:   [[ 512, 155], # bottom point
82:    [ 462, 70], # top left
83:    [ 562, 70]]; # top right
84:
85: # colors for animated Christmas tree lights
86: my integer::arrayref::arrayref $myLights_colors = [$my_color_pink, $my_color_purple, $my_color_orange];
87:
88: # https://metacpan.org/dist/SDL/view/lib/pods/SDL/Events.pod
89: my @SDL_EVENTS = qw(
90:   no_such_event
91:   SDL_ACTIVEEVENT
92:   SDL_KEYDOWN SDL_KEYUP
93:   SDL_MOUSEMOTION SDL_MOUSEBUTTONDOWN SDL_MOUSEBUTTONUP
94:   SDL_JOYAXISMOTION SDL_JOYBALLMOTION SDL_JOYHATMOTION SDL_JOYBUTTONDOWN SDL_JOYBUTTONUP
95:   SDL_QUIT
96:   SDL_SYSWMEVENT
97:   SDL_VIDEORESIZE SDL_VIDEOEXPOSE
98:   SDL_USEREVENT
99:   SDL_NUMEVENTS
100: ); # constant data
101:
102: # [[[ OPERATIONS ]]]
103:
104: # [ PREPARE & MAKE INITIAL RECURSIVE CALL ]
105:
106: # declare & initialize final array outside of the recursive subroutine for easy direct access by all recursive calls
107: my number::arrayref::arrayref::arrayref::arrayref $my_triangle_groups = [];
108:
109: # initial triangle is in a triangle group by itself
110: $my_triangle_groups->[$my_recursions_remaining] = [$my_triangle_initial];
111:
112: # initial call to recursive subroutine
113: sierpinski($my_triangle_initial, $my_recursions_remaining, $my_triangle_groups);
114:
115:
116: # regex to (globally) (s)earch for numbers incorrectly wrapped in 'single-quotes' by Dumper,
117: # replace by // empty string, no lvalue $variable so directly (r)eturn modified string;
118: # https://perldoc.perl.org/perlops#%2FPATTERN%2FREPLACEMENT%2Fasixpodualngcer
119: print 'have $my_triangle_groups = ', "\n", (Dumper($my_triangle_groups) =~ s/'//gr);
120:
121: # [ INITIALIZE GRAPHICS ]
122:
123: # SDL includes moved into [[[ INCLUDES ]]] section above
124:
125: # initialize SDL video & application & event;
126: # we do not call $my SDL_app->run() anywhere in this program, instead we use the while() run loop below
127: SDL::init(SDL_INIT_VIDEO);
128: my SDLx::App $my SDL_app = SDLx::App->new(
129:   title => 'Merry Christmas!! Perl Advent 2022!!!!',
130:   width => 1024, # hard-coded 1024x768 resolution
131:   height => 768,
132:   depth => 32, # hard-coded 32-bit color
133:   resizable => 1 # allow window resize; does not scale window contents
134: );
135: my $my SDL_event = SDL::Event->new;
136:
137: # [ RENDER STATIC GRAPHICS ]
138:
139: # draw Christmas tree branches & snow tinsel & ornaments & lights;
140: # iterate through triangle groups in reverse order, due to reverse population during recursion
141: for (my $i = ((scalar @{$my_triangle_groups}) - 1); $i >= 0; $i--) {
142:   my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[$i];
143:   my integer::arrayref $my_color = $my_triangle_colors->[$i];
144:
145:   for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
146:     my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
147:
148:     # https://metacpan.org/dist/SDL/view/lib/pods/SDLx/Surface.pod
149:     $my SDL_app->draw_trigon_filled( $my_triangle, $my_color );
150:
151:     # refresh window on every triangle for fun cascade drawing effect
152:     $my SDL_app->update();
153:   }
154: }
155:
156: # draw Christmas tree trunk & Star of Bethlehem
157: $my SDL_app->draw_rect( $my_rectangle_trunk, $my_color_brown );
158: $my SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
159: $my SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );
160: $my SDL_app->update(); # refresh window
161:
162: # [ RENDER DYNAMIC (ANIMATED) GRAPHICS ]
163:
164: # set initial index for accessing Christmas tree lights colors
165: my integer $myLights_colors_index = 0;
166:
167: # set initial time for changing Christmas tree lights colors
168: (my integer $seconds_start) = gettimeofday();
169: #print 'have $seconds_start = ', $seconds_start, "\n";
170:
171: # the main run loop, used instead of calling $my SDL_app->run();
172: # animate forever, until SDL_QUIT event is received in GUI window via <Alt-F4> keypress or window close mouse click,
173: # or in CLI window via <Ctrl-C> keypress
174: while(1)
175: {
176:   # pump the event loop, gathering events from input devices
177:   SDL::Events::pump_events();
178:
179:   # poll for currently pending events
180:   if(SDL::Events::poll_event($my SDL_event))
181:   {
182:     print 'have @SDL_EVENTS[' . $my SDL_event->type(), ']' = ' ', @SDL_EVENTS[$my SDL_event->type()], "\n";
183:
184:     # we only care about the SDL_QUIT event telling us to exit
185:     if ($my SDL_event->type == SDL_QUIT) {
186:       print 'SDL_QUIT event received, exiting', "\n";
187:       exit;
188:     }
189:
190:     # get current time, for comparison with start time of current Christmas tree lights color
191:     (my integer $seconds_current) = gettimeofday();
192:     # print 'have $seconds_current = ', $seconds_current, "\n";
193:
194:     # twinkle Christmas tree lights every 1 second
195:     if (($seconds_current - $seconds_start) >= 1) {
196:       # reset start time to current time, for time cycle of next animation frame
197:       $seconds_start = $seconds_current;
198:
199:       # iterate through lights colors
200:       my integer::arrayref $my_color = $myLights_colors->[$myLights_colors_index];
201:       print 'have $my_color = $myLights_colors->[' . $myLights_colors_index, ']' = ' ', Dumper($my_color);
202:
203:       # wrap back to beginning of lights colors when end is reached
204:       $myLights_colors_index++;
205:       if ($myLights_colors_index > ((scalar @{$myLights_colors}) - 1)) {
206:         $myLights_colors_index = 0;
207:       }
208:
209:       # only update second-smallest triangles, not the green of the Christmas tree itself
210:       my number::arrayref::arrayref::arrayref $my_triangle_group = $my_triangle_groups->[1];
211:       for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
212:         my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
213:         $my SDL_app->draw_trigon_filled( $my_triangle, $my_color );
214:       }
215:
216:       # redraw green of Christmas tree
217:       $my_triangle_group = $my_triangle_groups->[0];
218:       for (my $j = 0; $j < (scalar @{$my_triangle_group}); $j++) {
219:         my number::arrayref::arrayref $my_triangle = $my_triangle_group->[$j];
220:         $my SDL_app->draw_trigon_filled( $my_triangle, $my_triangle_colors->[0] );
221:       }
222:
223:       # redraw Star of Bethlehem
224:       $my SDL_app->draw_trigon_filled( $my_triangle_star_up, $my_color_yellow );
225:       $my SDL_app->draw_trigon_filled( $my_triangle_star_down, $my_color_yellow );
226:
227:       # refresh window once for every Christmas tree lights color change, for synchronized lights
228:       $my SDL_app->update();
229:     }
230:
231:     # briefly pause between each while() loop iteration, to avoid overloading CPU;
```

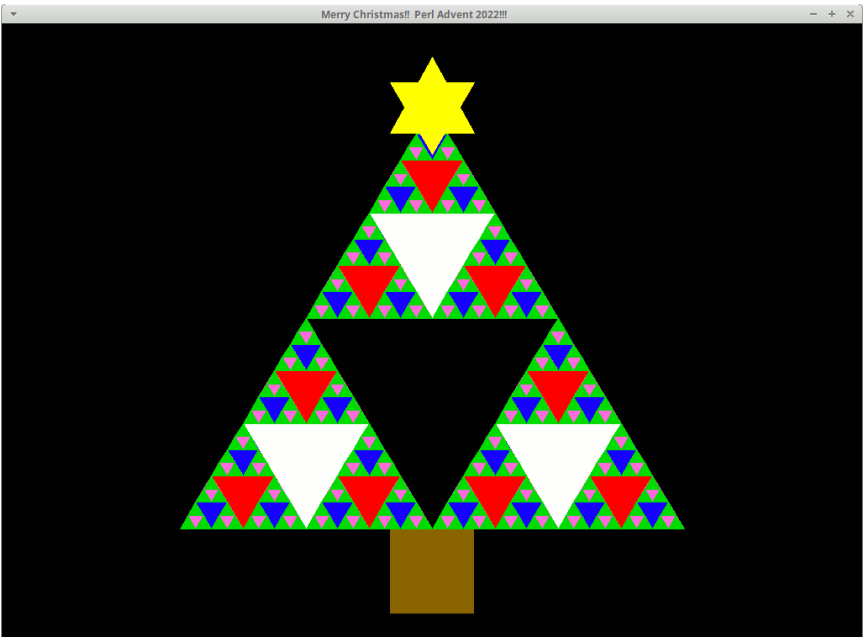
```
232: # ( 1_000_000 microseconds per second ) / ( 10_000 microseconds per iteration ) = 100 iterations per second;
233: # need at least 100 while loop iterations per second, in order to process all of the otherwise-ignored
234: # SDL_MOUSEMOTION events which are caused by simply moving the mouse over top of the window
235: usleep(10_000);
236: }
237:
238: # [[[ SUBROUTINES ]]]
239:
240: # recursively generate triangles, grouped by recursion level
241: sub sierpinski {
242:   { my void $RETURN_TYPE };
243:   (
244:     my number::arrayref::arrayref $triangle,
245:     my integer $recursions_remaining,
246:     my number::arrayref::arrayref::arrayref::arrayref $triangle_groups
247:   ) = @ARG;
248:
249:   print 'in sierpinski(), received $recursions_remaining = ', $recursions_remaining, "\n";
250:   print 'in sierpinski(), received $triangle = ', (Dumper($triangle) =~ s/'//gr), "\n";
251:
252:   if ($recursions_remaining > 0) {
253:     # shortcut variables, easier to read in midpoint calculations below
254:     my number::arrayref $point_a = $triangle->[0];
255:     my number::arrayref $point_b = $triangle->[1];
256:     my number::arrayref $point_c = $triangle->[2];
257:
258:     # calculate midpoints between two coordinates [x1, y1] and [x2, y2] is [(x1+x2)/2, (y1+y2)/2]
259:     my number::arrayref $point_a_b =
260:       [ (($point_a->[0] + $point_b->[0]) / 2),
261:         (($point_a->[1] + $point_b->[1]) / 2) ];
262:     my number::arrayref $point_a_c =
263:       [ (($point_a->[0] + $point_c->[0]) / 2),
264:         (($point_a->[1] + $point_c->[1]) / 2) ];
265:     my number::arrayref $point_b_c =
266:       [ (($point_b->[0] + $point_c->[0]) / 2),
267:         (($point_b->[1] + $point_c->[1]) / 2) ];
268:
269:     # construct 3 sub-triangles from orinal points and newly-calculated midpoints
270:     my number::arrayref::arrayref $subtriangle_a = [ $point_a, $point_a_b, $point_a_c ];
271:     my number::arrayref::arrayref $subtriangle_b = [ $point_a_b, $point_b, $point_b_c ];
272:     my number::arrayref::arrayref $subtriangle_c = [ $point_a_c, $point_b_c, $point_c ];
273:
274:     # $triangle_groups is zero-indexed like all other Perl arrays,
275:     # so we need to subtract one from $recursions_remaining before using as an index,
276:     # in order to avoid an undefined element at element 0;
277:     # also, we need to decrement $recursions_remaining before making recursive calls;
278:     # for both of these reasons, we can decrement now
279:     $recursions_remaining--;
280:
281:     # store all triangles grouped by recursion level
282:     push @$triangle_groups->[$recursions_remaining], $subtriangle_a;
283:     push @$triangle_groups->[$recursions_remaining], $subtriangle_b;
284:     push @$triangle_groups->[$recursions_remaining], $subtriangle_c;
285:
286:     # recurse once for each sub-triangle
287:     sierpinski( $subtriangle_a, $recursions_remaining, $triangle_groups);
288:     sierpinski( $subtriangle_b, $recursions_remaining, $triangle_groups);
289:     sierpinski( $subtriangle_c, $recursions_remaining, $triangle_groups);
290:   }
291:
292:   # return after maximum recursion level is reached (conditional block above not entered),
293:   # or all recursion calls have returned (conditional block above entered);
294:   # no return value, all generated data is stored directly in $triangle_groups
295:   return;
296: }
297:
298: 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!

```
1: $ ./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>

[Previous](#)

[Next](#)