Programmable Style for NPR Line Drawing

Stéphane Grabli[1], *Emmanuel Turquin*[1], *Frédo Durand*[2] *and François X. Sillion*[1]
[1]ARTIS/GRAVIR-IMAG – INRIA [2]MIT

Code examples

In the following code examples we highlight in yellow the significant lines of code, the remaining lines consisting in declarations or use of standard mechanisms.
The built-in objects are written in blue.

Code for the Guiding Lines Style Module

We show here the python code of the style module that was used the for the "Guiding Lines" layer of Fig. 4 in the paper.
The main aspects of this style module are the *recursive split* used to make strokes start and stop at points of highest curvature and the *shader* that displaces the vertices of the stroke along its tangent.
The functions, predicates, shaders needed in the style module are first defined followed by the style module itself.

```
## zero-dimensional (0D) Unary Predicate, returns true if the u parameter
## of the parameterized stroke is inside a given range.
## -- inherits from UnaryPredicate0D
class pyParameterUP0D(UnaryPredicate0D):
      ## Builds the functor given the range
      def __init__(self,pmin, pmax):
            UnaryPredicate0D.__init__(self)
            self._m = pmin
            self._M = pmax
      ## Operator() - inter is an Interface0DIterator
      def __call__(self, inter):
            return ((inter.u()>=self._m) and (inter.u()<=self._M))


## Shader to displace stroke's vertices along its tangent
## -- inherits from StrokeShader
class pyGuidingLineShader(StrokeShader):
      ## shading method
      def shade(self, stroke):
            it = stroke.strokeVerticesBegin()  ## get the first vertex
            itlast = stroke.strokeVerticesEnd()##
            itlast.decrement()                 ## get the last one
            ## t <- tangent direction
            t = itlast.getObject().getPoint() - it.getObject().getPoint()
            ## look for the stroke middle vertex
            itmiddle = StrokeVertexIterator(it)
            while(itmiddle.getObject().u()<0.5):
                  itmiddle.increment()
            ## position all the vertices along the tangent
            while(it.isEnd() == 0):
                  it.getObject().SetPoint(itmiddle.getObject().getPoint() \
                  +t*(it.getObject().u()-itmiddle.getObject().u()))
                  it.increment()



## Shader stretch stroke at its extremities
```

```
## -- inherits from StrokeShader
class pyBackboneStretcherShader(StrokeShader):
      ## Builds the shader. The user specifies the desired stretching
      ## amount l.
      def __init__(self, l):
            StrokeShader.__init__(self)
            self._l = l
      ## shading method
      def shade(self, stroke):
            it0 = stroke.strokeVerticesBegin() ## first vertex
            it1 = StrokeVertexIterator(it0) ## --
            it1.increment()                 ## second vertex
            itn = stroke.strokeVerticesEnd() ## --
            itn.decrement()                 ## last vertex
            itn_1 = StrokeVertexIterator(itn) ## --
            itn_1.decrement()                 ## vertex n-1
            ## get the 2D coordinates for all these vertices:
            p0 = it0.getObject().getPoint()
            p1 = it1.getObject().getPoint()
            pn_1 = itn_1.getObject().getPoint()
            pn = itn.getObject().getPoint()
            ## Compute the directions of the stroke's first and last
            ## segments
            d1 = (p0-p1).normalize() ## first segment
            dn = (pn-pn_1).normalize() ## last segment
            ## move the first point along the first segment's direction
            it0.getObject().SetPoint(p0+d1*float(self._l))
            ## move the last point along the last segment's direction
            itn.getObject().SetPoint(pn+dn*float(self._l))

## ////////////////////////////////////////////////////////
##                      Style Module
## ////////////////////////////////////////////////////////
## -- selects the ViewEdge that are visible
Operators.select(QuantitativeInvisibilityUP1D(0))
## -- chains using the standard chaining iterator.
## By default, the chaining stays in the selection.
Operators.bidirectionalChain(ChainSilhouetteIterator())
## -- splits the chains at points of highest 2D curvature.
## The chains are split until a certain length is reached.
## We prevent the splitting points from being near extremities
Operators.recursiveSplit(
              Curvature2DF0D(), ## the 0D function we evaluate
              pyParameterUP0D(0.2,0.8), ## The Predicate 0D that preselects
                                        ## the candidate splitting points
              NotUP1D(LengthHigherUP1D(75)), ## The 1D predicate that tells
                                             ## when to stop the recursion
              2) ## The sampling
## -- list of shaders
shaders_list =    [
            StrokeTextureShader("pencil.jpg", ##  pencil texture with tips
                                Stroke.DRY_MEDIUM, 1),
            ConstantColorShader(0,0,0,1), ## assigns a constant color RGBA
            ConstantThicknessShader(2.0)  ## assigns a constant thickness
            pyGuidingLineShader(),## modifies stroke's geometry with
                                  ## respect to its tangent
            pyBackboneStretcherShader(0.2) ## stretches the stroke
                  ]
## -- creates the strokes
Operators.create(TrueUP1D(),shaders_list)
```

Code for the External Contour Style Module

We show here the python code of the style module that was used the for the “External Contour” layer of Fig. 4 in the paper.
The main aspects of this style module are in the chaining and in the calligraphic shader.
The functions, predicates, shaders needed in the style module are first defined followed by the style module itself.

```
## Assigns a calligraphic thickness to the strokes.
## -- inherits from StrokeShader
class pyCalligraphicThicknessShader(StrokeShader):
      ## Builds the shader. The user specifies the min thickness m,
      ## the max thickness M, the calligraphic direction V.
      def __init__(self, m, M, V):
            StrokeShader.__init__(self)
            self._m = m
            self._M = M
            self._V = V
            self._getNormal2D = Normal2DF0D()##instanciate
                                                      ## the functor that
                                                      ##gets normal
                                                      ##info
      ## shading method
      def shade(self, stroke):
            it = stroke.strokeVerticesBegin()
            while(it.isEnd() == 0):
                  ## gets the normal to the curve
                  n2d = self._getNormal2D(it).normalize()
                  ## compute the thickness depending on the
                  ## angle between the normal and the desired
                  ## calligraphic direction
                  t = self._m + fabs(n2d*self._V)*(self._M-self._m)
                  ## assigns the found thickness to the StrokeVertex
                  ## attribute
                  it.getObject().attribute().setThickness(t/2.0,t/2.0)
                  it.increment()

## ////////////////////////////////////////////////////////
##                      Style Module
## ////////////////////////////////////////////////////////
## -- selects the ViewEdge that are visible and external contour
Operators.select(AndUP1D(QuantitativeInvisibilityUP1D(0),
                         ExternalContourUP1D())
## -- chains using the ChainPredicateIterator that takes a
## one-dimensional predicate as argument. This predicate
## is evaluated to decide whether a ViewEdge must be included in
## the chain or not. We pass a predicate that tests whether a ViewEdge
## is visible and is an external contour.
Operators.bidirectionalChain(ChainPredicateIterator(
                             AndUP1D(QuantitativeInvisibilityUP1D(0),
                                     ExternalContourUP1D())))
## -- list of shaders
shaders_list =    [
            ConstantColorShader(0.2,0.2,0.2,1),## assigns a constant color
                                               ## RGBA
            ## assigns a calligraphic
            ## thickness
            PyCalligraphicThicknessShader(6, ## Min thickness
                                          40, ## Max Thickness
                                          Vec2f(1, 1), ## orientation
                                          1)##clamp
                  ]
```

```
## -- creates the strokes
Operators.create(TrueUP1D(),shaders_list)
```

Code for the “not” External Contour Style Module

We show here the python code of the style module that was used the for the “not External Contour” layer of Fig. 4 in the paper.
This style module selects the ViewEdges that are visible but not external contours and apply a nonlinear varying thickness to the strokes.
The functions, predicates, shaders needed in the style module are first defined followed by the style module itself.

```
def smoothC( a, exp ):
      c = pow(float(a),exp)*pow(2.0,exp)
      return c

## Assigns a Thickness that increases and then decreases
## in a non-linear way.
## -- inherits from StrokeShader
class pyNonLinearVaryingThicknessShader(StrokeShader):
      ## Builds the shader. The user specifies the min thickness m,
      ## the max thickness M, the exponent
      def __init__(self, m, M, e):
            StrokeShader.__init__(self)
            self._m = m
            self._M = M ## M is reached in the middle of the stroke
            self._e = e
      ## shading method
      def shade(self, stroke):
                  n = stroke.strokeVerticesSize()
                  i = 0
                  it = stroke.strokeVerticesBegin()
                  while it.isEnd() == 0:
                        att = it.getObject().attribute()
                        if(i < float(n)/2.0):
                              c = float(i)/float(n)
                        else:
                              c = float(n-i)/float(n)
                        c = smoothC(c, self._e)
                        t = (1.0 - c)*self._M + c * self._m
                        att.setThickness(t/2.0, t/2.0)
                        i = i+1
                        it.increment()


## ////////////////////////////////////////////////////////
##                      Style Module
## ////////////////////////////////////////////////////////
## -- selects the ViewEdge that are visible and not external contour
Operators.select(AndUP1D(QuantitativeInvisibilityUP1D(0),
                         NotUP1D(ExternalContourUP1D())))
## -- chains using the standard chaining iterator.
## By default, the chaining stays in the selection.
Operators.bidirectionalChain(ChainSilhouetteIterator())
## -- list of shaders
shaders_list =    [
            ConstantColorShader(0.2,0.2,0.2,1),## assigns a constant color
                                               ## RGBA
            pyNonLinearVaryingThicknessShader(2,## min thickness
```

```
                                                    10,## max thickness
                                                    0.33),  ## the exponent

                  ]
## -- creates the strokes
Operators.create(TrueUP1D(),shaders_list)
```

Shader code examples

The first shader is used in Fig.12 to define the strokes wavy geometry. It basically consists in displacing the vertices along the normal to the curve, of an amount proportional to a sinusoidal curve.

```
## Shader to displace stroke's vertices with respect to a sinusoidal
## curve, whose frequency and amplitude are given as arguments.
## The amplitude is actually increasing from 0 to A and decreasing from
## A to 0 along the curve.
## -- inherits from StrokeShader
class pySinusDisplacementShader(StrokeShader):
      ## Builds the shader. The user specifies sinusoid amplitude a
      ## and frequency f.
      def __init__(self, f, a):
            StrokeShader.__init__(self)
            self._f = f
            self._a = a
            self._getNormal = Normal2DF0D() ## functor instanciation
      ## shading method
      def shade(self, stroke):
            it = stroke.strokeVerticesBegin()
            while it.isEnd() == 0:
                  v = it.getObject()## v is a StrokeVertex
                  ## we retrieve the normal n to the stroke at v
                  n = self._getNormal(it.castToInterface0DIterator())
                  ## the sinusoid amplitude is a(u) = A(1-2|u-0.5|).
                  ## This function looks like that :
                  ## /\ u belonging to [0,1]
                  a = self._a*(1-2*(fabs(v.u()-0.5)))
                  ## we modulate the amplitude by a cosinus of frequency
                  ## f, such as, for f=1, the curve spans a single
                  ## 2*PI period and displaces the vertex along its normal
                  ## of this amplitude
                  v.SetPoint(v.getPoint()+n*a*cos(self._f*v.u()*2*PI))
                  it.increment()
```

The second shader is used in this same figure (Fig.12) to define the stroke's color variation. It linearly varies from a first color c1 to a second color c2 as we are progressing from the first vertex toward the middle one, and from c2 to c1, as we're leaving the middle vertex to reach the last one.

```
## Shader to linearly interpolate between two colors. The stroke starts
## with the first color, reaches the second color in its middle and ends
## back with the first color.
## -- inherits from StrokeShader
class pyInterpolateColorShader(StrokeShader):
      ## Builds the shader. The user specifies the two colors as
      ## R,G,B,A components
      def __init__(self, r1,g1,b1,a1,r2,g2,b2,a2):
            StrokeShader.__init__(self)
            self._c1 = [r1,g1,b1,a1] ## first color array
            self._c2 = [r2,g2,b2,a2] ## second color array
      ## shading method
```

```
def shade(self, stroke):
    n = stroke.strokeVerticesSize()-1
    i = 0
    it = stroke.strokeVerticesBegin()
    while it.isEnd() == 0:
        att = it.getObject().attribute() ## we get the vertex's
                                         ## attribute
        ## The color for this vertex is defined by:
        ## color = (1-c)c1 + (c)c2 with:
        ## c(u) = (1-2|u-0.5|), and u belongs to [0,1]
        ## This function looks like that :
        ## /\
        c = 1-2*(fabs(float(i)/float(n)-0.5))
        ## Sets the interpolated RGB color
        att.setColor((1-c)* self._c1[0] + c* self._c2[0],
                     (1-c)* self._c1[1] + c* self._c2[1],
                     (1-c)* self._c1[2] + c* self._c2[2],)
        ## Sets the interpolated alpha component
        att.setAlpha((1-c)* self._c1[3] + c* self._c2[3],)
        i = i+1
        it.increment()
```

Function code examples

This example shows a zero-dimensional (0D) Function used to evaluate the norm of the gradient vector found in the global density map at a given scale for a vertex.

```
## 0D Function to compute the norm of the gradient vector computed
## at a given point and for a given scale in the global density map.
## -- inherits from UnaryFunction0D
class pyViewMapGradientNormF0D (UnaryFunction0DDouble):
    ## Builds the functor given the scale sigma
    def __init__(self,sigma):
        UnaryFunction0DDouble.__init__(self)
        self._s = sigma ## the sigma of the gaussian function
        self._step = pow(2, self._s)    ## the gap between two pixels
                                        ## used for computing the
                                        ## gradient at scale sigma
    ## Operator() - iter is an Interface0DIterator
    def __call__(self, iter):
        p = iter.getObject().getPoint2D()
        ## Compute the gradient's x component by reading the global
        ## density map at scale _s
        gx = ReadCompleteViewMapPixelCF(self._s,p.x()+self._step,p.y())
        -  ReadCompleteViewMapPixelCF(self._s, p.x()-self._step, p.y())
        ## Compute the gradient's x component by reading the global
        ## density map at scale _s
        gy = ReadCompleteViewMapPixelCF(self._s,p.x(),p.y()+self._step)
        - ReadCompleteViewMapPixelCF(self._s, p.x(), p.y()-self._step)
        return Vec2f(gx, gy).norm() ## returns the norm of the vector
```

Many built-in mechanisms facilitate the definition of new elements. For instance, we provide a standard integration mechanism that allows to easily implement a one-dimensional function from a zero-dimensional function. In particular, we show how to define the 1D function that evaluates the gradient intensity in the global density map at a given scale for a 1D element, from the 0D function defined above.

```
## 1D Function to compute the gradient intensity found in the global
## density map at a given scale for a 1D element.
## -- inherits from UnaryFunction1D
class pyViewMapGradientNormF1D(UnaryFunction1DDouble):
      ## Builds the functor given the scale sigma, the integration
      ## type and the sampling to use for the 0D evaluations.
      def __init__(self,sigma, integrationType=MEAN, sampling=2.0):
            UnaryFunction1DDouble.__init__(self, integrationType, sampling)
            self._getGradientNorm0D = pyViewMapGradientNormF0D(sigma)
      ## Operator() - inter is an Interface1D
      def __call__(self, inter):
            ## call to the standard integration mechanism
            return integrateDouble( self._getGradientNorm0D,
                                    inter.pointsBegin(self._sampling),
                                    inter.pointsEnd(self._sampling),
                                    self._integration)
```

Figures

Use of different style sheets on the same model

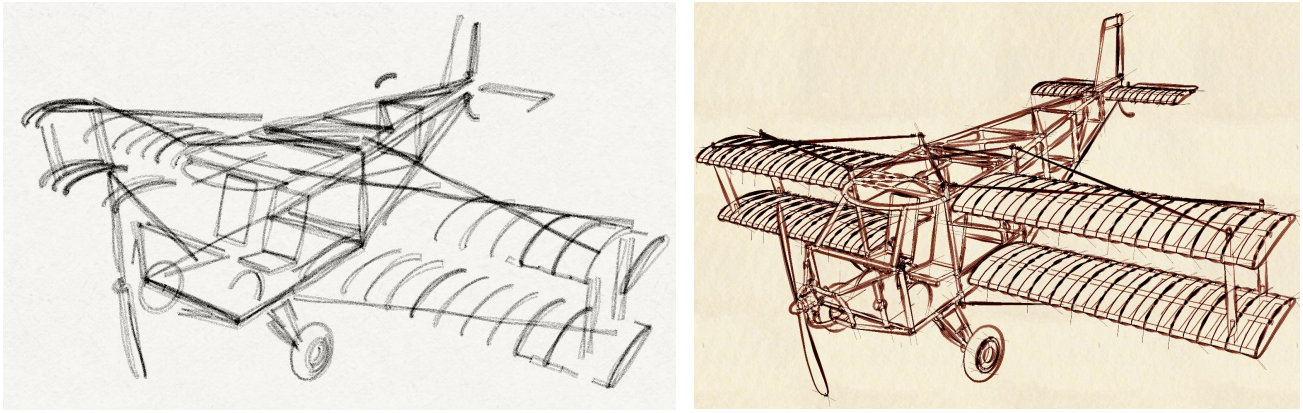

Figure 1 Two different styles applied to the same plane model

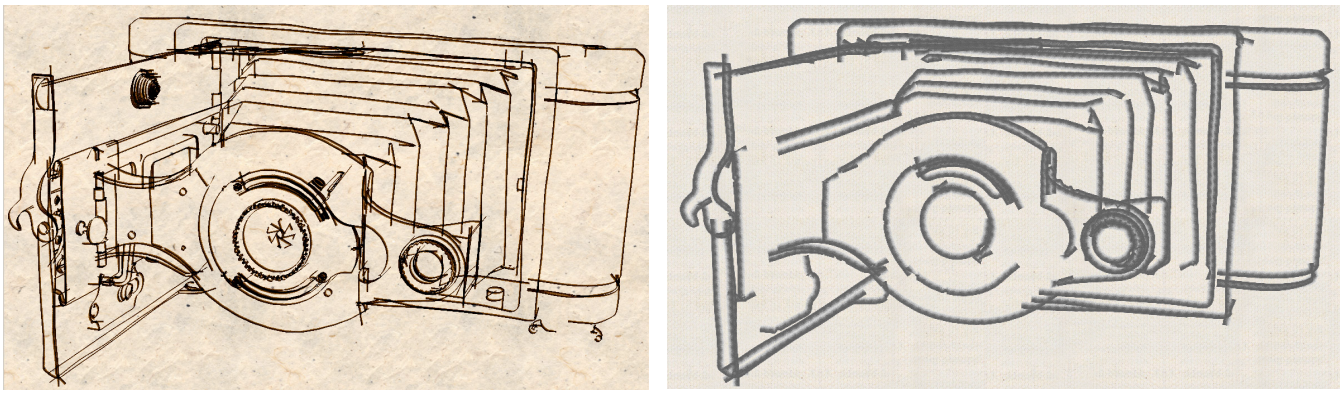

Figure 2 Two different styles applied to the same camera model

Use of Information to control attributes variation

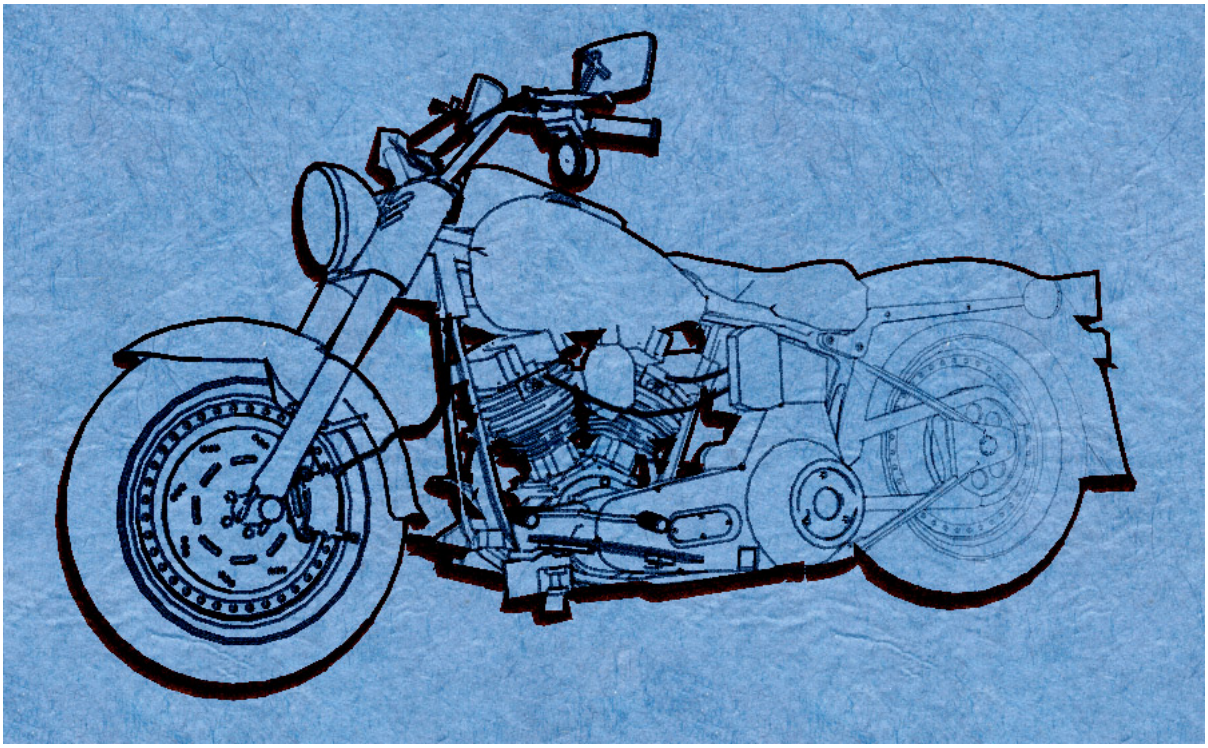

Figure 3 The lines thickness depends on the depth information. In addition, the external contour is drawn in a calligraphic manner.

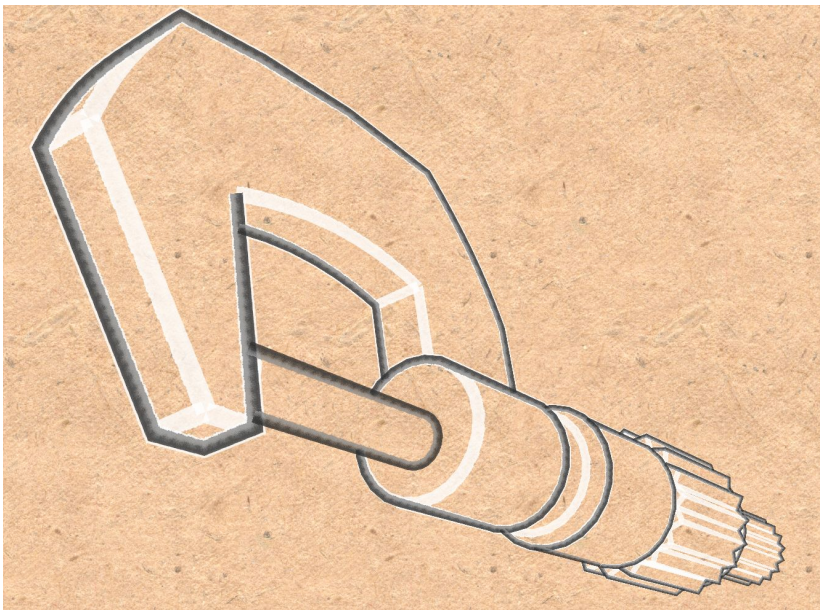

Figure 4 As in the previous figure, lines thickness depends on depth information. We use edges nature information to draw crease lines in white and silhouette lines in black.

Figure 5 The lines color depends on the material color information in order to simulate a cartoon like appearance.

Gallery of Styles obtained using our approach

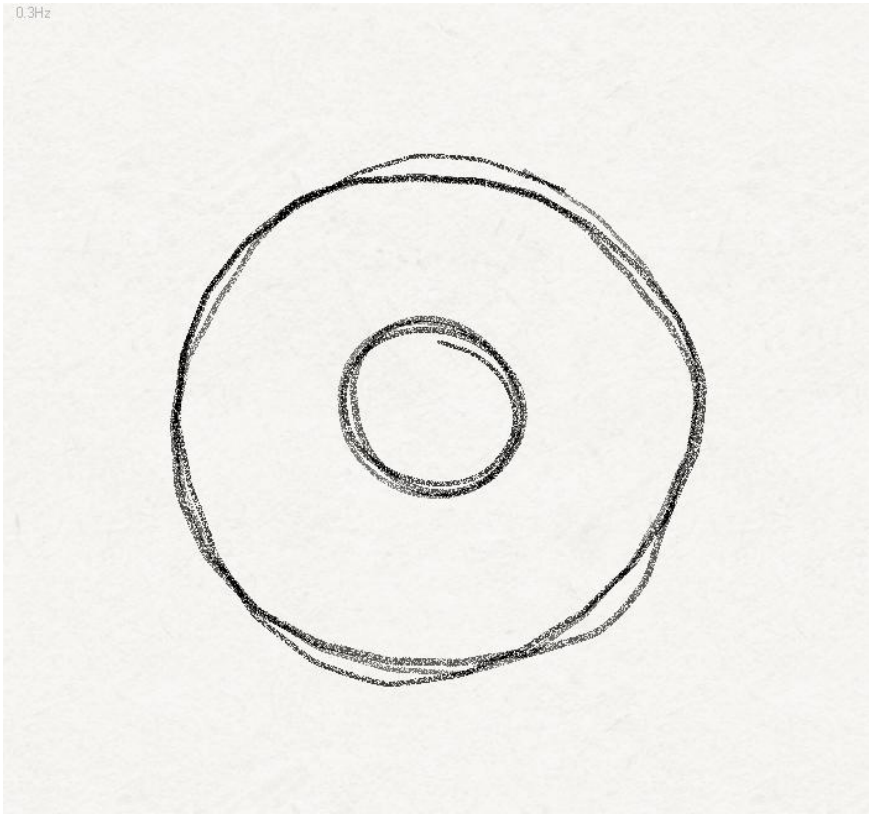

Figure 6 Sketchy look achieved by making the same stroke turn several times around the object.

Figure 7 The virgin example of the paper. It illustrates in particular the use of layering to produce complex drawings.

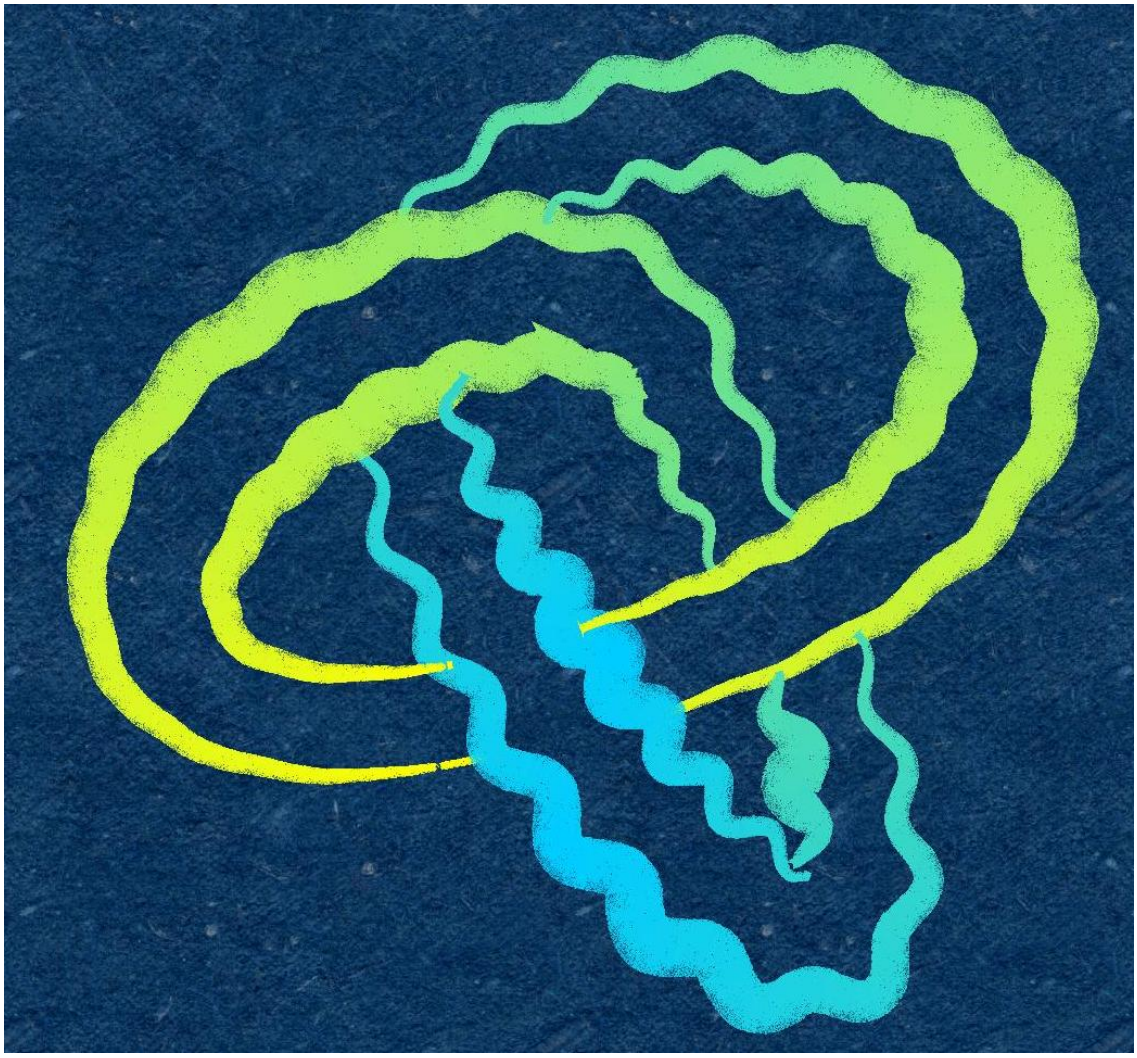

Figure 8 We illustrate here how the chaining operator, coupled with programmable shader allows to simulate multiple-parameterization for strokes. On this example, notice that the color variation relies on a parameterization covering the silhouette of the whole object, including occluded parts, to blend in a gradient way from yellow to blue and back to yellow. Similarly, the stroke's backbone geometry is displaced against a sinusoid defined on this same parameterization such as its amplitude linearly increases between 0 and 0.5, and then decreases back between 0.5 and 1. In contrast, the thickness variation is defined with respect to a parameterization that covers each visible segment of the silhouette, as increasing between 0 and 0.5 and then decreasing between 0.5 and 1.

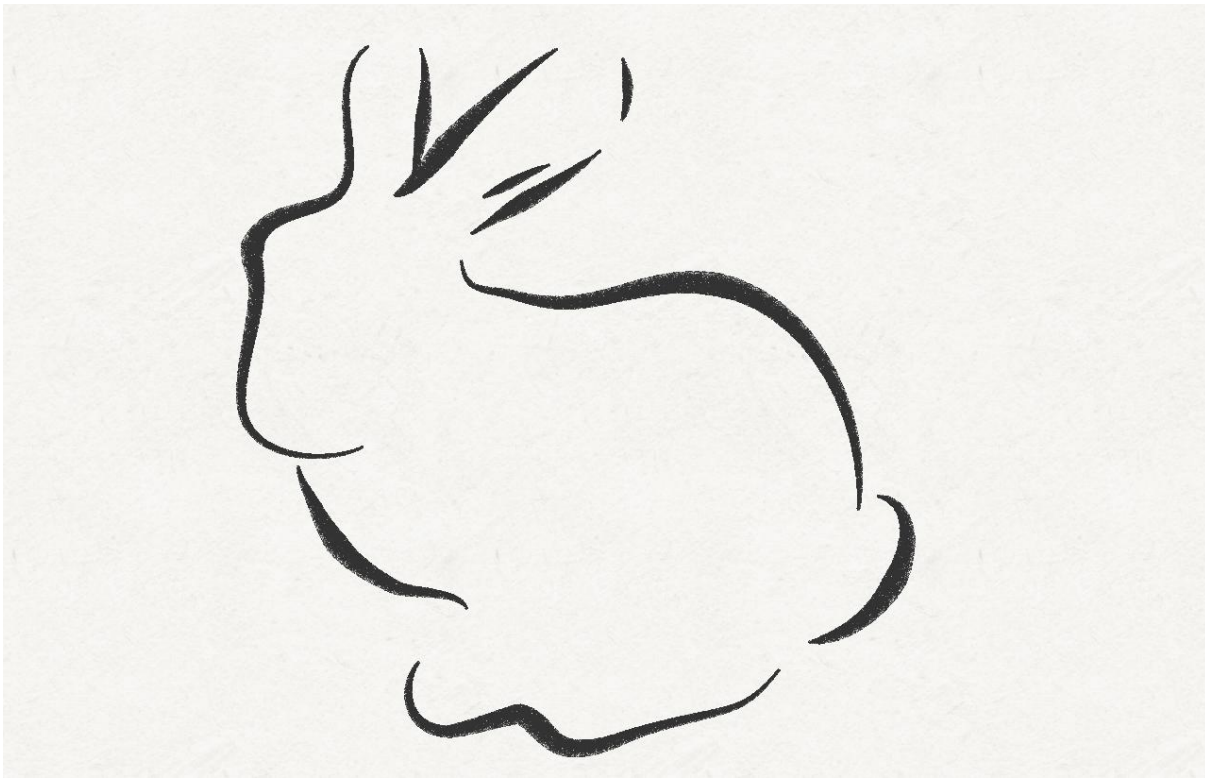

Figure 9 The Stanford bunny rendered in Japanese style.

Figure 10 Two models rendered using a style sheet made of two modules: the first one draws guiding lines and the second ones draws "finished" lines for close parts.

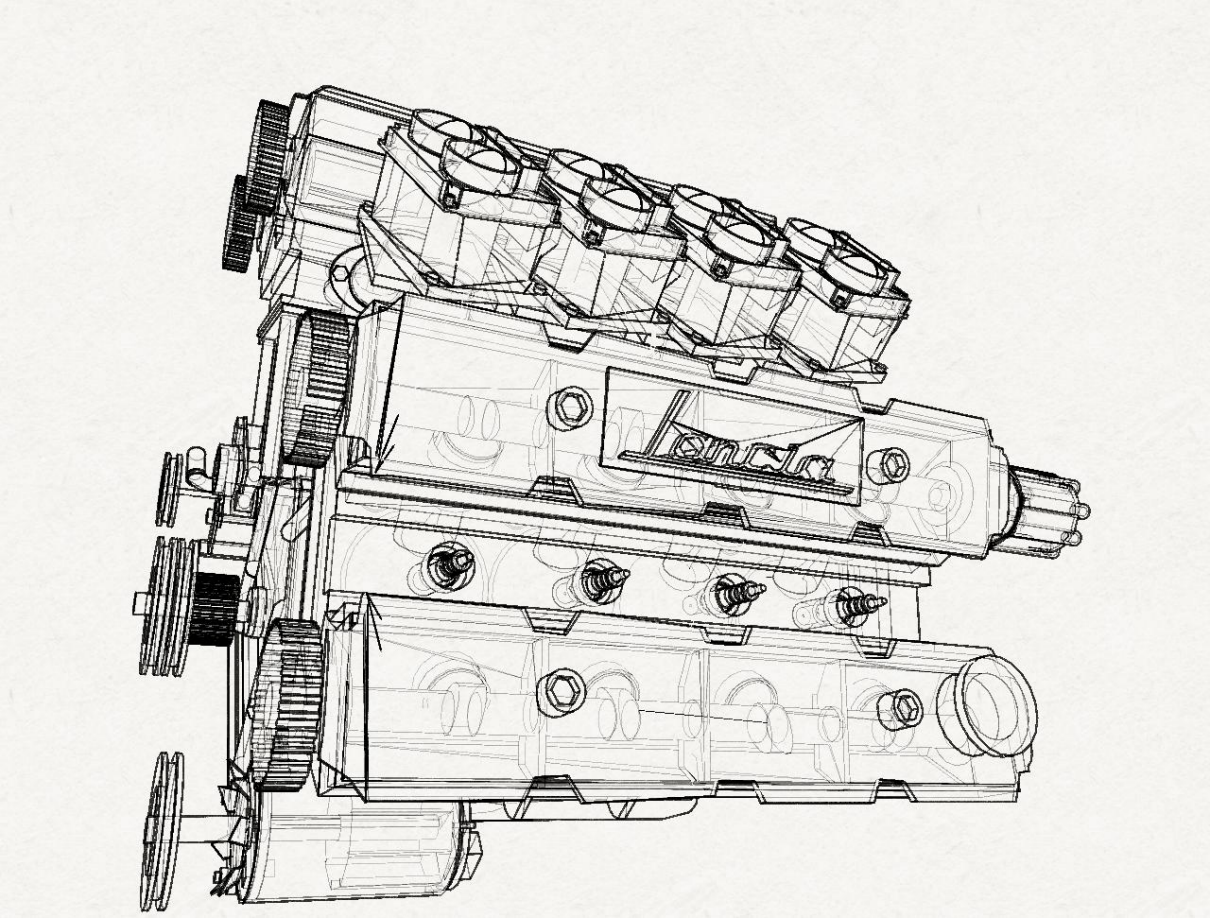

Figure 11 Technical rendering of a V5 engine, including lines that are hidden by only one object.