Using Logo to Model and Animate

Pavel Boytchev
Independent researcher
Sofia, Bulgaria
pavel@elica.net

Abstract

This paper presents special techniques for building 3D models. The main idea is based on having these models programmed entirely in Logo. Some of the models have been used in Logo-based 3D Computer Graphics course for high-school students with minimal programming knowledge. The presented methods demonstrate how students can be involved in the process of creating models and animations in an easy to understand way. The variety of models helps students to realise all the magic things they can do with Logo.

Keywords

Logo, 3D modelling, animation, Elica

1. Introduction

Logo is one of the most dynamic programming languages in the world. Throughout its history, Logo is generally considered as a language for children. This is partly due to it obvious advantage to have a low threshold. Unfortunately, the other advantage – the high ceiling – is not so well utilised.

Some teachers have noticed that many students start with Logo but later move to another programming language. There are many reasons for this: increased demands for performance, job requirements and utilisation of advanced technologies not available in mainstream Logos. Whatever the reason is, Logo still has a great potential.

This paper presents several techniques for building animated 3D models – one of the most requested feature from Logo. The unique contribution of this paper is to show how such models can be built by the high-school students and their teachers rather than having them preimplemented by Logo developers. Elica encourages programmatical experience instead of the easy point-and-click activities. That is why building these models requires gaining some experience with the system and cannot be done by absolute beginners.

The focus of the paper will be on the techniques and the tricks, rather than on the pedagogical issues. Most of the niceties in this paper have been inspired by students' questions. Finally, this paper is an attempt to support author's belief that **Logo is as strong as the combined power of all Logo dialects**.

2. Spying eyes

Eyes are considered one of the most expressive body part positioned in the most expressive area – the face. Naturally, it is very hard to reproduce eyes, so it was not a surprise when several students, attending a Computer Graphics course, asked for help in designing eyes of their 3D cartoon models. Of course, they could just use a small black circle in a bigger white

circle, but that was too simplistic. Their request had two aspects – how to model an eye as an object that can be reproduced where needed and when needed, and how to animate a pair of eyes.

Usually characters have two synchronously moving eyes. The orientation of the eyes is a clear indicator not only to which direction they look at, but also whether they are focused on a close-distance object or not. When people talk with other people, they can easily understand whether the other party is looking at them or is roving somewhere behind them.

Most often eyes are modelled easily by approximating them into spheres (Nagel D, 2001). Let's create our first definition of an eye:

```
to EYE :x :y
    make local "eye sphere (point :x :y 0) 60
    to ondrawimage end
end
```

Inputs `:x` and `:y` determine the screen position of the eye. The number 60 is the radius of the sphere and it affects eye's size. The following code uses the definition `EYE` and creates two independent eyes:

```
make "left_eye  EYE -100 0
make "right_eye EYE  100 0
```

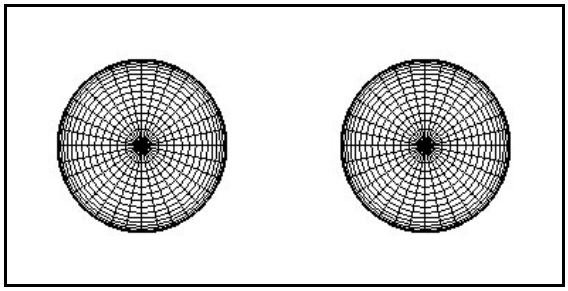

Figure 1. Two simple eyes

Although we have two eyes (see Figure 1), they do not look like real eyes. To fix this we can map a picture, also called *texture*, of an eye onto the surface of the eyeball. This technique is often used because when there are too many small details, it is faster to draw them as a picture, rather then to generate them as 3D objects (Sanders A, 2005).

It is easy to create such a picture by us ourselves without scanning a real eye. Even a low quality hand-made texture as the one shown in Figure 2 can produce adequate results. The black strip at the top is for the pupil, the blue one is for the iris, and the red threads on white background are for the tiny blood vessels.

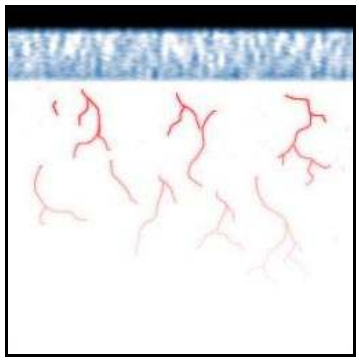

Figure 2. Eye texture

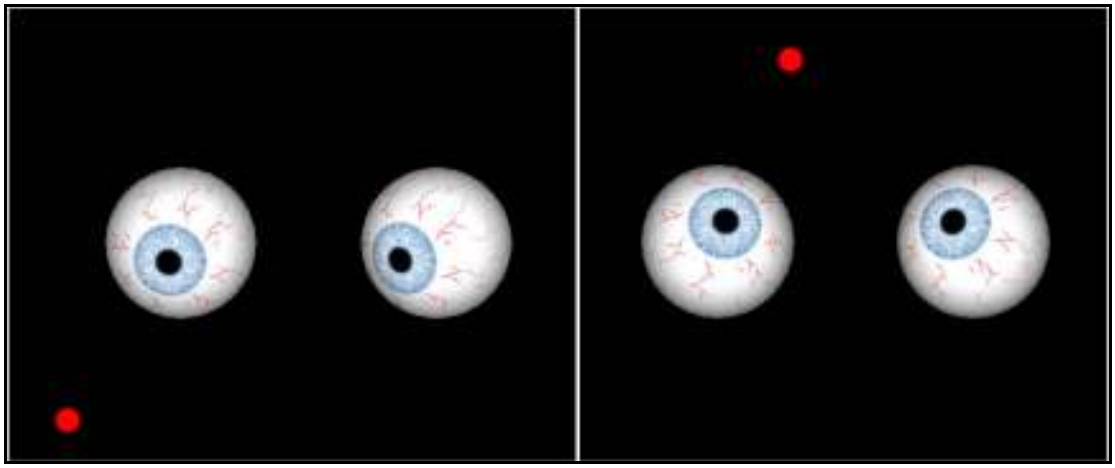

Figure 3. Textured eyes focused on a red point

When the texture is mapped onto the sphere, it starts to look like a real eye. If it is not possible to make a texture, there is an alternative method. Instead of a picture, we can create a smaller sphere inside a big one and then protract it in such a way that it comes slightly out of the sphere and looks like the iris.

What we still want from the eyes is to synchronously focus on one point and keep focused on it (Shillcock R, 2005) even if it moves – see Figure 3. But how to focus eyes on a target? Moreover, how to do this in real time?

Each graphical object has an eigen axis called `FOCUS`. Changing the orientation of this axis changes the orientation of the object too. We can control the object by changing its focus.

The traditional two-dimensional Turtle Graphics has a primitive called `TOWARDS`. It outputs the angle between the current heading of the turtle and the heading towards a selected point. In the 3D world one angle is not enough. We must use two angles or another way to determine directions. The easiest way is just to use the co-ordinates of the target. If we assume that the eye is at point `(X,Y,Z)`, and the target is at `(XX,YY,ZZ)`, then the direction is determined by the vector `(XX-X,YY-Y,ZZ-Z)`. The eigen axis is uniquely determined by this vector and we are now ready to write the eye's `TOWARDS` command:

```
to towards :xx :yy
   make "eye.focus vector :xx-:x :yy-:y 500
end
```

The new `TOWARDS` command must be placed inside the definition of `EYE`. It sets `FOCUS` with the value of the vector from the centre of the eye to point `(XX,YY,500)`. To make code simpler, we assume that the target moves in the plane ZZ=500, and the eyes are in the plane Z=0.

To test the eyes we may add one more command, which is activated whenever the mouse pointer is moved. This command captures the screen co-ordinates of the mouse and calls the `TOWARDS` command of each eye.

```
to onmousemove :x :y :xp :yp :stat
   left_eye.towards :x :y
   right_eye.towards :x :y
   resumepaint
end
```

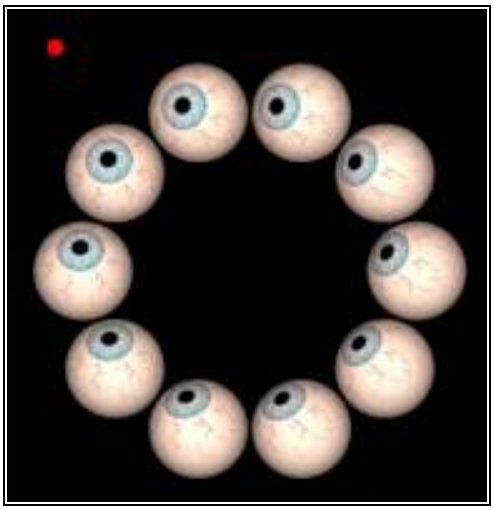

Figure 4. A ring of tired eyes

By default `onmousemove` has five inputs: `(x,y)` is the logical coordinates of the mouse cursor, `(xp,yp)` is the physical window coordinates in pixels, and `stat` contains button status flags.

The method, presented here, is not limited to two eyes only. We may have many eyes focused on a target as shown in Figure 4, or several groups of eyes each is focused on its own target.

3. Water waves

Author's experience with students show that they are often interested in animating objects. However, it is difficult for them to judge which animation is easy to implement and which is not.

Let's take one such example – animating water (Ervin S and Hasbrouck H, 2001). Is it hard or easy? If you ask students, they will most probably say that they do not know. If you ask scientists, the answer will be that it is hard and resource consuming (Teining V, 2005). You need a super computer, lot of time and a solid mathematical background.

But on the other side, when students make animation, they do not need scientifically correct motion. They are happy if they achieve almost-realistic behaviour. So, instead of trying to figure out the answer let's just do the animation.

Figure 5 presents one possible method for generating pseudo-waves (Virtual Terrain Project, 2005). We start with a grid of points and connect them with segments. When we oscillate each point up and down, we hope that segments will move like waves. The small arrow in the second image in Figure 5 shows how one particular point is offset from its neutral position. The third image is a surface with increased number of interpolated points and textured with an image.

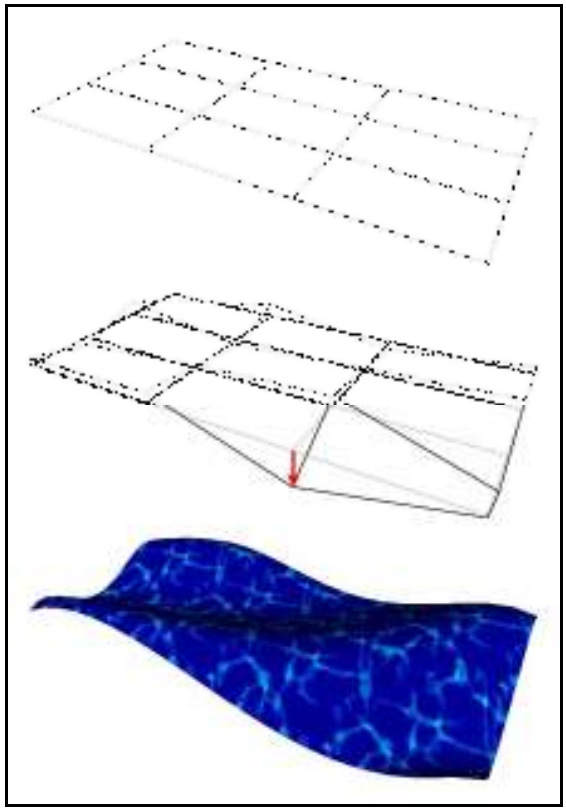

Figure 5. Turning a flat grid into water waves

There is one nicety here. If we randomly move points, we will get unwavy boiling water. To achieve the feeling of waves each point should oscillate as if *"sailing"* on the graph of a sine function:

```
make "z :k*sin (20*:time+:f)
```

The code above shows a way to move a single point. Variable `:k` is the size of the wave, `:time` controls the animation timing, and `:f` is an offset (for each point there is a constant offset). The existence of this offset is very important. It is used to stir

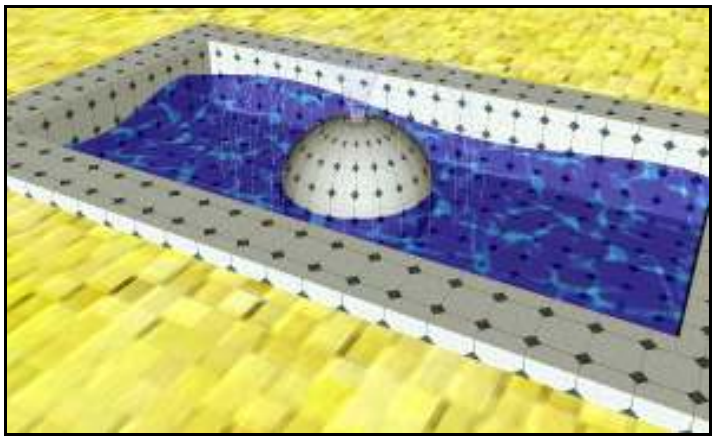

Figure 6. Water pool and fountain

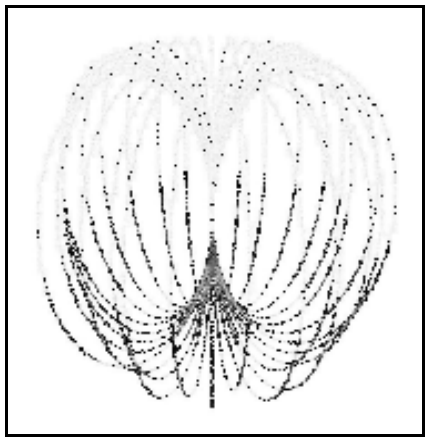

Figure 7. A fountain made of ellipses

points synchronously. While a point is going up, the next point may be going down. The offset can be defined in many ways but it must depend on the point to which it is bound. Figure 5 uses offsets for circular waves:

```
make "f 90*(1+sqrt :x*:x+:y*:y)
```

However, we may also use linear ones:

```
make "f 90*(1+:x+:y)
```

Let's use waves in a complex animated 3D model. Figure 6 is a snapshot of a pool with water waves and a fountain. The fountain is implemented as a ring of ellipses, which lines are in stipple mode – Figure 7. Instead of processing hundreds of points, it is enough just to continuously shift the pattern in one direction.

Of course, using an ellipse to simulate the parabolic flow of water is not scientifically correct, but visually it looks acceptable and is fast enough.

For many reasons it is preferred to define the fountain as a standalone object that can be reused and animated in other programs with minimal additional efforts. The following code shows how a fountain object might be defined:

```
to fountain
   local "mode "pattern "stream "a "rad
   make "mode 1
   make "pattern "oxoxooxoxooxoxx [autoredraw]
   make "stream set

   repeat 30 [
      make "a 12*(repeat)
      make "rad 3.5+random 1
      make "stream(repeat) custom (ellipse
        (point :rad*(cos :a) :rad*(sin :a) -1) 8 :rad) (set
         "focus point (cos :a+90) (sin :a+90)  0
         "spin :a+90 )
    ]

   to pour
      make "pattern word bf :pattern first :pattern
   end

   to ondrawimage end
end
```

When it is used, this definition of `FOUNTAIN` creates a fountain object with local array `STREAM` containing all ellipses (they are called `STREAM(1)`, `STREAM(2)`, etc). There is also a local command `POUR` that changes the pattern by moving the first character at the end. The pattern is defined in a local variable called `PATTERN` that has an attribute `AUTOREDRAW`. This means that any change in the pattern will request automatic redraw of the object.

To test the object we may write:

```
make "MyFountain Fountain
repeat 1000 [ MyFountain.pour ]
```

If we need to place several fountains, it would be good to change the definition by adding parameters for the co-ordinates of the object and use these co-ordinates in the definition of individual `STREAM`-s. Alternatively we may create all fountains at (0,0,0) and then translate their images to the desired locations.

Ghost stories

Whenever a student makes a 3D model, it is important to know that even the most complex one can be decomposed into a bunch of simple tricks. A special effects guru is a person who can mentally construct and play with many chains of effects in order to find the most suitable ones.

To illustrate this concept we will try to build a ghost with the following properties – *appearance*: the ghost has the traditional bed-sheet appearance, a head closely bound up with the hands, a semitransparent body, weakly fixed eyes; *behaviour*: a continuous floating in the air and rolling eyes, from time to time hands are moving to various positions.

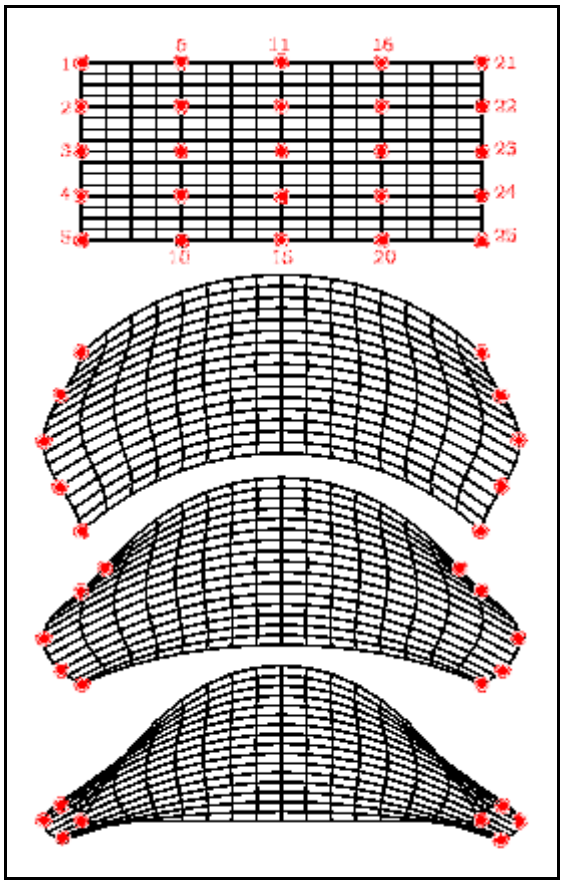

Figure 8. Making the hands of the ghost

Let's start with the body of the ghost. We will apply the same trick as with water waves. The body will be a grid controlled by 25 basic points – see the top image in Figure 8. They are marked as red dots numbered from #1 to #25.

Then we gradually curl the grid so that the left five points #1, #2, #3, #4 and #5 form the left hand (from user's point of view); and points from #21 up to #25 are used for the right hand.

The head is shaped like the hands. Point #13 lifts up and becomes the top of the head. Point #11 falls down towards point #15 – see Figure 9. They form the front and the back of ghost's body.

After applying transparent colouring, we get the final form of the ghost's body – the bottom image in Figure 9.

To complete the ghost we only need to add eyes. Because they will be small, we do not need to supply all the details as we did in the beginning of the paper.

Instead we can use another trick – each eye will be a slice of a sphere. The bottom of the sphere and a small portion of top will be removed as shown in Figure 10. Because the bottom is missing and the background is black, the hole at the top will look like eye's iris:

```
make "eye1 custom sphere (point -0.1 0 2) 0.1 (set "rangev range 20 90)
```

The size of the iris is controlled by the RANGEV property. Vertically a sphere spans from 0^o (top) to 180^o (bottom). A range from 20^o to 90^o will cut the slice in Figure 10.

Now we are ready with all body parts – a cephalopodal torso with head and hands, and a pair of eyes. To animate the ghost we may implement several rudimentary movements: overall body roaming, hands movement and eye rolling.

The cyclic nature of roaming and rolling means that we may use trigonometric functions to simulate them. For example, we can control eyes by simply changing their focus:

$$focus\ axis = \left(\frac{1}{2}cos\ \alpha,\ 1+\frac{1}{2}sin\ \alpha,\ -\frac{1}{5}-\frac{1}{5}sin\frac{\alpha}{2}\right)$$

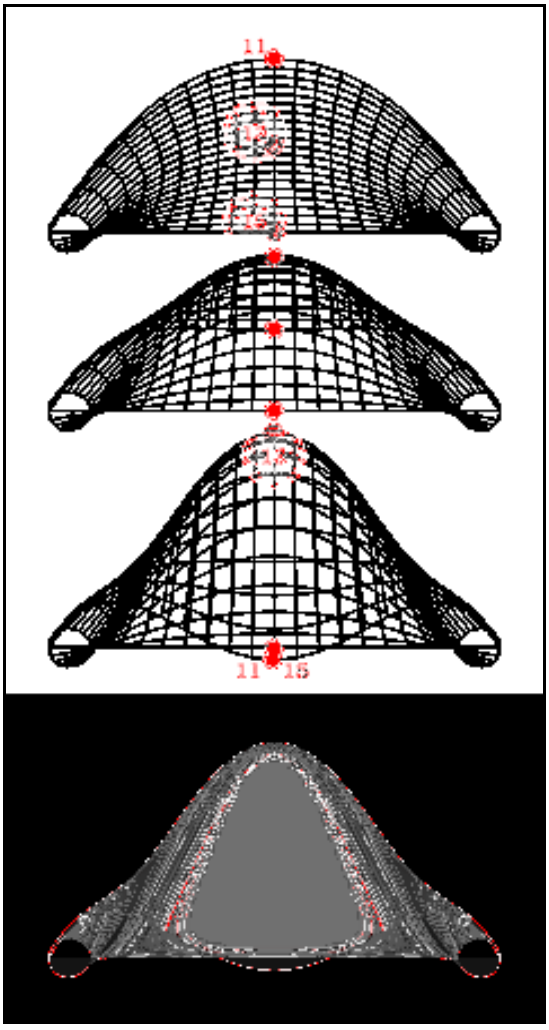

Figure 9. Making the head of the ghost

Ghost body can be controlled by changing the co-ordinates of only three points using these formulas:

$$point_{11} = \left(sin\ 3\alpha, 0, -5+sin\ \alpha\right)$$

$$point_{13} = \left(sin\ 3\alpha, 0, 11+sin\ \alpha\right)$$

$$point_{15} = \left(cos\ 3\alpha, 0, cos\ \alpha\right)$$

These formulas look complex and one may ask how to find them. The answer is that it is easy to generate such formulas – just mix sine and cosine with various offsets, frequencies and amplitudes. It is important if all points have slightly different formulas, otherwise whey will move together and the ghost will look like solid structure

To finalise all movements we will use the last trick: hands are defined by 10 points. It is possible to control individually all of them, but this will make animations slower and more complex. Figure 9 shows that for each hand two of the points coincide, so there are 4 distinct points which are placed uniformly around an imaginary centre. Therefore, instead

of moving 10 points we can calculate the position of only two imaginary ones – the two centers – and all other points can be set relatively to them.

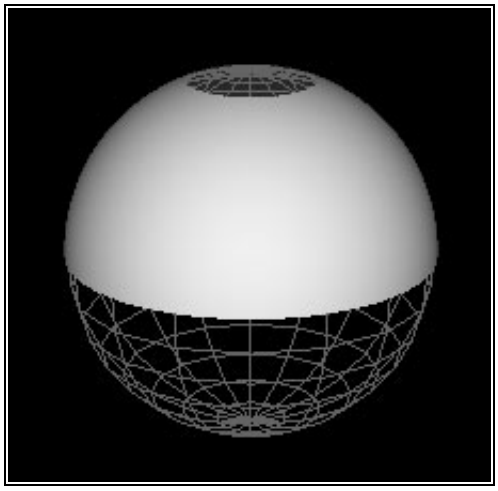

Figure 10. Ghost's eye

We are now ready with the ghost. A complete Elica program that builds it and animates it is provided at the end of this section. The ghost has a body and moves in real time.

The last Figure 11 shows a collection of the ghost in different postures. By changing hands' positions, we can achieve a variety of actions. While playing with the animated ghost, it is even possible to sense its mood – sometimes it looks bored, other times it is curious. All this is achieved only with the three rudimentary movements. Advanced Logo users may add more distinguishable facial expressions, but this is another topic.

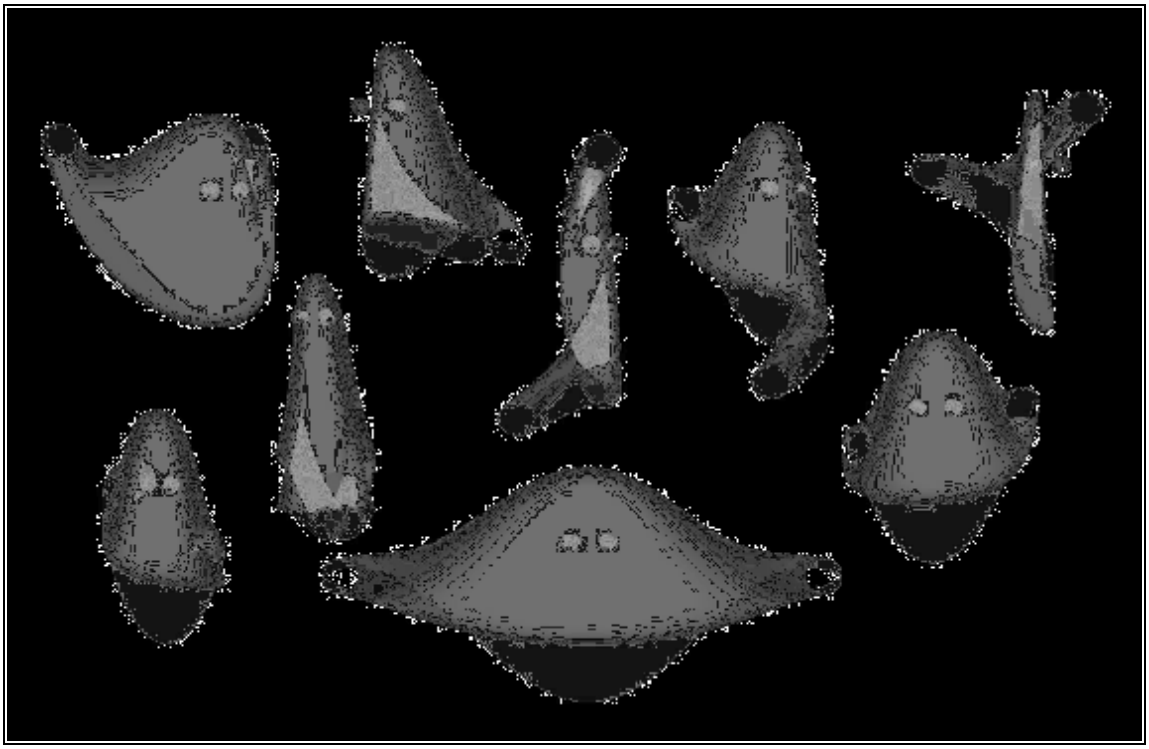

Figure 11. Ghost fashion show

```
; the following program code is a complete Elica Logo program that
; creates and animates a ghost – see Figure 11 for some examples
run "graphix

; global properties of the 3D scene and all 3D objects
make "onbeforedraw.back background rgb 0 0 0
make "onbeforedraw.proj orthogonal
setmatrix [[1 0 0 0] [0 1 0 0] [0 0 1 0] [0 0 0 0.01]]
lookat vector 0 6 3 vector 0 0 0 vector 0 0 1

make "light "true
make "mode 2
make "color (rgb 255 255 255 100)
make "smooth "true

; ghost's eyes are made from small spheres
make "eye1 custom sphere (point -0.1 0 2) 0.1 (set "rangev range 20 90)
make "eye2 custom sphere (point  0.1 0 2) 0.1 (set "rangev range 20 90)

; this is the definition of the ghost body – a 5x5 grid of uniformly spread points,
; most of these points will be modified during the animation
make "ghost nurbs (set
    point  2 -2 0 point  2 -1 0 point  2 0 0 point  2 1 0 point  2 2 0
    point  1 -2 0 point  1 -1 0 point  1 0 0 point  1 1 0 point  1 2 0
    point  0 -2 0 point  0 -1 0 point  0 0 0 point  0 1 0 point  0 2 0
    point -1 -2 0 point -1 -1 0 point -1 0 0 point -1 1 0 point -1 2 0
    point -2 -2 0 point -2 -1 0 point -2 0 0 point -2 1 0 point -2 2 0 )

make "a 0
while "true
[   suspendpaint
    ; precalculates sine and cosine of various angles, the increment
    ; of variable a determines the speed of ghost movements
    make "a :a+2
    make "s1 sin :a   make "c1 cos :a
    make "s2 sin :a/2 make "c2 cos :a/2
    make "s3 sin :a/3 make "c3 cos :a/3

    ; controls the head and the torso of the ghost
    make "ghost.points.#11 point :s3 0 :s1-5
    make "ghost.points.#13 point :s3 0 :s1+11
    make "ghost.points.#15 point :c3 0 :c1

    ; processes the left hand – the 1st and the 5th points are the
    ; same, this is where sleeve edges wrap around ghost's wrist
    make "ghost.points.#1  point :s1+1 0 :c3
    make "ghost.points.#2  point :s1+1 -1 :c3
    make "ghost.points.#3  point :s1 0 :c3
    make "ghost.points.#4  point :s1+1 1 :c3
    make "ghost.points.#5  :ghost.points.#1

    ; this code controls the other hand
    make "ghost.points.#21 point -1-:s2 0 :c1
    make "ghost.points.#22 point -1-:s2 -1 :c1
    make "ghost.points.#23 point 0-:s2 0 :c1
    make "ghost.points.#24 point -1-:s2 1 :c1
    make "ghost.points.#25 :ghost.points.#21

    ; synchronously hover and focus ghost's eyes
    make "eye1.center point :s3/3-0.2 5 3+:s1/4
    make "eye2.center point :s3/3+0.2 5 3+:s1/4
    make "eye1.focus point :c1/3 1+:s1/5 -0.2-:s2/5
    make "eye2.focus :eye1.focus
    resumepaint
]
```

4. Some final words

Logo is as strong as the combined power of all Logo dialects. Each dialect adds something unique to the power of Logo and this makes it look like a kaleidoscope of potentialities. That diversity is something that is simply impossible with the majority of other programming languages.

The variety of properties, functionality and goals makes Logo suitable for any age and any level of experience – it can be used by young children, but it can be used by high school students and scientists too.

The current paper is a very brief introspection of just one particular Logo dialect – Elica Logo. Its audience is generally users who have increased requirements from Logo. Although fundamental, asking the turtle to make circles is not an activity that can grab attention for years. While users work with one specific Logo implementation they gather knowledge and experience, but at some point, they start to feel squeezed for room. Instead of switching to another language, they can just move to another Logo dialect. This is maybe one of the best benefits of Logo – to provide a lifelong learning environment.

References

Ervin S and Hasbrouck H (2001*), Landscape Modeling - Digital Techniques for Landscape Visualization*, McGraw-Hill, 2001, ISBN: 0-07-135745-9, <http://www.landscapemodeling.org/>

Nagel D (2001), *The Eyes Have It – Modeling the human eye in Amorphium Pro*, <http://www.creativemac.com/2001/06_jun/tutorials/amorphiumeye/amorphiumeye-page1.htm>

Sanders A (2005), *From 2D to 3D: Using Textures to Create Raised Surfaces*, <http://animation.about.com/od/moviemagic/a/texturebump.htm>

Shillcock R (2005), *The Computational Modelling of Eye Movements: Interhemispheric, Syntactic and Semantic Influences*, <http://www.iccs.inf.ed.ac.uk/~smcdonal/Wellcome/project_home.html>

Teining V (2005), *Mathematical/Numerical Modelling of Water Waves*, <http://www.mek.dtu.dk/English/Research/Feature_Articles/Mathematical_Numerical%20_Modelling_of_Water_Waves.aspx>

Virtual Terrain Project (2005), *Water Rendering and Simulation*, <http://www.vterrain.org/Water/>