COASTER: TEACHING COMPUTER GRAPHICS WITH A COMPREHENSIVE PROJECT (A WORK IN PROGRESS)

Robert R. Lewis
School of EECS; Washington State University, Tri-Cities; Richland, WA 99354
(509) 372-7178
bobl@tricity.wsu.edu

ABSTRACT
We describe *coaster*, a project intended to teach an introductory computer graphics class by means of a sequence of programming assignments. The assignments are incremental -- each one building on the previous ones -- and ultimately bring in most of the course content. We discuss the sequence of assignments and how they relate to the topics, how all assignments themselves share code (from the instructor's view), course policy strategies needed to maintain student progress, and (preliminary) results on course effectiveness.

1. INTRODUCTION

Computer graphics is a very demanding course for most Computer Science students. In addition to programming skill, it requires more mathematics, especially geometry and linear algebra, than most students encounter in any other CS course. To create realistic three-dimensional images, knowledge of physics -- especially the interaction of light and matter -- is also required. To complicate matters, most CS students take the graphics course some several semesters after they took those non-CS courses to satisfy departmental requirements and have had little occasion to make use of that material since.

Like most CS courses, a graphics course should include some degree of programming, but unlike most CS courses, the successful completion of such an assignment is often self-evident -- the desired image is either visible or it is not -- and does not require the extensive testing needed by, say, a compiler or a web-based database application. (We draw a distinction here between the traditional graphics class that focuses on producing three-dimensional -- but not stereo -- images and a class on building a user interface.)

Another atypical aspect of a graphics course is that the student may show their assignments to family and friends, who can be impressed visually with no further explanation required by the student. This can be a strong motivation to the student.

One drawback, however, of this approach is that these assignments are often "one shots": The student gets one assignment to draw a mesh, completes it, and then moves on to one to draw a B-spline curve, and so on. While this allows a broad selection of topics to be covered within the term, this makes the individual assignments less impressive and therefore less motivational.

In this paper we will discuss an alternative approach, where students build a series of ten assignments, each one evolving from the previous one by adding features they learn about in the textbook [4] and corresponding lectures.

For this to work, the overall project has to be carefully chosen for its comprehensive use of graphics topics and the lectures scheduled around its most logical

development path. The overall project in the class we describe here (as a work-in-progress) was a first-person simulation of a rollercoaster called *coaster*.

Students did their work in C++ using the OpenGL graphics library, the GLUT utility toolkit, and, in the final assignment, the GLSL shading language. The textbook [4] covers all of these. By restricting the libraries to these alone, code was portable between Linux, MacOS, and (using MinGW [2]) Windows platforms.

2. COURSE CONTENT

Figure 1 shows views of the individual programming assignments (hereafter, "PA"s). The overall content of the course is not substantially different from that of other computer graphics courses, but each PA is linked to a set of one or more traditional lecture units with a balance of implementational, algorithmic, and mathematical topics.

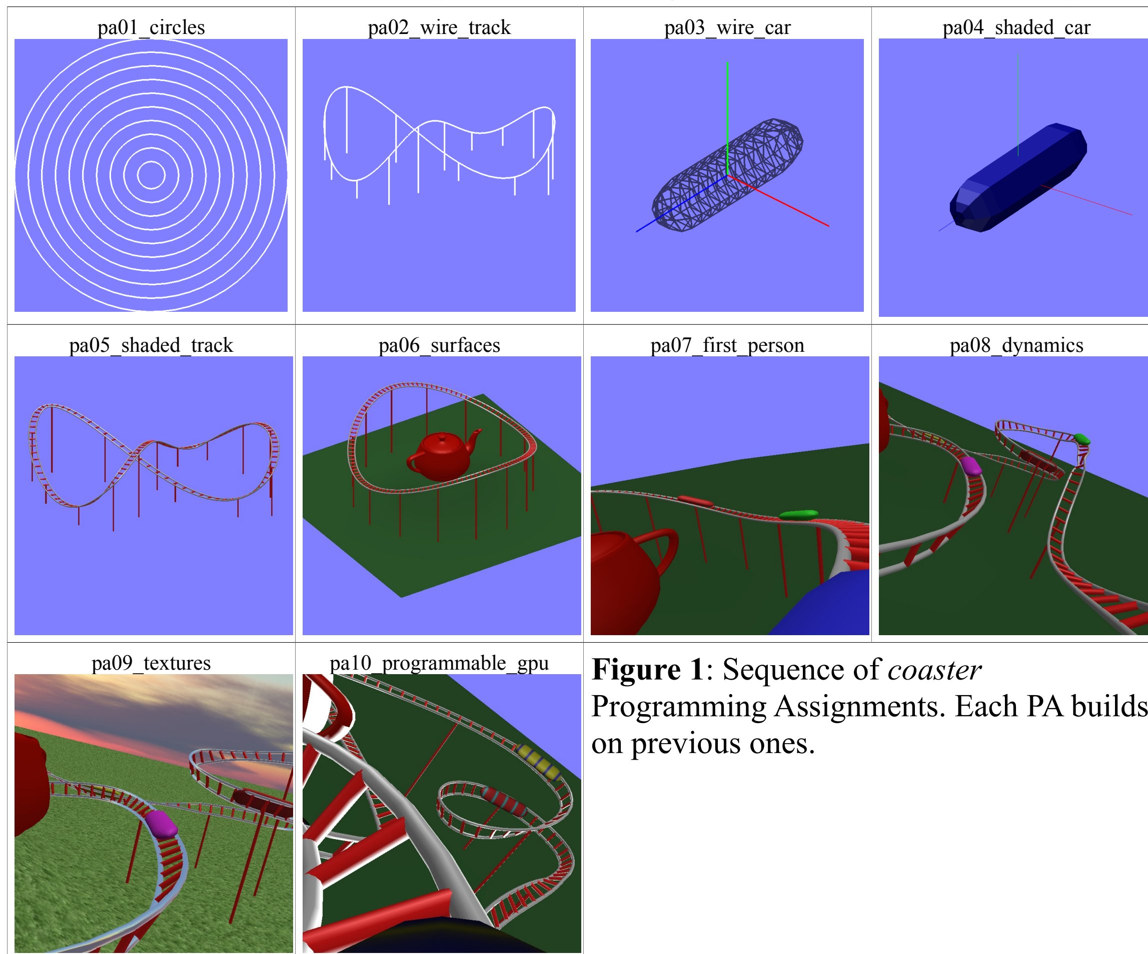

Figure 1: Sequence of *coaster* Programming Assignments. Each PA builds on previous ones.

Those projects and units are:

pa01_circles: Draw a regular polygon that, for a given pixel resolution, is indistinguishable from a circle. This is a "warm up". Lectures: an overview of computer graphics, the midpoint (Bresenham, essentially) line drawing algorithm

pa02_wire_track: Use a 3D trigonometrically-parameterized (easier to understand than a spline) curve to create a simple track for the rollercoaster, including track-to-ground support segments. Lectures: connections between computer graphics and geometry.

pa03_wire_car: Putting the track aside for the time being, create the body of the rollercoaster car. Lectures: regular mesh geometry and topology, creating solids of rotation, how to use triangle strips to draw meshes efficiently

pa04_shaded_car: Draw the car again, but this time determine vertex colors based on lighting calculations (not using OpenGL's) and use either flat or smooth shading. Lectures: the use of vectors in graphics, computing normals from grids, Gouraud shading, the physics and mathematics of reflectance

pa05_shaded_track: Using the same curve that was used for the wire track, create a more realistic, 3D track with extruded tubes for the rails, the ties, and the support structure. Lectures: extrusion and Frenet frames.

pa06_surfaces: Replace the trigonometrically-parameterized track curve with a B-spline (permitting creative track designs) and use the provided Newell [3] teapot (Bézier surface) data (as meshes) to render a teapot in the middle of the scene. Lectures: parametric curves and surfaces

pa07_first_person: Use transforms to allow the user to "ride" in a moving car (at a constant speed), controlling their position and (default) gaze direction. Lectures: modeling and viewing (orthographic vs. perspective) transforms, clipping algorithms

pa08_dynamics: Incorporate physics to produce a more realistic ride experience, including acceleration/deceleration and banking. Also replace reflectance calculations with calls to OpenGL for performance speedup. Lectures: rollercoaster physics, OpenGL lighting and material properties

pa09_textures: Apply a texture to the ground and use a "skybox" to create a surrounding view of the distance. Apply an environment map based on the same skybox to the rails so that they look like chrome. Lectures: texture and environment mapping, mipmaps

pa10_programmable_gpu: Replace OpenGL lighting (i.e. the fixed GPU program) with custom vertex and fragment shader (GLSL) programs. Also replace CPU-located vertex data with GPU-located vertex buffer objects. Lectures: GLSL

3. CODE ORGANIZATION

coaster consists of just over 14,000 lines of C++ code and several auxiliary data files (mostly textures) contained in the top or "root" directory (hereafter, "root") of the source code tree. Beneath the root are 20 subdirectories, two for each PA: a "template" subdirectory and a "solution" subdirectory. The contents of those subdirectories are managed automatically: Each PA has a manifest (contained in the root's "`Makefile`") that lists those source and auxiliary files that belong in both the template and solution subdirectories for that PA.

In general, all instructor code modification is done in the root. *make* dependencies are tracked, so that a change to a file in the root causes an update in all PA directories that use it (although it's perfectly acceptable -- and sometimes necessary -- to regenerate all subdirectory contents from scratch). Here, "update" means that the file in the root is passed through a filter that extracts only that code used in an identically-named file in the PA's template or solution subdirectory.

We will describe filtering in detail later, but by effectively keeping the code for all ten PAs in the root, maintaining those 14,000 lines of code automatically maintains a total of over 90,000 lines of code in the subdirectories. Furthermore, bug fixes that affect

multiple PA's are propagated automatically.

When a PA is made, students download from the course web page a compressed *tar* archive (a "tarball") that contains the source and auxiliary files from that PA's template directory together with a compiled (for Linux) executable from its solution directory. Students therefore see a running version of what they're supposed to accomplish for each PA, although we recommend that grading allow room for personal creativity, especially in the later PAs.

The template files contain comments that students follow to fill in the "blanks" -- regions that require code -- in various methods and data structures. Generally, header files (and therefore class definitions) do not contain blanks. The instructor creates those blanks within methods (usually) which require the student to use the knowledge of both mathematics and OpenGL they need to acquire. In the solution files, those blanks are replaced with working code, which students never see (see below).

4. CODE FILTERING

As we have seen, one source ("`.cpp`" or "`.h`") file in the root can give rise to as many as 20 files in the subdirectories, each of them a distinct result of varying the parameters of the filter it is passed through. We choose to implement "parameters" as a set of C macro preprocessor (*cpp* -- not to be confused with the C++ compiler) tokens which may either have specific values or may simply be defined or not defined. *cpp* directives such as "`#if`" and "`#ifdef`" embedded in the source file can control exactly what code the filter generates.

It would be possible to use the C++ compiler, which of course understands *cpp* directives (indeed, it may use *cpp* directly), to generate the subdirectory file using the "`-E`" command line option, but the trouble with doing this is that it will expand <u>all</u> macros as well as transitively insert the contents of all "`#include`" directives, resulting in a file that is large and unreadable. This is a serious drawback for files in template subdirectories, which must be readable by students.

Fortunately, there is a solution. The open source program *unifdef* [1] acts like a selective *cpp*: It will only expand macros that are specified on its command line. This makes it the ideal filter for our purposes.

Figure 2 shows how filtering works. It contains a section (one method) of a *coaster* source file as it would appear in the PD beside its four filtered template and solution versions. Prior to pa08_dynamics, the code is omitted. When compiling pa08_dynamics, the upper template and solution code is generated. For all assignments after pa08_dynamics, the lower template and solution code is generated. Here, then, there are four different code sequences generated. Some code gets generated more often than this, while some code is used throughout all projects unchanged.

5. COURSE POLICIES

A great problem in incremental project courses like this one arises when students fall behind. Students need to have at least a marginally-working PA completed before moving on to the next one. Getting stuck on one PA might mean not only a low grade on that PA but on all subsequent PAs. To minimize the impact of falling behind, we chose several unusual (for a graphics course, at least) course policies:

- Students may collaborate with each other on code and may even turn in identical

submissions. (A student's individual content knowledge is determined by exams.)
- Each PA has a due date, but late projects are accepted up until the end of the semester with only a 10% penalty.
- The instructor and teaching assistant(s) make themselves available to resolve problems with a very short turnaround time, including evenings and weekends.

One implication of these policies is that at no time can we distribute the "correct" solution to a PA, as students could simply turn that in as their (late) work. Even if those policies were not in place, however, distributing solutions would not be desirable if we expected to use the same incremental project in subsequent years.

original code (in root):

```
#if PA >= PA08_DYNAMICS

const double Curve::zMax(void) const
// returns the maximum z value along the curve
{
# if PA == PA08_DYNAMICS
    //
    //  ASSIGNMENT (PA08)
    //
    //  Move along the parametric curve looking
    //  for the maximum z value, which you
    //  return. Choose at least 1000 steps, but
    //  don't overdo it.
    //
# elif !defined(SOLUTION)
    //
    //  Copy your previous (PA08) solution here.
    //
# endif
# ifdef SOLUTION

    // fairly but not ridiculously large
    int nSamples = 1024;

    double h = 1.0 / nSamples;
    Point3 p = (*this)(0.0);
    double result = p.u.g.z;
    for (int i = 0; i < nSamples; i++) {
        Point3 p = (*this)(i * h);
        if (p.u.g.z > result)
            result = p.u.g.z;
    }
    return result;
# elif PA == PA08_DYNAMICS
    return 0.0; // replace
# endif
}

#endif // PA >= PA08_DYNAMICS
```

template for pa08_dynamics:

```
const double Curve::zMax(void) const
// returns the maximum z value along the curve
{
    //
    //  ASSIGNMENT (PA08)
    //
    //  Move along the parametric curve looking
    //  for the maximum z value, which you
    //  return. Choose at least 1000 steps, but
    //  don't overdo it.
    //
    return 0.0; // replace
}
```

solution for pa08_dynamics:

```
const double Curve::zMax(void) const
// returns the maximum z value along the curve
{
    //
    //  ASSIGNMENT (PA08)
    //
    //  Move along the parametric curve looking
    //  for the maximum z value, which you
    //  return. Choose at least 1000 steps, but
    //  don't overdo it.
    //

    // fairly but not ridiculously large
    int nSamples = 1024;

    double h = 1.0 / nSamples;
    Point3 p = (*this)(0.0);
    double result = p.u.g.z;
    for (int i = 0; i < nSamples; i++) {
        Point3 p = (*this)(i * h);
        if (p.u.g.z > result)
            result = p.u.g.z;
    }
    return result;
}
```

subsequent templates:

```
const double Curve::zMax(void) const
// returns the maximum z value along the curve
{
    //
    //  Copy your previous (PA08) solution here.
    //
}
```

subsequent solutions:

```
const double Curve::zMax(void) const
// returns the maximum z value along the curve
{
    // fairly but not ridiculously large
    int nSamples = 1024;

    double h = 1.0 / nSamples;
    Point3 p = (*this)(0.0);
    double result = p.u.g.z;
    for (int i = 0; i < nSamples; i++) {
        Point3 p = (*this)(i * h);
        if (p.u.g.z > result)
            result = p.u.g.z;
    }
    return result;
}
```

Figure 2: Example of Code Filtering. The code above, a method needed for a dynamic rollercoaster, is filtered into each of the four results shown on the right. (Some cosmetic editing has been done due to publication constraints.)

6. RESPONSE

On the basis of the first offering of this class, results are promising. Of the 23 students who completed the first PA, 18 remained in the class and 13 of them completed all PA's. (It was not necessary to complete all assignments to pass the class.) This compares favorably with pre-*coaster* attrition rates. (As we said above, computer graphics is a very demanding course.) Despite the fact that collaboration was permitted, it appears that only three pairs of students worked together.

The class was taught to two student populations: one local and one remote (via closed-circuit television and shared LCD display). Local student evaluations were uniformly positive ("I learned far more with this approach"), while remote student evaluations were highly variable (from "a great learning tool" to the proverbial "worst class I ever took"). We believe the latter is due to the fact that owing to budget cutbacks, there were no "live" teaching assistants at the remote site and our email and in-person visits were not sufficient to provide the help students needed. Local students, on the other hand, were able to bring their laptops to our office, which allowed us to resolve problems very quickly (while at the same time teaching graphics one-on-one, of course).

A notable -- if subjective -- indication of the success of *coaster* is in the quality and number of questions that it elicited from students. Students were asking important questions about meshing, illumination, geometry, textures, splines, and the like that their predecessors had seldom been motivated enough to ask before. We believe it was because these were things they *needed* to know to make their PAs work, and it was exciting.

7. CONCLUSIONS AND FUTURE DIRECTIONS

We believe *coaster* to be a more effective way to teach computer graphics than the traditional approach, but it requires a high degree of interactivity between students and instructor(s). (This is a good thing.)

Anticipated future improvements on *coaster* include

- tools to improve the code organization. In particular, we would like to analyze the changes from one PA to the next to make sure the necessary topics are covered in lecture with the appropriate weight and to match the time allotted for a student to make the necessary modifications for a PA to the effort (lines of code, at least) involved in doing it.
- increasing the emphasis on GPU usage and texturing, possibly at the expense of removing less important features such as environment mapping
- support for handheld and embedded graphics (via OpenGL ES)

We are also seeking technology that will allow more of a "look-over-the-student's-shoulder" approach with remote students (working on their laptops) and a textbook that more closedly matches our syllabus.

8. REFERENCES

[1] Finch, T., "unifdef -- selectively remove C preprocessor conditionals," dotat.at/prog/unifdef, retrieved 3/21/12.

[2] MinGW team, "MinGW | Minimalist GNU for Windows," www.mingw.org. retrieved 3/21/12.

[3] Torrence, A., "Martin Newell's original teapot," Article No. 29, ACM SIGGRAPH 2006.

[4] Wright, et al., *OpenGL SuperBible* (5th ed.), Addison-Wesley, 2011.