Curve Language Reference Manual

Kun An
John Chan
Dave Mauskop
Wisdom Omuya
Zitong Wang

October 27, 2012

Contents

1 Introduction

Curve is a vector graphics manipulation language specifically targeted for graphics processing. It is a text-based graphics language that allows for rendering of images, both static and moving.

The goal of the Curve syntax is to make conceptually simple graphics and manipulations using simple, yet powerful, language constructs. The type system in Curve is described in Section 4.2. The basic type, Curve, while simple, is very expressive. e.g. a group of curves may be used to represent a layer, a group of layers used to form an image. This language reference manual describes the syntax of Curve.

2 Program Definition

A Curve program consists of a sequence of zero or more expressions.

3 Lexical Conventions

3.1 Comments

In-line comments are preceded by `//`, while block comments are delimited by `/*` and `*/`. Nesting will not be allowed.

3.2 Identifiers

Identifiers are comprised of uppercase letters, lowercase letters, digits and underscore (`_`). The first character cannot be a digit, nor can it be an underscore (`_`).

3.3 Keywords

The following keywords are reserved:

```
Integer                    isControl
Double                     get
String                     set
Layer                      insert
Curve                      while
Point                      break
fun                        continue
if                         switch
elif                       start
el                         true
for                        false
getx                       draw
gety                       print
```

3.4 Constants

3.4.1 Integer constants

Integers are represented by a combination of digits and nothing else. Only decimal representations are allowed. Example:

```
//Declaration and assignment of an Integer
Integer m = 99;
```

3.4.2 Double constants

Doubles must have one and only one decimal point, with digits on either side and must have at least one digit. Only decimal representations are allowed. For example, these are all Doubles:

```
Double a = 0.;
Double b = .13;
Double c = 26.40;
```

3.4.3 Boolean constants

Booleans can either be `true` or `false` (case sensitive).

3.4.4 String constants

Strings are combinations of characters, delimited by either single quotes or double quotes. These escaped sequences are also allowed:

Sequence	Definition
\n	New Line
\'	Single Quote
\"	Double Quote
\\	Backslash

3.5 Operators

Operator	Definition
+	Addition
-	Subtraction
*	Multiplication
/	Division
!	Logical Negation
>>	Translation - moves an object to a new relative location
=	Assignment

3.6 Punctuators

The following symbols are used to organize code and to specify different organizations of objects:

Symbol	Definition
;	Marks the end of a statement
[]	Marks the beginning and end of a group of Curves - i.e. a Layer
{}	Marks the beginning and end of a group of statements
'	Marks the beginning and end of a control point in a Curve
,	Separates different dimension values of a point
:	Used in for loops - specifies a range - i.e. 1:4 means 1 through 4 inclusive
.	Used for dot notation access - i.e. accessing points of a curve
()	Used for grouping parameters in function call

4 Object Types

An Curve object is a manipulatable region of storage. There are two broad classes of objects supported in Curve.

4.1 Basic Types

There are four basic types defined by the Curve language. Type identifiers always begin with an upper-case letter followed by a sequence of one or more legal identifier characters. The built-in types include:

- `Point`: A pair of `Double`s representing Cartesian coordinates
- `Curve`: A list of `Point`s
- `Layer`: A list of `Curve`s

4.2 Atom Types

Legal atom-types are as follows:

- `Integer`: An `Integer` object may be used anywhere an integer may be used. See section 5 for a discussion of operations possible for `Integer`s.
- `Double`. The `Double` type is used for all floating point calculations in Curve. See section 5 for a discussion of operations possible for `Double`s.
- `Boolean`: The `Boolean` type can take of one of two possible values – `true` or `false`.
- `String`: A `String` is a sequence of characters surrounded by double quotes " " .
- `Array`: An `Array` is a contiguous region of storage for any zero or more of any given object type in Curve.

4.3 Object Scope

All identifiers in Curve are global. All existing objects in the namespace will be displayed on calling the draw() function.

5 Expressions and Operations

In this section we describe the built-in operators for Curve and define what constitutes an expression in our language. Operators are listed in order of precedence. All operators associate left to right, except for assignment, which associates right to left.

1. Primary expressions
   (a) identifier
   See section 3.2.
   (b) constant
   See section 3.4.
   (c) (expression)
   (d) primary-expression [expression]
   Index into a list.
   (e) primary-expression (list of 0 or more expressions, comma separated)
   Call a function with optional arguments. See section 8 for a description of the built-in functions getX, getY, isControl, and draw.
2. Unary operators
   (a) !expression
   Performs negation on the expression, which must be boolean.
3. Multiplicative operators
   (a) expression * expression
   Multiplication is valid between two integers, two doubles, an integer and a double, or a Point and an integer or double. If one of the expressions is an integer and the other is a double, the result is a double. If one of the expressions is a Point, its x and y values are each multiplied by the other expression.
   (b) expression / expression
   Division is defined identically to multiplication, except of course that division by zero is not allowed.
   (c) expression mod expression
   Gives the remainder from expression / expression, is only valid for two integers.

(d) expression >> (double, double, double, double)
Performs multiplication by a 2x2 translation matrix. The expression must be a Point or a list of Points which are each regarded as 2x1 matrices. The 2x2 matrix of doubles is given in left-to-right, top-to-bottom order.

4. Additive operators

(a) expression + expression
Addition is valid between two integers, two doubles, a double and an integer, or two Points. In the case of addition of a double and an integer, the result is an integer. When adding two points (x1, y1) + (x2, y2), the result is (x1+x2, y1 + y2).

(b) expression - expression
Subtraction is defined identically to addition.

5. Relational operators

(a) expression < expression
Less than.

(b) expression > expression
Greater than.

(c) expression <= expression
Less than or equal to.

(d) expression >= expression
Greater than or equal to.

The relational operators are valid for comparison between two integers, two doubles, or a double and an integer. The result is a boolean.

6. Equality operators

(a) expression == expression
Equal to.

(b) expression ! = expression
Not equal to.

The equality operators are valid for comparison between two integers, two doubles, a double and an integer, or two Points. The result is a boolean. When comparing the equality of two points, the result is true is the x and y values of the two points are identical.

7. Logical operators

(a) expression & expression
Logical AND between two boolean expressions.

(b) expression | expression
Logical OR between two boolean expressions.

8. Assignment operators

(a) primary-expression = expression
The value of the expression replaces the value of the object that the primary-expression refers to. Types must match.

6 Statements

6.1 Statements

All statements in Curve end in a semi-colon. Statements must either declare a object/variable, modify an existing object/variable, execute some calculation using some constant(s) or previously declared variable(s), or render some Curve(s) on screen.

7 Declarations

7.1 Function Declaration

Our language supports user-defined functions. All functions must be preceded with the fun keyword. Functions are declared and implemented at the same time, which means before using an user-defined function user must first implement it. The return type of a function needs to be declared. A function can have any number of parameters. The parameters are passed by VALUE. And the types of parameters also need to be declared clearly. Here is an example:

```
fun Double dist (Double x1, Double y1, Double x2, Double y2) {
  return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}
```

7.2 Variable Declaration

Users need to declare the type of variable

```
Integer:
    Integer a = 10;
Double:
    Double a = 10;
    Double a = 10.;
    Double a = .8;
    Double a = 4.7;
Boolean:
    Boolean b = true;
String:
```

```
    String s = "Hello";
Point:
    Point p = (0,1);
    Point p = (0,1,true); //true means the point is a control point
Curve:
    Curve c = (0,1)(1,0);
    Curve c = (0,0)'(1,1)(2,2)(3,1)'(4,0);
    Curve c = [p1, p2]; //p1, p2 are points
Layer:
    Layer l = [c1, c2]; //c1,c2 are curves
    Layer l = c::l; //l is layer and c is curve
Array:
    Integer a[5] = {0, 1, 2, 3, 4};
    Integer a[100] = 0; //assign 0 to all the elements of a
```

8 System Functions

8.1 Draw Function

Draw function can be used to draw a frame to output with some controls.

`draw()`:

Draw all the objects (curve, layer) existing in the namespace. The time of current frame is set to default.

`draw(Object a1, Object a2, ...)`:

draw each object according to the order of parameters, which means a2 will be drawn after a1. The object can be Curve or Layer.

8.2 Math Functions

Our language supports basic math functions:
`Double sqrt(Double)`: return the square of the parameter.
`Double sin(Double)`: return the sine value of the parameter. The unit of the parameter is radian.
`Double cos(Double)`: return the cosine value of the parameter.
`Double tan(Double)`: return the tangent value of the parameter.
`Double abs(Double)`: return the absolute value of the parameter.

8.3 Functions for Point

Our language has some built-in functions for type Point to retrieve or change attribute of Point.

```
Double getx();
Double gety();
Double x = p.getx(); //p is a Point
```

Return the x or y coodinate of a Point.

```
Boolean setx(Double);
Boolean sety(Double);
p.setx(3.5); //p is a Point
```

Set the x or y coodinate of a Point. The functions will return `true` if they run successfully.

```
Boolean isControl();
Boolean b = p.isControl();
```

Return a boolean value to indicate whether the point is a control point. If the return is `true`, then it's a control point.

8.4 Functions for Curve

Our language has some built-in functions for type Curve to retrieve information of Curve.

```
Point get(Integer);
Point p = c.get(0);
```

The get(n) function will return the `n`th point of a curve. If n is greater or equal to the number of points that compose the curve, the function will give an error.

```
Boolean insert(Integer index, Point);
c.insert(2, p); //insert point p after the 3rd point of curve c
```

The insert(n, p) function will insert a Point p after the (n+1) th point. If the insertion is successful, it will return `true`.

```
Boolean delete(Integer index);
c.delete(1); //delete the second point of curve c
```

The delete(n) function will delete the (n+1) th point from curve c. The start point and end point cannot be deleted. The function will return `true` if the deletion is successful.

```
Boolean set(Integer index, Point);
c.set(1, p); //set the 2nd point of c to point p
```

The set(n, p) function will set the (n+1)th point of curve c to p. If the setting is successful, the function will return `true`.

```
Integer size();
c.size();
```

The size() function will return the number of points of a curve.

9 Examples

9.1 First Thing First

```
// Let's draw a circle first.
// Drawing a circle in Bezier way is a four-step process.
// Each step, we draw a quarter of the circle.

Layer circle = [];

Curve topRight = (0, 1)'(0.552, 1)(1, 0.552)'(1, 0);
circle = topRight :: circle;

Curve bottomRight = topRight >> (0, 1, -1, 0);
circle = bottomRight :: circle;

Curve bottomLeft = bottomRight >> (0, 1, -1, 0);
circle = bottmLeft :: circle;

Curve topLeft = bottomLeft >> (0, 1, -1, 0);
circle = topLeft :: circle;

draw(circle);
```

9.2 Do It the Other Way

```
// Though we could specify how to draw a circle segment step by step,
// there is an alternative way for doing this.
// Let's utilize functions and control statements.

// Let's write a function first that rotate a curve clockwise
// around origin by given degrees.
fun Curve rotate(Curve original, Double deg) {
  Double rad = deg * 3.1416 / 180;
  Curve result = original >> (cos(rad), sin(rad), -sin(rad), cos(rad));
  return result;
}


// Let's start to draw a circle using the function we wrote above.
Layer circle = [];
Curve topRight = (0, 1)'(0.552, 1)(1, 0.552)'(1, 0);
Curve nextCurve = topRight;

for (Integer i = 0 : 3) {
```

```
  nextCurve = rotate(nextCurve, 90);
  circle = nextCurve :: circle;
}

draw(circle);
```

9.3 Make It Move

```
// Let's first assume that we have a function called getCircle that
// encapsulates the code in previous example and returns a circle.
// We will use this function directly to avoid verbosity.

Layer circle = getCircle();

// Let's make the circle move acrose the screen along a sine wave.

for (Integer i = 1 : 301) {

  Double pi = 3.1416;

  // For each frame, we define the extent to which the circle moves.
  Double deltaX = 2;
  Double deltaY = sin(2 * pi * i / 300) - sin(2 * pi * (i - 1) / 300);

  circle = circle >> (deltaX, deltaY);
  draw(circle);
}

// The circle would end up being at position (0, 600).
```

9.4 Plan the Path

```
// We could plan the path along which our circle moves.
// For a basic example, we specify that the circle will move
// within an 800 * 600 rectangle.
// Whenever it comes to the edge, it will make a 90 degree turn,
// bouncing off the edge.

Layer circle = getCircle();

// Let the circle start from (400, 300), moving along vector (1, 1).
circle = circle >> (400, 300);
Point vector = (1, 1);
```

```
while (true) {
  if (circleAtEdge(circle)) {
    vector = vector >> (0, 1, -1, 0);
  }
  circle = circle >> vector;
  draw(circle);
}

fun Boolean circleAtEdge(Layer circle) {
  Boolean atEdge = false;
  for (Curve segment : circle) {
    Integer x = segment.getX(0);
    Integer y = segment.getY(0);
    if (x == 800 | x == 0 | y == 600 | y == 0) {
      atEdge = true;
      break;
    }
  }
  return atEdge;
}
```