Morphing Super-shapes

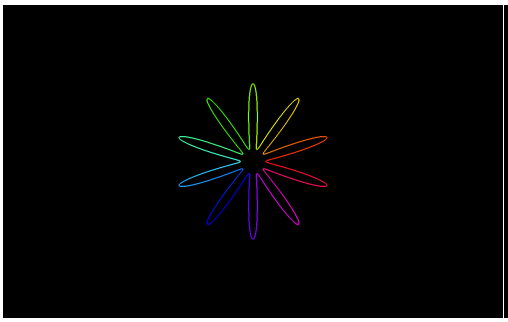
I have created a program that investigates the super-formula and it's 'super-shapes' and the transitions in between the different shapes. Therefore you can morph between all the shapes I've created, play with the morphing speed and completely stop the morphing, if you find an interesting shape. Wikipedia says that the SuperFormula was created to describe complex shapes and curves found in nature by Johan Gielis.

By pressing down a keyboard button the shape morphs into a new shape and melody gets played. Each shape has a melody piece, which I have composed, that can be played over the chords of the saw-wave-piece, which plays through the whole program.

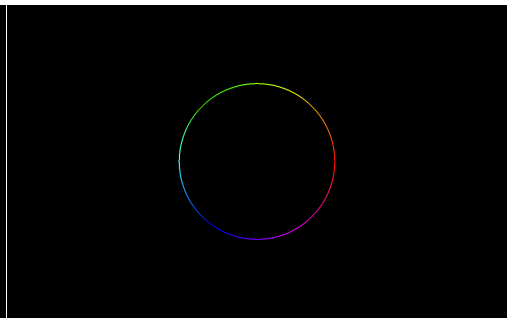
Since I don't know all the shape names, I have named them myself; the name of the shape is on the top of the picture.

Here are the six shapes used in the program:

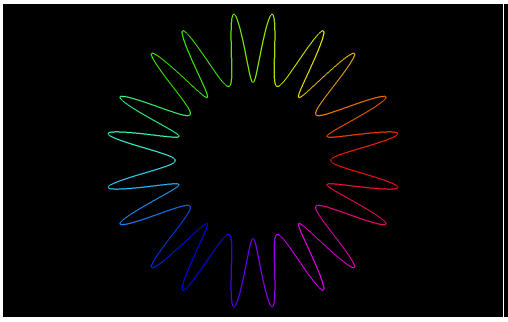
Flower



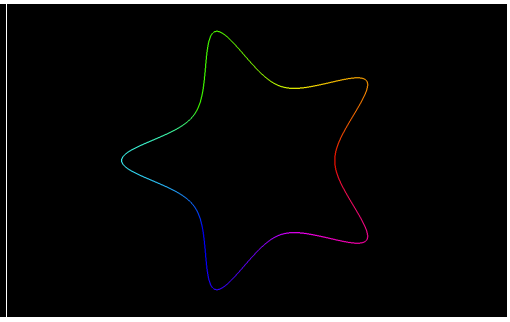
Circle



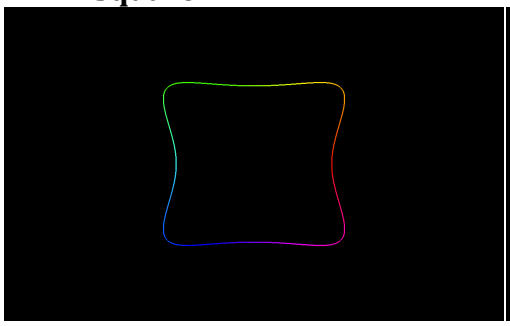
Sun



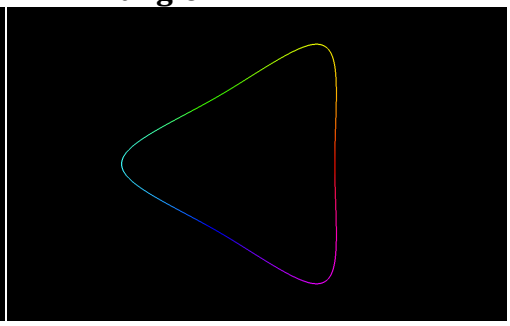
Star



Square



Triangle



Generating the shapes

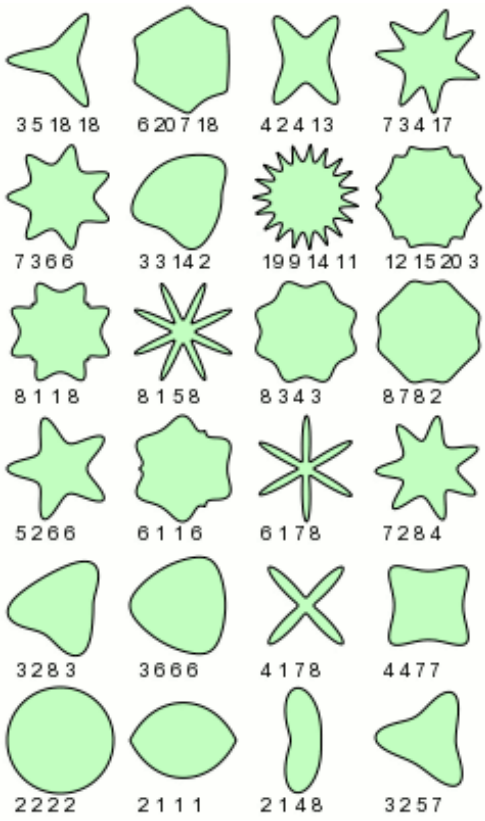
Instead of making the shapes from scratch for every new shape that I wanted, I have created a class called Shape, which handles the basic setup for every new shape. The class' default constructor set the primitive, how many lines/how high resolution, color, all the member variables and generates the first shape as a circle. I wanted to have all the different shapes to have the same primitive, size and color. So when a new shape is created from the Shape class, only the parameters for what kind of shape is wanted, needs to be dealt with.

SuperFormula

$$r(\varphi) = \left[\left| \frac{\cos\left(\frac{m\varphi}{4}\right)}{a} \right|^{n2} + \left| \frac{\sin\left(\frac{m\varphi}{4}\right)}{b} \right|^{n3} \right]^{-\frac{1}{n1}}$$

Superformula equation[1]

The algorithm Superformula can generate different super shapes and it has six different unknown variables in the equation.
By changing the different variables you can create different shapes. I've chosen to focus on 'm', 'n1', 'n2', 'n3' and the 'a' value. I have played around with the different combinations and I found the six shapes the program has now, works well.
It is quite difficult for me to understand how every parameter work, since it is a complex formula and I have only tried different combinations in my own program.
The 'm' variable controls how many sides or 'spikes' the shape should have, the circle has two, square 4 and the 'sun' shape has 20. The five other variables control the size, proportion and curve shapes. You can also make shapes that



[1] Calculation used in the source code line 33Picture from Wikipedia, http://en.wikipedia.org/wiki/Superformula, Both pictures from Wikipedia

are not connected. Unfortunately, my program doesn't take exactly the same numbers as the chart suggest.
For example, to make an almost perfect circle, you need to plot in m = 2, n1 = 1, n2 = 1 and n3 = 2 , not m = 2,n1 = 2, n2 = 2 and n3 = 2 as Wikipedia says.
My 2, 2, 2 and 2 circle looks more like an ellipse instead of a circle.[2]

The formula in the code gets looped through, as many times as there is 'dots'. The variable theta changes for every loop and when you put theta in the Superformula in line 35[3], one line of the super-shape gets created. The output of the formula is the radius, but it is in polar coordinates, so another calculation with radius multiplied by cos(theta) and sin(theta) needs to be done. It gets created in the same way as a circle, just with a bit more calculation.
At last, the mesh with the new x and y coordinate is created. The color is set for each dot using the HSV model, where the hue goes from 0 to 1. So all the colors are represented with a saturation and value of 1.

Draw function

The Shape class also has a member function, void draw, which takes the argument Graphics& g, which is an AlloSystem function. The draw function draws the mesh, set the scale, stroke and the position.[4] It is all done in reverse, because of the way the AlloSystem model matrix work.
The draw member function also makes sure that the push and pop functions are called, which allow us to modify the mesh.
The draw function gets called from the AlloSystems onDraw function, which gets called at every frame.

Morphing calculation

In the AlloSystem function onAnimate there is one if statement, that makes all the morphing, because I only wanted the shape to morph when a button is pressed. When one of the keyboard buttons is pressed, the value of sampleNumber changes, morphs mesh resets and

[2] See Figure 1 in the Appendix
[3] See Figure 2 in the Appendix
[4] See Figure 3 in the Appendix

the morphing starts, so if sampleNumber is not equal to minus 1 becomes true.[5] When the program enters the if statement, the variable frac starts counting up from 0 to 1, as fast as the onAnimate function gets called. These frame rates are stored in dt and I multiply with the mapped x coordinate, transformSpeed, from 0 to 1 to be able to control the speed of the morphing.[6] When frac is assigned to the first value, a for loop begins, going through all the vertices in morph and setting new coordinates of morph with v, relative to the shape it is morphing into. So in the first loop, i = 0 and the first vertex of morph is assigned to v1, the new shape's first vertex, is assigned to v2. Then the v1 value is multiplied by 1-frac and v2 is multiplied by frac. Then the two values are added together and assign to v. Therefor the v is now the coordinate for the first vertex in morph. The for-loop continues through all the vertices, and when it is done, morph gets reset again and a bigger value is assigned to frac. This keeps on going over and over again, until frac is more or equal to 1, meaning that morph has morphed into the new shape.

Sound

The program also includes seven different samples and I've created the sound to give the program a kind of 80's game vibe.

The first sample is a 30 second saw-wave piece that loops through out the entire program. The six other samples are small sine-wave melody pieces that can be played over the looped saw-wave sample.

The music combines unusual progressions in between chords and I like the way it matches the morphing. All the sample files are loaded into identifier using the samplePlayer class . All audio from the samplePlayer class gets called in the AlloSystem function onSound and out through speaker channel 0 and 1, that by convention relates left and right speaker.

Input and interaction

The keyboard input:

- m : sets sampleNumber to 0, creates a circle shape and plays the first sine-wave sample.
- n : sets sampleNumber to 1, creates a triangle shape and plays the second sine-wave sample.
- b : sets sampleNumber to 2, creates a square shape and plays the third sine-wave sample.
- v : sets sampleNumber to 3, creates a star shape and plays the fourth sine-wave sample.
- c : sets sampleNumber to 4, creates a sun shape and plays the fifth sine-wave sample.

[5] See Figure 4 in the Appendix
[6] See Figure 4, line 127

- x : sets sampleNumber to 5, creates a flower shape and plays the sixth sine-wave sample.
- Spacebar: sets sampleNumber to -1, which stops the sine samples and the morphing
- Enter/Return: restarts the saw-wave-piece, so it starts playing from the beginning.

Mouse drag and mouse left- and right-click uses the x and y coordinates.

- The x coordinate controls the morphing speed. The closer the mouse gets dragged or left-clicked to the right, the faster the transformation in between shapes happens.
- The y coordinates control the volume of the saw-wave track. The closer the mouse gets dragged or right-clicked to the top of the screen, the higher the volume is. If it is on the bottom of the screen, the volume is 0.

Improvements

The program has one problem. The frac is not going from 0 to 1 along with the morphing of the shape. So when frac is 1, the shape is already finished morphing. My thesis is that morph resets every time frac gets bigger, which means the morphing goes faster, every time it 'jumps' into the new shape. Also, it means that the sine samples get the end of the sample cut off when frac hits 1 or more.

Another way of improving the program would be to create different classes for each form that would inherit from the Shape class. So instead of having MyApp's constructor setting the form, it is already set from a derived class. Also a 'morphing function' would be quite useful if I wanted more meshes rendered, so I could use it simultaneously on two morphing shapes. To give even more control over the shapes, an interface where you could control all the parameters of the Superformula could be made. Then the morphing would be a real-time response to what the user wants.

Appendix

Wikipedia on Superformula

- Link: http://en.wikipedia.org/wiki/Superformula

Inspiration

I had this video in mind, when I decided to work with morphing shapes.

- Journey to the center of a triangle, www.youtube.com

Link: https://www.youtube.com/watch?v=v_oZ9Pe0yRg

Soundfile names:

Saw-wave-piece.wav

Sineone.wav

Sinetwo.wav

Sinethree.wav

Sinefour.wav

Sinefive.wav

Sinesix.wav

Source code:

- MorphApp.cpp

Pictures

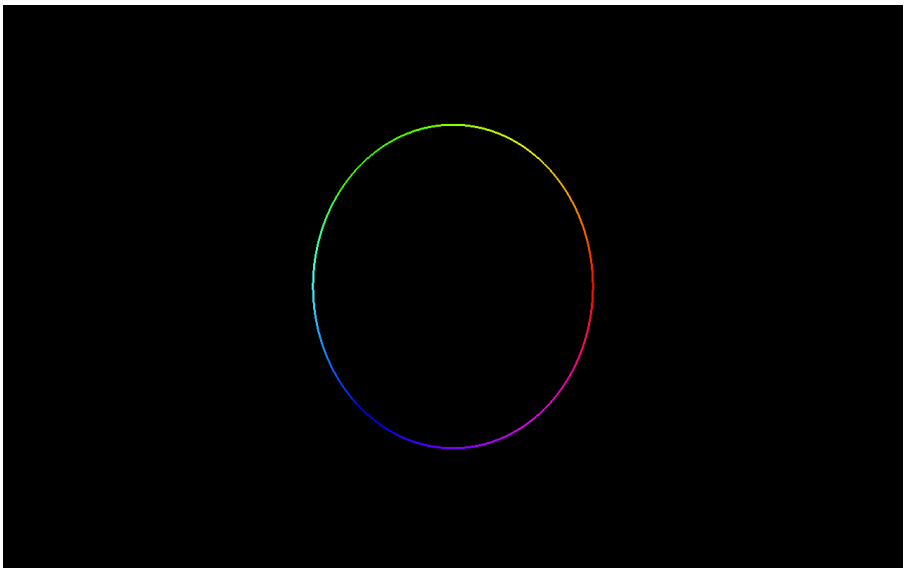


Figure 1

```
        // set up the primitive line for for mMesh
        mMesh.primitive(Graphics::LINE_LOOP);

        for(int i = 0; i < dots; i++){
        // calculate the angle, theta
        float theta = float(i) / dots * 2*M_PI;

        // Superformula takes six variables and theta
        radius = pow(((pow((cos(m*theta/4)/a), (n2*2))) + pow((sin(m * theta/4)/b), n3) ), (-1/n2));

        // convert polar to cartesian coordinates
        float x = radius * cos(theta);
        float y = radius * sin(theta);

        // create the mesh, using the generated x and y values
        mMesh.vertex(x, y);
        // add hue to each dot, going from 0 to 1
        mMesh.color(HSV(float(i)/dots, 1, 1));
        }
```

Figure 2

```
    void draw(Graphics& g){
        g.pushMatrix(Graphics::MODELVIEW);
            g.translate(mPosition);
            g.stroke(3);
            g.scale(mScale);
            g.draw(mMesh);
        g.popMatrix();
    }
```

Figure 3

```
void onAnimate(double dt){

// if sampleNumber not equal to one, change shape to the new desired shape with the index number of sampleNumber
if(sampleNumber != -1){

    morph.mMesh.reset();    // clears morphs mesh

    // frac is how far the morphing is, if 0 then nothing happens, if 1, then new desired shape is created
    // everything in between 0 and 1 is the actual morphing, the faster frac counts up, the faster the morphing
    frac += transformSpeed * dt;

    // for loops through all the vertices in the shapes and set all the vertices
    for(int i = 0; i < circle.mMesh.vertices().size(); i++){
        // create a 3-vector point v
        Vec3f v;

        // access the mesh through an array store every vertex in morph and the new shape
        Vec3f v1 = morph.mMesh.vertices()[i];
        Vec3f v2 = shape[sampleNumber].mMesh.vertices()[i];

        // frac count up from 0 to 1
        // the closer the frac is to 1, the more of the new shape is visible
        v = (v1 * (1-frac)) + (v2 * frac);

        // set morphs mesh to the v value
        morph.mMesh.vertex(v);
        // since the mesh was reseted, color also needs to added to every mesh point
        morph.mMesh.color(HSV(float(i)/circle.mMesh.vertices().size(), 1, 1));
    }
    if(frac >= 1){        // if frac is more or equal to 1, it means the morphing is done
        sampleNumber = -1; // set sampleNumber to -1, which will stop the morphing
    }
}
}
```

Figure 4