# Teaching Statement of Jack Tumblin

**2006, Northwestern University, EECS Department**

Fast computer graphics hardware is now available almost everywhere, from servers to laptops to cell-phones and PDAs. However, most engineering students and researchers still regard computer graphics programming as an exotic, time-consuming and difficult task, suitable only for large commercial software products such 3DStudioMAX, MATLAB plots, Maya or Blender modeling software, Renderman, Photoshop, Origin Graphing, Illustrator and Premiere. I disagree, and I have designed a series of Computer Graphics courses to demonstrate that any student or researcher ready to write a problem-solving program in a modern language (C/C++, Java, Python, etc.) is also ready to enhance and extend its usefulness with interactive 2D and 3D graphics, drawings, and visualizations, once they understand the underlying principles involved. The increasing success of these courses in terms both of enrollment and CTEC evaluations has proven the utility of my approach.

The two core graphics courses that I have designed and that I teach (CS351: Introduction to Computer Graphics; and CS395/495 (CS352) Intermediate Computer Graphics) provide the knowledge, give students 'starter code,' practical experience gained from projects, the ability to debug, and a clear understanding of how to move beyond these libraries. As well as providing particular skills, Computer Graphics can also teach particular ways of thinking. I see classroom teaching and course development as perhaps the best way to help the NU community to adopt the fundamental problem solving abilities available through interactive 2D and 3D graphics, drawing and visualization.

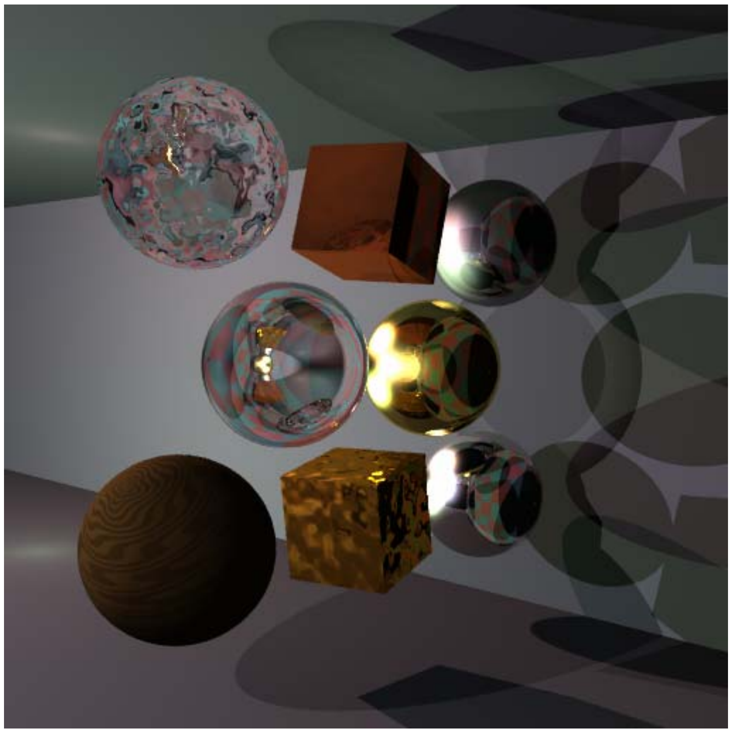
**Figure 1: CS352 Advanced Computer Graphics, Winter 2005; student Eric Russell's ray-tracer exhibits shadows, texture, antialiasing, chromatic aberration (colored fringes on glass) 'faked' caustics, Perlin Noise and more. One of two project assignments in the course, we follow this with particle systems.**

## Essential Approaches:

I'm still surprised by how much I like to teach classes, by how much I've learned doing it, and by how much more I am still learning and improving after 5 years. I have concentrated on basic courses that help attract more students to Computer Science. For many students, I have found just the right balance:

> **"Works wonders in explaining complex programming ideas...**
> **"One of the best profs I've had at NU in terms of human quality!"** [CTEC] CS110 Spr2006
>
> **"This is one of the few classes where not only do I KNOW more than I did when I started, but I can practically DO more as well. I'll use the concepts I learned in this class throughout my career."** [CTEC] CS 351, Fall 2005

and in 2005-6, my CTEC average was 4.6 / 6 (1==very low; 6==very high).

Most students with a typical undergraduate course load confront a jumbled torrent of information that no ordinary person can absorb; faculty says "Here, take this: a whole gallon of the world's very best paint! Don't waste it! (and I'll give you a bucket for it at the end of today's lecture)". Too much of undergraduate education sticks with us about as well as paint dumped in our hands—we get drenched in it, we'll always recognize the color and feel of it anywhere we see it, and we'll know a book or two where we can always find more, but we wash off most of our course materials before graduation day. How can we get the most useful information across to students in the most memorable, applicable ways?

My best answers are still evolving, but my key strategies in these courses are:

1) **Teach Debugging Strategies as Thoroughly as Programming.** Surprisingly many 'experienced' engineering students, even CS students, have poor debugging skills, backup, and version-control strategies that can fail catastrophically for larger programs. Accordingly, I teach simple safeguards to *all* my students, not just the beginners; for example, if they e-mail their source code to themselves every day, then even a stolen laptop will not destroy their work. Beginning students are especially vulnerable to frustrations and discouragement in debugging, and as frustrations grow they resort to increasingly random changes to their programs, often hopelessly compounding their errors and sense of helplessness. To combat this, I give *every* class some hands-on instruction in debugging (even in advanced classes and seminars), and reinforce it with short handout (see file: **Tumblin2.02_DebugAdvice03.pdf**) giving polished guidelines I've developed over several years. Debugging instruction has been extremely effective in eliminating over-ambitious student projects that 'crash and burn', heartbreaking file losses, and programs that worked the week before due dates but don't work days later. It reduces dropout rates, and seems to greatly increase student happiness and confidence. Projects are now far more ambitious; I've had CS student tackle fluid dynamics and finite-element re-meshing with happy outcomes under very tight time constraints.
2) **Studio-Style Instruction, Free-form Projects, and 'Starter Code'.** Lectures are boring for most students most of the time, and in computer science we're often trying to teach skills students can practice at their desks. Bored students at Northwestern skip classes and fall behind. Early on, I found that the more I prepared written notes and slides for lectures, the less the students needed (or appreciate) the lectures those notes were intended to illustrate. Instead, they'd rather read lecture notes and PowerPoint slides before class, discuss them in class, and spend that class time resolving any confusions. We try other examples and try exploring a broader set of related ideas, and *expand* on the contents of PowerPoint slides and written lecture notes. Whenever practical, we collaboratively write programs to test ideas. This 'studio style' approach makes projects more relevant, and I find that the least-restrictive assignments ("do something that looks amazing using this graphical technique") often produce the most ambitious and satisfying results; I've dropped my step-by-step 'dictatorial' early assignments entirely. To help student efforts, I provide starter code to act as 'scaffolding' for students to build graphics projects. Freed from the usual irrelevant difficulties of getting the first colored pixel or rendered triangle on-screen, students get to spend more of their project time experimenting with ideas and learning course content.

## Accomplishments:

To this end, I have re-designed or developed a new course almost every year since my arrival at Northwestern, for a total of five courses in five years.

- In my first year (2001-2), Ben Watson taught CS351 "Introduction to Computer Programming," and we jointly designed and taught a research seminar (**Winter 2002: Advanced Computer Graphics**) to broadly survey current research work; I continued the course the following year (Winter 2003), revising and improving it.

- Over Winter and Spring 2002 I also designed a wholly new course on **Image-Based Modeling and Rendering (IBMR)** methods, combining conventional photography with in-depth study of projective geometry, finding homographies by DLT, 3D warping and re-projection. I refined this course and taught it again in 2004 and 2005, modifying its contents towards computational photography topics, improving the projects and starter code substantially.
- In my 3rd year (2003-4) we **revised the graphics curriculum** to include Bruce Gooch, planning a 3-course core curriculum for the following year. To fit last-minute schedule constraints,
- I **taught both the introductory (CS351) and the intermediate (CS352) graphics courses** in my 4th year (2004-5). Despite the short preparatory period, I substantially revised both CS351 and CS352. I reorganized 351 into a 4-part project-oriented course, moving away from its largely lecture-driven previous format, re-writing and updating course materials as we went, and using the old book sparingly (Fall 2004 CTECs confirmed the old book was widely disliked). At the same time I designed a smoothly-compatible follow-on syllabus for CS-352.
- I converted both courses to follow a much more interactive, studio-style instructional format, giving intentionally broad assignments to encourage more imaginative and ambitious work. The general form was: “using these tools, show me something amazing,” and I was delighted with the range of projects; from fluid-dynamic-driven flags flying in the wind, to geometric model simplifications of dendritic crystals formed in cooling metals.
- In my **5th year**, greater preparation time permitted substantial improvements, and **I again taught both CS351 and CS352**; instituting a new, much better book, a refined syllabus, and incorporation of student suggestions on teaching and project guidance. I completed new ‘starter code’ to jump-start student projects, and refined the course notes and outlines as well. CTEC scores suggest these improvements have been very effective at increasing students’ knowledge, and their enthusiasm for the course.

I have also contributed substantially to the McCormick and CS undergraduate curriculum in each of my 5 years at NU by teaching **CS110 “Introduction to Computer Programming**”. This course, intended for non-majors with no background in computer programming at all, is made more challenging by a split enrollment, as it also attracts incoming CS students who wish to strengthen their grasp of programming basics. When I arrived in Fall 2001 to teach the course, I found an aging book on C, an obsolete and very slow, old 2D drawing package used in an ambitious 4-stage progressive programming project, and a reputation as extremely demanding for non-CS majors. I first worked to re-organize the course notes, and then the syllabus and project assignments to provide more introductory materials and explanation, lowering entry barriers for non-majors. I did my best to supply any missing definitions and explanations, and remove any assumptions or ‘jargon’, and devised a new approach to lead students into pointers with minimal confusion and debugging difficulties. I spent a great deal of time with students after hours to learn more about the difficulties confronting them, and used this knowledge to improve the course organization and instructional materials.

With instructor Vana Doufexi, we **chose a newer, better book for CS110**, streamlined the project requirements, and removed linkages between project stages. In the 3rd year, I guided an undergraduate student project to develop **an OpenGL-based replacement for the old CS110 drawing library**. The new library is self-contained, copiously commented and well explained. It greatly reduced CS110 students’ difficulties in starting their projects, improved their performance, provided additional instructional material on organizing programs into modules, and enabled instructors to easily extend the library as needed. I also adopted this library for use in the first project of CS351, with very favorable comments from students; these students can make pictures in the first week of class.

As my CTEC scores and comments will show, I have built a reputation for lowering entry barriers to computer science, attracting high attendance to my CS110 sections, equalizing the numbers of men and women students. I have tried to ‘humanize’ the topic area with humor, clarity, tools, and practical strategies everyone can use.  I work to bolster student confidence in their abilities, to realize that with persistence they can tackle any topic; nothing is out of reach, and to teach genuinely applicable skills.

## Goals and Plans

With the unexpected departure of both Ben Watson (2005) and Bruce Gooch (2006), I am fortunate to be well-prepared for both of our core graphics courses (CS351,CS352), continuing the strong EECS offerings in this area.  I would also like to teach a practical course on computational photography once we regain another graphics faculty member to help with our existing courses.   In addition, over the next year I would like to prepare for a major change in CS110, shifting instruction from the C programming language (with preparations for C++) to instruction in Python.  As a more modern, weakly-typed language, Python is much more forgiving for novice programmers, much more appropriate for teaching object-oriented problem-solving.  Python is an open-source project; it is well documented, well-supported, platform independent, links easily to C, C++, FORTRAN and JAVA libraries, and includes a simple and well-designed development environment that encourages student experimentation. CS faculty discussed such a move last year, with widespread positive opinions.

With the EECS merger, I also see great opportunities to strengthen our curriculum by more closely integrating coursework for graphics (Tumblin), audio and music (Pardo), the Animate Arts program (Horswill), computer vision (Wu) and image- and signal-processing (Pappas, Katsaggelos). We share many instructional sub-topics (e.g. Fourier Transforms, wavelets, convolution, edge-finding, projective geometry, color science, etc.), and we might all benefit from some shared instruction on these topics.