Software & Tools

Asymptote: A vector graphics language

John C. Bowman and Andy Hammerlindl

Abstract

Asymptote is a powerful descriptive vector graphics language inspired by METAPOST that features robust floating-point numerics, automatic picture sizing, native three-dimensional graphics, and a C++/Java-like syntax enhanced with high-order functions.

1 Motivation

The descriptive vector graphics language Asymptote[1] was developed to provide a standard for drawing mathematical figures, just as TEX and LATEX have become the standard for typesetting equations in the mathematics, physics, and computer science communities. Asymptote has been aptly described as "the ruler and compass of typesetting" [1]. For professional quality and portability, Asymptote natively produces PostScript or PDF output. Graphics labels are typeset directly by TEX to achieve overall document consistency: identical fonts and equations should be used in graphics and text portions of a document.

In this article we first highlight Asymptote's basic graphics capabilities with an example and then proceed to review the origins and distinguishing features of this powerful vector graphics language. Further examples of Asymptote diagrams, graphs, and animations are available in the Asymptote gallery and user-written wiki:
http://asymptote.sourceforge.net/gallery/
http://asymptote.sourceforge.net/links.html

2 An example

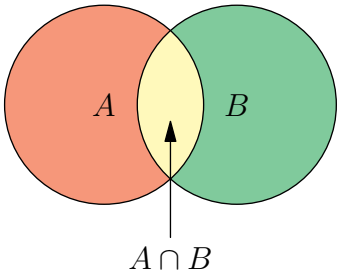
The following example illustrates the four Asymptote graphics primitives (draw, fill, clip, and label):

```
size(0,100);
pair z1=(-1,0);
pair z2=(1,0);
real r=1.5;
path c1=circle(z1,r);
path c2=circle(z2,r);

fill(c1,lightred);
fill(c2,lightgreen);
picture intersection;
fill(intersection,c1,lightred+lightgreen);
clip(intersection,c2);
add(intersection);

draw(c1);
draw(c2);

label("$A$",z1);
label("$B$",z2);
path g=(0,-2)--(0,-0.25);
draw(Label("$A\cap B$",0),g,Arrow);
```



3 History

Asymptote began as a University of Alberta summer undergraduate research project in 2002, after looking into the feasibility of overhauling Hobby's METAPOST[2] to use floating-point numerics. Many of the current limitations of METAPOST derive from METAFONT: numbers are stored in a low-precision fixed-point format that is adequate for representing points in a glyph but restrictive for diagrams and scientific computations. While the IEEE floating-point numeric format was not standardized until 1985, the initial development of TEX dates back to 1978. At that time, the decision to use only fixed-point (integer-based) arithmetic was perfectly reasonable: Knuth wanted to guarantee that TEX and METAFONT would produce exactly the same bit-mapped output on any existing hardware.

We quickly determined that a complete rewrite of the underlying graphics engine would be necessary. After six months of work, our compiler for a new graphics language could finally draw a sine curve. One of the four Asymptote primitives had now been implemented!

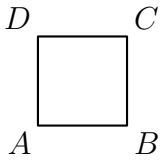
From this very humble beginning, Asymptote evolved rapidly. The basic fill operation was straightforward to implement. The most crucial advance,

[1] Andy Hammerlindl, John Bowman, and Tom Prince, available under the GNU General Public License from http://asymptote.sourceforge.net/

[2] METAPOST is a modified version of METAFONT, the program that Knuth wrote to produce the Computer Modern fonts used with TEX.

aligning TEX labels at the correct positions, was accomplished three months later, using compass directions or arbitrary angles to specify label alignments:

```
size(0,2cm);
draw((0,0)--(1,0)--(1,1)--(0,1)--cycle);
label("$A$",(0,0),SW);
label("$B$",(1,0),SE);
label("$C$",(1,1),NE);
label("$D$",(0,1),NW);
```



The idea was to leave the entire label typesetting to LATEX, inserting PostScript layers with `\includegraphics`, and thereby avoid the complications and kerning issues inherent in the METAPOST approach of post-processing the DVI file. To accomplish this, `Asymptote` communicates with TEX *via* a bidirectional pipe in two passes: the first pass is used to obtain label sizing information, while the second pass performs the final typesetting directly into DVI/PostScript or PDF. Label clipping and transforms are implemented with PostScript or PDF specials.

A third co-developer, Tom Prince, joined us in 2004. He contributed a method for embedding `Asymptote` code directly in LATEX source files. With Tom's help, we ported more reliable versions of the METAPOST algorithms for basic Bézier path operations such as splitting into subpaths, computing points of tangency, determining path bounds, and finding intersection points. Robust arc length and arc time computations were implemented with adaptive Simpson integration, which was determined to be more efficient than Bézier subdivision.

On November 7, 2004, we posted our first public release, version 0.51, on `sourceforge.net`. Since then, the user base and the list of new features have grown dramatically. The current version at the time of this writing is 1.42. Like METAPOST, `Asymptote` runs on GNU/Linux and other UNIX-like operating systems, Microsoft Windows, and Mac OS X. Precompiled `Asymptote` binaries are now included in several major Linux distributions.

4 Language features

`Asymptote` uses lexical analysis, parsing, and intermediate code generation to compile commands into virtual machine code, optimizing speed without sacrificing portability. Double-precision floating-point numbers and 64-bit integers make arithmetic overflow, underflow, and loss of precision issues much less troublesome than they are in METAPOST programs. `Asymptote` represents curves as cubic Bézier splines, but can easily handle large data values and the pathological behaviour of functions like $x\sin(1/x)$ near the origin. It also supports new path operations like computing the winding number of a path relative to a given point, which is useful for identifying the region bounded by a closed path.

Most users find the `Asymptote` language much easier to program in, with its C++/Java-like syntax (augmented to support high-order functions), than METAPOST, with its awkward and somewhat confusing `vardef` macros. `Asymptote` also borrows several ideas from `Python`, such as named function arguments and array slices. High-level graphics commands are implemented in the `Asymptote` language itself, allowing them to be easily tailored to specific user applications.

Like METAPOST, `Asymptote` is mathematically oriented. For example, one can rotate vectors by complex multiplication and apply affine transformations (shifts, rotations, reflections, and scalings) to pairs, triples, paths, pens, strings, pictures, and other transforms.

4.1 Functions

`Asymptote` is the only language we know of that treats functions as variables, but allows overloading by distinguishing variables based on their signatures. In fact, function definitions are just syntactic sugar for assigning function objects to variables:

```
real square(real x) {return x^2;}
```

is equivalent to

```
real square(real x);
square=new real(real x) {return x^2;};
```

`Asymptote` supports a more flexible mechanism for default function arguments than C++: they may appear anywhere in the function prototype. This feature underlies `Asymptote`'s greatest strength: sensible default values for the basic graphical elements allow beautiful graphs and drawings to be created with extremely short scripts, without sacrificing the flexibility for detailed customization. Default arguments are evaluated as `Asymptote` expressions in the scope where the function is defined.

Because certain data types are implicitly cast to more sophisticated types, one can often avoid ambiguities in function calls by ordering function arguments from the simplest to the most complicated. For example, given

```
real f(int a=1, real b=0) {return a+b;}
```

the call `f(1)` returns 1.0, but `f(1.0)` returns 2.0. It is sometimes difficult to remember the order in which arguments appear in a function declaration. Python-style named (keyword) arguments make calling functions with multiple arguments easier: the above examples could respectively be written `f(a=1)` and `f(b=1)`. An assignment of a function argument is interpreted as an assignment to a parameter of the same name in the function signature, not in the local scope of the calling routine.

Rest arguments allow one to write functions that take a variable number of arguments. For example, the following function sums its arguments:

```
real sum(... real[] nums) {
  real total=0;
  for(real x : nums)
    total += x;
  return total;
}
```

As in other modern languages, functions can call themselves recursively. Operators, including all of `Asymptote`'s built-in arithmetic and path operations, are just syntactic sugar for functions that can be addressed and defined with the `operator` keyword.

`Asymptote` functions are first-class values, allowing them to be defined within, passed to, and returned by other functions. This is convenient when one wants to graph a sequence of functions such as $f_n(x) = n\sin(x/n)$ for $n = 1$ to 5 from $x = -10$ to 10:

```
import graph;
typedef real function(real);

function f(int n) {
  real fn(real x) {
    return n*sin(x/n);
  }
  return fn;
}

for(int n=1; n <= 5; ++n)
  draw(graph(f(n),-10,10));
```

Anonymous functions can be created with the keyword `new`, so that the function definition in the previous example could be simplified to

```
function f(int n) {
  return new real(real x) {
    return n*sin(x/n);
  };
}
```

5 Modules

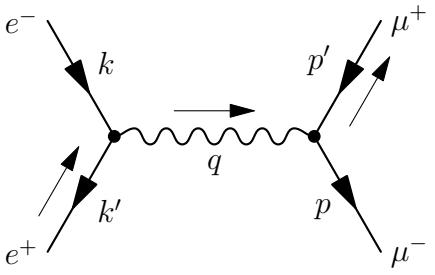
Function and structure definitions can be grouped into modules:

```
// powers.asy
real square(real x) {return x^2;}
real cube(real x) {return x^3;}
```

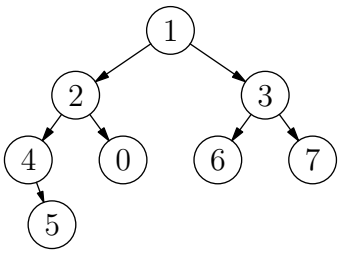
and imported:

```
import powers;
path square(real x) {
  return scale(x)*unitsquare;
}
real four=powers.square(2.0);
real eight=cube(2.0);
```

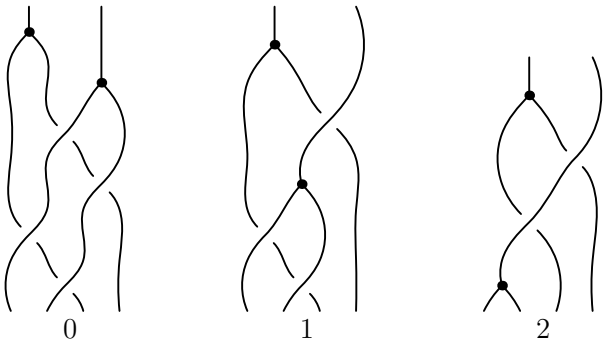
For example, `Asymptote` ships with modules for Feynman diagrams:



data structures,



and algebraic knot theory:



Modules are written in high-level `Asymptote` code. Users have contributed modules tailored to many other specialized applications (such as flowcharts and computer-aided design).

6 Graphics features

METAPOST does not support many important features of PostScript. For example, only connected PostScript subpaths are supported. Regions must be simply connected (have no holes) and can only be filled with uniform RGB colours. In addition to native support for three-dimensional graphics, also lacking in METAPOST, these missing features have been implemented in `Asymptote`.

6.1 Pens

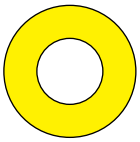
Pens provide a context for the four primitive drawing commands: they specify attributes such as color, line type, line width, text alignment, font, font size, fill rule, and filling patterns. For non-solid line types, dash lengths are by default slightly adjusted to fit the path arc length (for example, to allow publication quality legend entries in graphs with multiple line types). Interesting calligraphic effects are possible by applying transforms to the (normally circular) pen nib or even using a polygonal pen nib (which need not be convex):

The default pen, called `currentpen`, provides the same functionality as the METAPOST `pickup` command. Colors can be specified in any one of the PostScript colorspaces: grayscale, RGB, and CMYK.

6.2 Subpaths

An `Asymptote` path, being connected, is equivalent to a PostScript subpath. The binary operator `^^`, which requests that the pen be moved (without drawing or affecting endpoint curvatures) from the final point of the left-hand path to the initial point of the right-hand path, may be used to group several `Asymptote` connected paths into a `path[]` array (equivalent to a PostScript path). While this facility is merely convenient for drawing an object like the skeleton of a cube (without retracing), it is essential for filling nonsimply-connected regions:

```
path g=scale(2)*unitcircle;
filldraw(unitcircle^^g, evenodd+yellow, black);
```



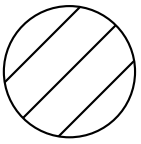
The PostScript even-odd fill rule here specifies that only the region bounded between the two unit circles is to be filled. In this example, the same effect can be achieved by using the default zero winding number fill rule, if one is careful to alternate the orientation of the paths:

```
filldraw(unitcircle^^reverse(g), yellow,
         black);
```

6.3 Patterns

One can also construct custom pictures to be used as tiling patterns for fill or draw operations. The tiling pattern can be assigned a name that can subsequently be used to construct a patterned pen. For example, a hatch pattern can be generated like this:

```
import patterns;
add("hatch",hatch());
filldraw(unitcircle,pattern("hatch"));
```



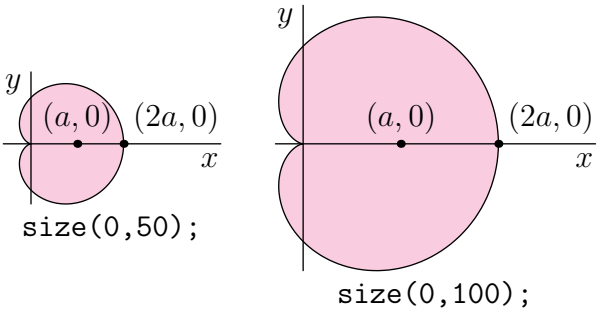
6.4 Shading

`Asymptote` supports axial and radial gradient shading, lattice shading, Gouraud shading, and shading of Coons and tensor product patches; the latter forms are essential for three-dimensional rendering in the presence of a light source. `Asymptote` also supports a true unfill operation implemented with clipping (the METAPOST unfill command simply fills with a fixed background color).

6.5 Automatic picture sizing

A frame is a canvas for drawing in PostScript coordinates, much like a `picture` in METAPOST. However, working directly in PostScript coordinates is often inconvenient, requiring the tedious introduction of manual scaling factors.

Pictures are high-level structures that provide canvases for drawing in a user-specified Cartesian coordinate system. Automatic sizing allows pictures to be constructed in user coordinates and then automatically scaled to the desired final size:



Eventually, one must fit a picture to a PostScript frame. This requires deferred drawing: a graphical object cannot be drawn until the actual scaling of the user coordinates (in terms of PostScript coordinates) is known. One needs to queue a function that can draw the scaled object later, when this scaling is known. For example, the `draw` function for pictures in scalable user coordinates is implemented in terms of the underlying PostScript-coordinate `draw` primitive for frames like this:

```
void draw(picture pic=currentpicture,
          path g, pen p=currentpen) {
  pic.add(new void(frame f, transform t) {
      draw(f,t*g,p);
    });
  pic.addPoint(min(g),min(p));
  pic.addPoint(max(g),max(p));
}
```

Here, the `addPoint` function stores bounding box information as user (e.g. path) coordinates, which scale linearly with the picture size, and true-size (e.g. pen) coordinates, which remain fixed.

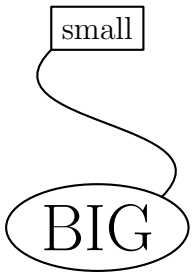
The sizing constraints that arise between scalable objects and fixed-sized attributes (typically labels, dots, linewidths, and arrowheads) reduce to a linear programming problem that is solved by the simplex method. However, a figure can easily produce thousands of restrictions, making direct application of the simplex method time consuming. In practice, most of these restrictions are redundant: in the case of concentric circles, only the largest circle needs to be accounted for. When sizing a picture, `Asymptote` first determines which coordinates are maximal (or minimal) and sends only active constraints to the simplex algorithm. The entire picture-sizing algorithm, including the simplex method, is implemented in high-level `Asymptote` code.

This example illustrates how deferred drawing can be used to draw paths around text labels and then connect them (an `object` is a unifying structure that a label or frame can be implicitly cast to):

```
size(0,100);
pair A=(0,1);
pair B=(0,0);

object small=draw("small",box,A,1mm);
object big=draw("\huge BIG",ellipse,B,1mm);

add(new void(frame f, transform t) {
    draw(f,point(small,SW,t){SW}
           ..{SW}point(big,NE,t));
  });
```

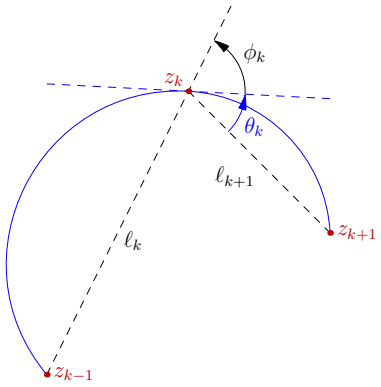


6.6 Three-dimensional graphics

We now describe our three-dimensional generalization of Hobby's prescription for drawing an aesthetically pleasing, numerically efficient, interpolating spline through a set of nodes, given optional tangent directions and endpoint curvatures [2, 3]. This generalization is shape invariant under three-dimensional rotation, scaling, and translation. In the planar case, it reduces to the two-dimensional algorithms found in METAFONT, METAPOST, and `Asymptote`.

In two dimensions, a tridiagonal system of linear equations is first solved to determine any unspecified directions θ_k and ϕ_k through each node z_k:

$$\frac{\theta_{k-1} - 2\phi_k}{\ell_k} = \frac{\phi_{k+1} - 2\theta_k}{\ell_{k+1}}.$$



The resulting shape may be adjusted by modifying the default *tension* parameters and *curl* boundary conditions (cf. [3]).

Having prescribed outgoing and incoming path directions $e^{i\theta}$ at node z_0 and $e^{i\phi}$ at node z_1 relative to the vector $z_1 - z_0$, any unspecified control points are then determined by the equations

$$u = z_0 + e^{i\theta}(z_1 - z_0)f(\theta, -\phi),$$
$$v = z_1 - e^{i\phi}(z_1 - z_0)f(-\phi, \theta),$$

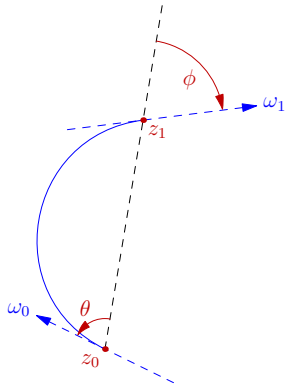
where the *relative distance* function $f(\theta, \phi)$ is given in [2] and [3].

In three dimensions, it is natural to require that our generalization reduce in the planar case to the usual two-dimensional algorithm. Therefore, any unknown incoming or outgoing tangent directions are first determined by applying Hobby's direction

algorithm in successive planes containing the three points z_{k-1}, z_k, and z_{k+1}. The only ambiguity that can arise is in the overall sign of the angles, which relates to viewing each local two-dimensional plane from opposing normal directions. A reference vector constructed from the mean unit normal of successive segments is used to resolve such ambiguities.

A formula for the three-dimensional control points u and v follows on expressing Hobby's algorithm in terms of the absolute incoming and outgoing unit direction vectors ω_0 and ω_1, respectively:

$$u = z_0 + \omega_0 \, |z_1 - z_0| \, f(\theta, -\phi),$$
$$v = z_1 - \omega_1 \, |z_1 - z_0| \, f(-\phi, \theta),$$



where we interpret θ and ϕ as the angle between the corresponding path direction vector and $z_1 - z_0$. In this case there is an unambiguous reference vector for determining the relative sign of the angles θ and ϕ.

Unfortunately, PostScript and PDF support only Bézier splines, which are shape invariant under affine (orthographic) projections (parallel lines being projected to parallel lines), but not perspective projections. In any dimension, applying an affine transformation $x_i' = A_{ij}x_j + C_i$ to a cubic Bézier curve $x(t) = \sum_{k=0}^{3} B_k(t)P_k$ for $t \in [0,1]$, where $B_k(t)$ is the kth cubic Bernstein polynomial yields the Bézier curve

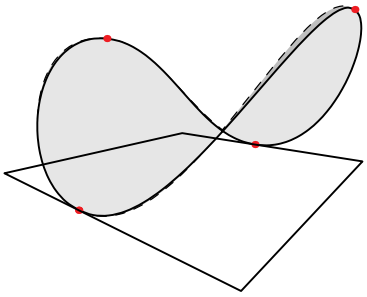
$$x_i'(t) = \sum_{k=0}^{3} B_k(t)A_{ij}(P_k)_j + C_i = \sum_{k=0}^{3} B_k(t)P_k'$$

in terms of the transformed k^{th} control points P_k', noting that $\sum_{k=0}^{3} B_k(t) = 1$. Thus, for orthographic projections, both the nodes and control points of three-dimensional Bézier curves can simply be projected to obtain their two-dimensional counterparts.

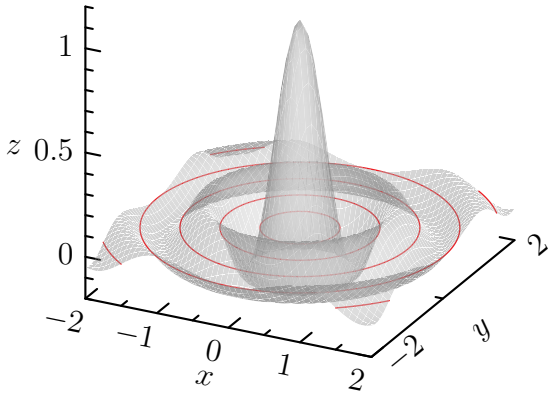
Non-uniform rational B-splines have the advantage of being invariant even in the presence of perspective distortion, since they are Bézier curves in the projective space described by *homogeneous coordinates*, where (x, y, z, w) is considered as equivalent to $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$. For example, the `Asymptote` syntax

```
(1,0,0)..(0,1,1)..(-1,0,0)..(0,-1,1)..cycle
```

describes a saddle-like path in three dimensions. The result of simply projecting the nodes and control points of this three-dimensional Bézier curve to a two dimensional Bézier curve is indicated by the dashed path in the following perspective projection. The true projection, described by the two-dimensional nonuniform rational B-spline represented by the solid curve, can be efficiently approximated as a two-dimensional Bézier curve by introducing additional nodes and control points. An algorithm for doing this efficiently will be presented in a future publication.



The three-dimensional graph of the sinc function below illustrates Gouraud shading, advanced contour path computations, and PDF transparency:



6.7 Scientific graphs

`Asymptote` ships with a sophisticated `graph` module that quickly allows one to draw publication-quality scientific and textbook-style graphs, such as the two-dimensional polar coordinate graph of the cardioid shown in Section 6.5 and the three-dimensional surface plot shown in Section 6.6. It supports features such as legends, custom graph markers, secondary axes, custom (e.g. base 2) axes scalings, broken axes, and custom tick label formats and locations.

7 Graphical user interface

Recent versions of `Asymptote` include an innovative graphical user interface, written in Python/TK, that allows one to modify existing graphical objects and

draw new ones. The modified figure can then be saved and processed as a normal `Asymptote` file. This allows the user to exploit the best features of the script (command-driven) and graphical-interface methods for producing figures.

8 Slide presentations

`Asymptote` also includes a convenient `slide` module for preparing slide presentations, including embedded clickable high-resolution PDF movies (with optional control panels). This module has the advantage over existing LaTeX presentation packages of providing built-in graphics support, including object alignment, in addition to the full power of TeX.

9 Animations

While `Asymptote` can create MPEG and animated GIF movies, the lossless inline PDF movies it can generate with the help of the LaTeX `animate.sty` package are of a much higher quality. Sample animations can be found in the `Asymptote` gallery.

10 Equation solving

Unlike METAFONT and METAPOST, `Asymptote` is not built on top of an implicit linear equation solver and therefore does not automatically have the notion of a `whatever` unknown. Although such an implicit equation facility could certainly be added (perhaps using the notation `?=` since `=` denotes assignment in `Asymptote`), we have noticed that the most common uses of `whatever` in METAPOST are covered by explicit functions like `extension` in the `math` module (which returns the intersection point of the extensions of two line segments). We find the use of routines like `extension` to be more explicit and less confusing, particularly to new users. But we could be persuaded to add implicit equation solving if someone can justify the need (so far no one has provided us with an example that cannot already be done elegantly in `Asymptote`). In the meantime, one can always use the explicit built-in linear solver `solve` (based on LU decomposition) or one of the numerically robust specialized solvers `tridiagonal`, `quadraticroots`, `cubicroots`, and `quarticroots`.

11 Future plans

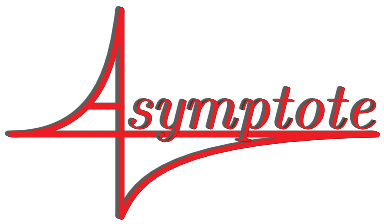
Thanks to the LaTeX `movie15` package, `Asymptote` can embed three-dimensional U3D files into PDF files. In the near future, we plan to generate U3D data (or possibly the more advanced PRC format) directly from `Asymptote`'s internal three-dimensional representations. This will provide the scientific community with a self-contained and powerful facility for generating interactive three-dimensional PDF files.

12 Credits

Financial support for the development of `Asymptote` was generously provided by the Natural Sciences and Engineering Research Council of Canada, the Pacific Institute for Mathematical Sciences, and the University of Alberta Faculty of Science.

We would like to acknowledge enthusiastically the previous work of John D. Hobby, which inspired the development of `Asymptote`, and Donald E. Knuth, for his exceptionally high quality TeX typesetting system, without which none of these later developments would have been possible. We also thank L. Nobre and Troy Henderson for suggesting techniques for approximating nonuniform rational B-splines.

The authors of `Asymptote` are Andy Hammerlindl, John Bowman, and Tom Prince. Sean Healy designed the `Asymptote` logo (below). Other contributors include Radoslav Marinov, Orest Shardt, Chris Savage, Philippe Ivaldi, Olivier Guibé, Jacques Pienaar, Mark Henning, Steve Melenchuk, Martin Wiebusch, and Stefan Knorr.



References

[1] The Art of Problem Solving: Asymptote (Vector Graphics Language). Available online at `http://www.artofproblemsolving.com`, 2007.

[2] John D. Hobby. Smooth, easy to compute interpolating splines. *Discrete Comput. Geom.*, 1:123–140, 1986.

[3] Donald E. Knuth. *The METAFONTbook*. Addison-Wesley, Reading, Massachusetts, 1986.

◇ John C. Bowman
Dept. of Mathematical and Statistical Sciences
University of Alberta
Edmonton, Alberta
Canada T6G 2G1
`bowman (at) math dot ualberta dot ca`
`http://www.math.ualberta.ca/~bowman/`

◇ Andy Hammerlindl
University of Toronto
Dept. of Mathematics
Toronto, Ontario
Canada M5S 2E4
`andy (at) math dot toronto dot edu`