Parametric Polynomial Curves

Dr. David J Stahl, Jr.
US Naval Academy
stahl@usna.edu

Abstract: *Spline curves and surface patches have an innate mathematical beauty and broad practical application in the field of computer graphics. Yet the subject proves difficult to convey to beginning graphics students averse to math and theory in general. The difficulty is mitigated by having students complete an implementation of carefully prepared scaffold code. A particular code framework allows focusing student effort on understanding the algorithm and the theory rather than the visualization details. In this manner understanding is developed and reinforced by means of an exercise no more difficult than a short lab assignment.*

Keywords: 2D modeling, algorithm development, rendering.

1 Introduction

Few subjects in computer graphics of such practical application and visually pleasing results as spline curves and surface patches have their innate mathematical beauty. Yet for this author, over a number of years teaching undergraduate introductory computer graphics the subject has proven to be one of the most difficult to convey. Blank student expressions and outright groans at the first mention of "piecewise continuous parametric polynomial curves and surfaces" serve notice of the difficult task ahead.

The problem is multifaceted. A complete treatment of the subject could easily consume an entire semester covering the myriad related topics, techniques and details: history and development, forms and conversions (Hermite, Bezier, B-spline, …), derivations and representations (Bernstein, Cox-deBoor, de Casteljau …), manipulation (knot insertion, curve elevation, hierarchical refinement), and so on. A rigorous theoretical treatment requires students have a level of comfort with matrix algebra, summation notation, recurrences, and so on. Careful presentation by the instructor and meticulous, close attention to detail on the part of the student are both necessary to understand this theory - and then there's the implementation to deal with. Unfortunately for such a core topic in the computer graphics body of knowledge, students with a weak foundation in math, aversion to theory and a general impatience with detail easily perceive it as dreary and tedious. End-of-course student evaluations reveal this quite clearly!

Student dissatisfaction with this particular topic of computer graphics, although perhaps more pronounced, fundamentally appears no different than in other areas. Surveys conducted at the beginning of multiple offerings of an undergraduate introductory computer graphics course show that impressive, sophisticated game and movie graphics attract them to the subject. These "instant gratification", YouTube video savvy consumers however are also not interested in the math and theory behind the visuals. They seek to quickly create impressive results without the encumbrance of having to understand the algorithms or mathematics on which they are based.

The challenge of course is to make these theoretical underpinnings of computer graphics – and in this particular case, the mathematics of spline curves - more palatable. This paper describes one exercise in a series that approaches the problem with active learning. The classic approach of lecture and reading prepares students with a basic understanding of an algorithm or technique, and a carefully designed follow-on lab exercise reinforces understanding through an implementation. Students can typically complete the exercise during a one to two hour lab period, thus putting theory to practice in a manner that rewards effort with immediate visual results. Moreover, the programming framework imposed on student solutions permits focusing their effort on the algorithm rather than the API used to render it.

2 Educational Goals

The student should be able to explain the following fundamental mathematical concepts associated with parametric polynomial curves: geometric constraints, blending functions, piecewise continuity, and basis matrix formulation. The student will have completed a source code implementation to render 2D curves using cubic Lagrange polynomial interpolation, as well as cubic curves of the following forms: Hermite, Bezier, uniform B-Spline, and Cardinal. Students will have benefited from the additional practical programming experience gained. The student should subsequently be able to extend the implementation to 3D curves and surface patches and apply it to camera motion. The exercise could serve as an introduction to more advanced concepts such as non-uniform B-Splines, conversion between bases, degree elevation and knot insertion.

3 Methodology

The approach described in this paper is employed in an elective first course in undergraduate computer graphics offered to computer science and information technology majors. As prerequisites, students will have completed courses in discrete math and programming through data structures. No prior background in computer graphics is assumed. The course meets twice weekly in two-hour sessions, nominally scheduled as one hour of lecture followed immediately by one hour of lab. With the lab tied closely to the lecture topic, two-hour sessions allow the flexibility of adjusting lecture or lab time as needed.

In the lab we use OpenGL as the graphics API and GLUT3.7 as the user interaction and windowing system interface API. GLUT was chosen for its straightforward simplicity, supporting the course design objective of providing short lab exercises that focus student programming effort on the algorithm rather than the API. A more capable user interface/windowing API such as fltK could be used, at the time-consuming expense of students having to attend to more programming details not related to the algorithm.

The course has been taught using several different programming environments: a Sun SPARC workstation lab using *Solaris*, *g++*, *xemacs* and *make*; a PC lab using *Windows XP* and *Microsoft Visual Studio C++ 6.0*; and most recently a dual-boot Solaris/XP lab. Any platform supporting the OpenGL/GLUT API could be used. Students more comfortable with the Mac OS or Linux, for example, have sometimes brought personal laptops to the lab to program in those environments.

The course employs the following pedagogical approach. Preparatory material that is required to successfully complete the lab exercise comes from assigned out-of-class reading [1] [2] followed by a formal lecture. Students are expected to take notes during the lecture, supplementing a minimal handout with details of the lecture topic being discussed. Work begins on the lab exercise immediately after the lecture, with students encouraged to collaborate on developing a solution. Materials required to complete the lab are obtained by downloading a tar or zip file from the course website. The typical assignment archive contains a document describing the exercise, a correct solution executable, sample data if needed, and one or more source code files that includes scaffold code.

One of the key elements of the approach is the organization of the provided code. Students are required by course policy to adhere to a rather rigid arrangement of files, which appear in most lab exercises. A program is organized as a separate header and source code file pair for each set of related GLUT callbacks, as follows:

Filename:	Purpose:
`main.[h, cpp]`	- Main program event-driven loop
`init.[h, cpp]`	- Graphics and application setup
`render[.h, cpp]`	- Display callback
`view.[h, cpp]`	- View setup and window reshape callback
`mouse.[h, cpp]`	- Mouse interaction callbacks
`keyboard.[h, cpp]`	- Keyboard interaction callbacks
`menu.[h, cpp]`	- Menu callbacks

An additional one or two pairs of files specifically related to the particular problem at hand are also usually present – for example, `splines.[h, cpp]`. Enforcing this separation by file according to logical purpose has several benefits. The common framework makes it easy for both student and instructor to locate specific code according to its functionality, and additionally affords collaborative student work. As a practical matter it makes grading assignments much easier and simplifies the Makefile when one is required by the development environment. Most importantly however, an easily understood common code framework obviates students having to spend time and effort in designing their own high-level code structure, but rather allows them to focus effort on implementing the algorithm. The rigid code organization is introduced early in the course during the discussion of event-driven programming. Students quickly become familiar with the code structure, easily moving from file to file as needed., and in fact, have come to expect similar structuring in follow-on advanced graphics courses.

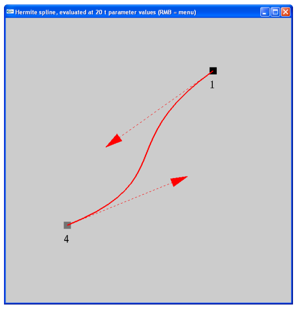
Scaffold code provided to the students is prepared by removing key parts of the algorithm from a thoroughly commented working solution. Students must refer to lecture notes, handouts they have filled in, or texts to finish the implementation.

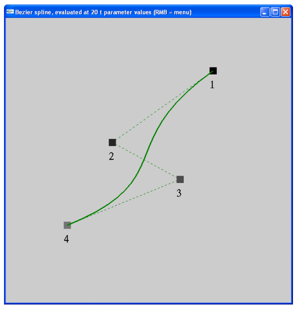
Complete the given source code to draw various parametric *cubic* polynomial curves.

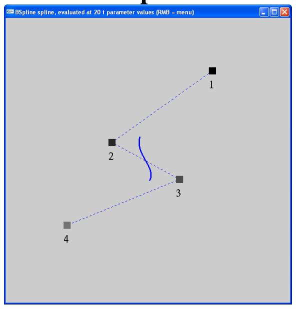
The coordinates of the 4 control points are given: P_1, P_2, P_3, P_4.

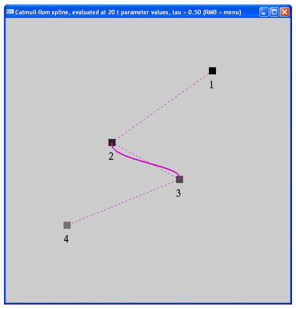
The program should draw Hermite, Bezier, B-Spline and Cardinal splines: $Q(t) = \sum_{i=1}^{4} B_i(t) P_i$.

Hermite **Bezier** **B-Spline** **Cardinal**


Hermite spline, evaluated at 20 t parameter values (RMB = menu)
1
4


Bezier spline, evaluated at 20 t parameter values (RMB = menu)
1
2
3
4

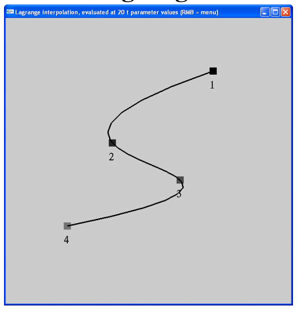

BSpline spline, evaluated at 20 t parameter values (RMB = menu)
1
2
3
4


Catmull-Rom spline, evaluated at 20 t parameter values, tau = 0.50 (RMB = menu)
1
2
3
4

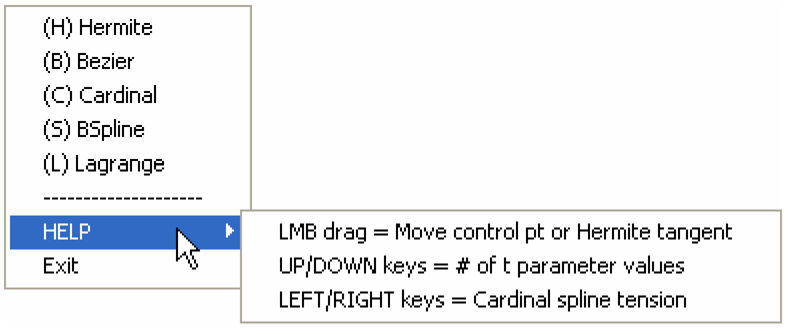
It should also use Lagrange polynomials to interpolate the control points:

$$Q(t) = P_0 L_0^n(t) + P_1 L_1^n(t) + \cdots + P_n L_n^n(t)$$

Lagrange


Lagrange interpolation, evaluated at 20 t parameter values (RMB = menu)
1
2
3
4

The menu structure is as shown:

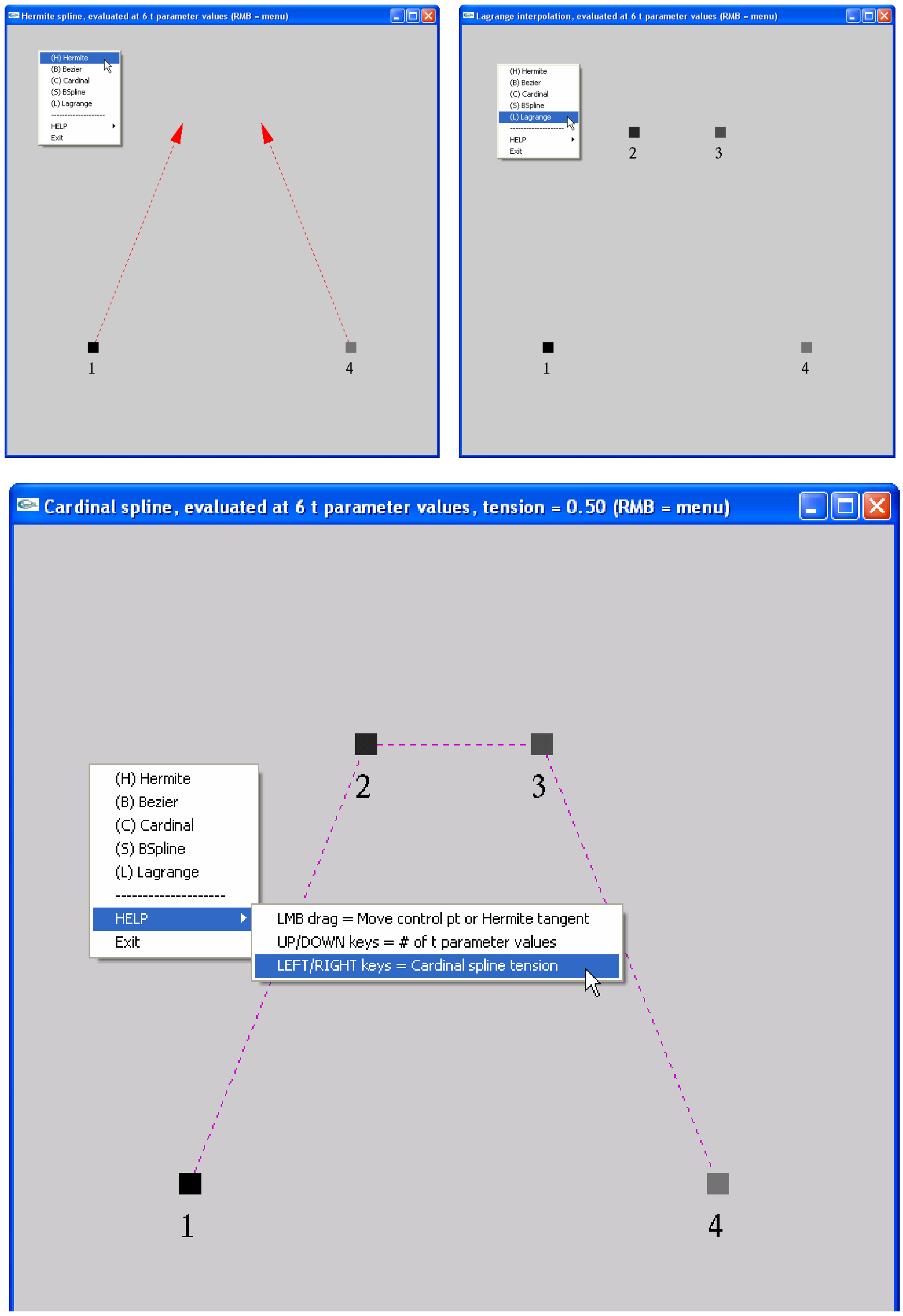

(H) Hermite
(B) Bezier
(C) Cardinal
(S) BSpline
(L) Lagrange

HELP
Exit

LMB drag = Move control pt or Hermite tangent
UP/DOWN keys = # of t parameter values
LEFT/RIGHT keys = Cardinal spline tension

You should only have to add code to **`splines.cpp`** and **`Lagrange.cpp`**.

Figure 1. Spline curves assignment. Students are given scaffold code that must be completed.


Hermite spline, evaluated at 6 t parameter values (RMB = menu)
(H) Hermite
(B) Bezier
(C) Cardinal
(S) BSpline
(L) Lagrange

HELP
Exit
1
4
Lagrange interpolation, evaluated at 6 t parameter values (RMB = menu)
(H) Hermite
(B) Bezier
(C) Cardinal
(S) BSpline
(L) Lagrange

HELP
Exit
1
2
3
4
Cardinal spline, evaluated at 6 t parameter values, tension = 0.50 (RMB = menu)
(H) Hermite
(B) Bezier
(C) Cardinal
(S) BSpline
(L) Lagrange

HELP
Exit
LMB drag = Move control pt or Hermite tangent
UP/DOWN keys = # of t parameter values
LEFT/RIGHT keys = Cardinal spline tension
1
2
3
4

Figure 2. Output of compiled scaffold code as given to the students. Upper left: Hermite curve is selected. Upper right: Lagrange interpolation is selected. Bottom: Cardinal spline. The student must complete the implementation to render cubic parametric polynomial curves.

```cpp
// Lagrange.cpp

#include "main.h"

//             n
// Implements L(t) for n = 3 (interpolating 4 points)
//             i
double Lin(int i, double t)
{
      // Initialize the running product
      double L; // = ?;

      // Use equally spaced parameter values
      // (cubic curve: 4 values in [0.0 1.0])
      double tj[4] = { 0.0, 0.0, 0.0, 0.0 };
      //           = {   ?,   ?,   ?,   ? };

      //                         3        3   t - tj
      // Loop over j to compute L(t) = PROD( ------- ), j != i
       //                         i      j=0  ti - tj
      for( /* ? */; 0 /* ? */; /* ? */ )
      {
            L; // ?
      }

      return L;
}
//                         n
// Implements Q(t) = SUM( L(t) * Pi ) (a point on the Lagrange curve)
//                         i
void LagrangePoint( double t, double& x, double& y )
{
      x = 0; // = ?
      y = 0; // = ?
}
```

Figure 3. Scaffold code provided to the student for cubic Lagrange interpolation.

```
// splines.cpp
#include "main.h"

// globals:
// geometric constraints: control points and Hermite
tangents
double P1[2], P2[2], P3[2], P4[2];
double R1[2], R4[2];

// Which form: Hermite, Bezier, BSpline, Catmull-Rom,
Lagrange ?
int method;

int n_tval;             // # of parameter values to
evaluate
double tension, tau;    // Cardinal spline tension

// Hermite basis functions
double BH0( double t )  { return ( 0  /* ? */ ); }
double BH1( double t )  { return ( 0  /* ? */ ); }
double BH2( double t )  { return ( 0  /* ? */ ); }
double BH3( double t )  { return ( 0  /* ? */ ); }

// Bezier basis functions
double BB0( double t )  { return ( 0  /* ? */ ); }
double BB1( double t )  { return ( 0  /* ? */ ); }
double BB2( double t )  { return ( 0  /* ? */ ); }
double BB3( double t )  { return ( 0  /* ? */ ); }

// ...

// Implements Q(t) = SUM( Bi * Pi )
void HermitePoint( double t, double& x, double& y )
{
  x = 0; // = ?
  y = 0; // = ?
}

void BezierPoint( double t, double& x, double& y )
{
  x = 0; // = ?
  y = 0; // = ?
}

// ...
```

Figure 4. Parts of scaffold code provided to the student for drawing cubic spline curves.

Lagrange interpolation

A set of $n+1$ points, $P_0, P_1, \cdots, P_n$, can be interpolated,

Curve $Q(t) =$ ____

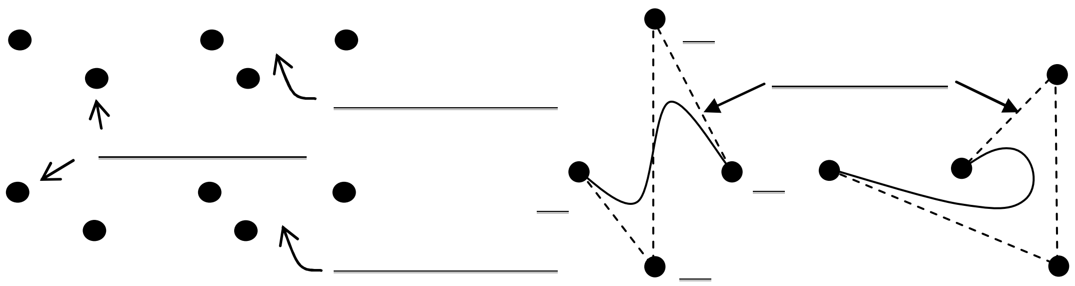
where $L_i^n(t)$ are *Lagrange polynomials of degree n*:

$L_i^n(t) =$ ____________________ $,(j \neq i) =$ ________________________ $,(j \neq i).$

Example: 3rd degree Lagrange polynomials (n = 3) are:

$L_i^3(t) =$ ____ $,(j \neq i).$

Piecewise continuous parametric polynomial curves

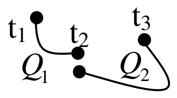


Continuity at the join point between segments

C^0, C^1, C^2, ... "______________" continuity.

G^0, G^1, G^2, ... "______________" continuity.

Two polynomial curve segments: $Q_1(t) \quad t \in [t_1, t_2]$, $Q_2(t) \quad t \in [t_2, t_3]$



C^0, G^0: $Q_1(t_2)$ = ________ (coordinates are equal at the join point)

G^1: $Q_1'(t_2)$ = ________ (parametric 1st derivatives are *proportional* at the join point)

___: $Q_1'(t_2) = Q_2'(t_2)$ (parametric 1st derivatives (tangents) are equal at the join point)

G^2: ________ $= kQ_2''(t_2)$ (____________________________________)

C^2: ________ = ______ (____________________________________)

Figure 5. Part of a handout tied to subject material presented to students in a lecture. Completing the handout facilitates completing the associated lab assignment.

Cubic Bezier spline

Shape is determined by: • two ____________________________, ___ and ___ .

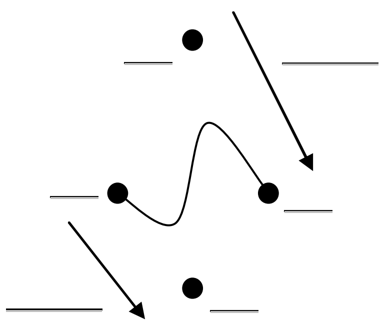
• two ___________________________, ___ and ___ .

Geometry vector: ________________________ .

Endpoint tangent vectors: _______, and _______ .

A cubic Bezier curve _______________the two endpoints and __________________ the intermediate two points:

$$G_B = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$



The cubic Bezier endpoint tangents are related to the cubic Hermite endpoint tangents:

$R_1 =$___________ $R_4 =$___________

M_{HB}, relates ________________________ to ___________________________:

$$G_H = \begin{bmatrix} \_\_\_ \\ \_\_\_ \\ \_\_\_ \\ \_\_\_ \end{bmatrix} = M_{HB} \cdot G_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} \_\_\_ \\ \_\_\_ \\ \_\_\_ \\ \_\_\_ \end{bmatrix}$$

Hermite basis: $Q(t) = T \cdot M_H \cdot G_H$. Substitute $M_{HB} \cdot G_B$ for G_H:

$$Q(t) = T \cdot M_H \cdot (M_{HB} \cdot G_B)$$
$$Q(t) = T \cdot (M_H \cdot M_{HB}) \cdot G_B$$

$M_H \cdot M_{HB} \Rightarrow M_B$, the _______________________

$$M_H \cdot M_{HB} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad & \quad \\ & & & \\ & & & \\ & & & \end{bmatrix} = M_B$$

Figure 6. More handout. Students are expected to complete the handout from the reading and the lecture.

4 Assessment

The exercise described in this paper is one of a series that address a recurring curriculum-wide issue raised in formal student end-of-course critiques. Students consistently seek more hands-on practical programming experience yet would prefer less exposure to theory in general and math in particular. These are conflicting goals in any creditable curriculum however. Curriculum guidelines published by the major professional computing societies clearly emphasize connecting theory and practice and the equal importance of an implementation and its theoretical underpinnings [3]. In the graphics curriculum, the exercise and approach described in this paper addresses our student concerns but does not sacrifice theory to emphasize practice.

As an informal measure of success, in all offerings of the undergraduate computer graphics course that include the exercise described in this paper, not a single student has been unable to complete the assignment. In refining the lab exercises over several course offerings, comments made by students in end-of-course critiques have shifted from complaints about "too much theory", to remarks that the approach is useful and appreciated. When asked to comment about the appropriateness of topics, where students previously pleaded that parametric curves be dropped, they now ask that the course additionally address parametric surfaces. As a further measure of success, it is noted that our undergraduate introductory graphics course materials have been used at the University of Pittsburgh and DelMar College. Informal and anecdotal evidence thus suggests the approach is effective.

5 Conclusions

With theoretical material presented first, being able to work exclusively on implementing an algorithm and having immediate visual feedback - whether it indicates error or success in the implementation - seems to make students more willing to absorb the underlying theory. Giving students all or most of the code not directly related to the algorithm or technique in question avoids their frustration in having to attend to windowing, interaction, and rendering details that are ancillary to the problem at hand. For the most part this also results in short lab exercises that students are able to complete in the hour following presentation of the theoretical material, providing an immediate sense of accomplishment and ownership of the material.

Developing graphics course content that effectively makes the connection between theory and practice, using the approach described here, is no small task. It takes considerable instructor time to carefully prepare an integrated set of lecture notes, handout materials, and scaffold code designed to allow students to reasonably complete an algorithm implementation in one lab period. But it appears to be worth the effort: while perhaps not enthusiastically embracing the theoretical underpinnings of key graphics techniques and algorithms, undergraduate students are at least able to apply theoretical knowledge to create correct implementations. These conclusions are based on informal observations however. Quantitative measures of the effectiveness of the approach are suggested.

References

[1] Foley, J.D., Van Dam, A., Feiner, S. K., Hughes, J.F., Phillips, R.L., 1994. Introduction to Computer Graphics, Addison-Wesley, New York, pp. 328-343.

[2] Foley, J.D., Van Dam, A., Feiner, S. K., Hughes, J.F., 1995. Computer Graphics: Principles and Practice in C (2nd Edition), Addison-Wesley, New York, pp. 478-491.

[3] Computing Curricula 2001 Computer Science (CC2001). The Joint Task Force on Computing Curricula: IEEE Computer Society and Association for Computing Machinery. Dec 2001, http://acm.org/education/curric_vols.