A CS1 Course on Media Computation
Working Document Version 3

Mark Guzdial

May 1, 2002

CS1321 *Introduction to Computing* is a course that seems most successful aimed at computing professionals: Students who have chosen to major in a computing-related field and who want an intensive introduction to computer science. However, many students on campus want an introduction to computing that prepares them to be professionals in an increasingly technological society. The focus for these latter students is on applications of the computing, with computer science concepts appearing as tools for enabling the applications.

This working document proposes and describes a course that we're currently referring to as CS1315 *An Introduction to Media Computation.* The focus for this course is on the computer as a tool for communication, through manipulation of digital media. Teaching the skill of programming is an explicit, high-priority goal for the course, and it is taught in the context of creation, transformation, and manipulation of digital media.

CS1315 will be offered as a *three* credit hour course: 3 hour lecture, and 1 hour required recitation.

The design in this document is based on a review of CS1321, discussions with many, the COE 1361 curriculum[1], the "Imperative-first" model of a CS1 (introduction to Computing) course as defined by the *ACM/IEEE Computing Curriculum 2001*[2], and research literature on CS1 and novice programming. This document is meant to be a living document, frequently being updated in response to more discussions and more elaborated requirements and objectives for this course. This version is based on discussions with many faculty in various units, as well as student feedback via on-line surveys[3].

Contents

[1]`http://isg.ce.gatech.edu/classes/200108/cee4803/syllabus.asp`
[2]`http://www.acm.org/sigcse/cc2001/`
[3]`http://coweb.cc.gatech.edu/guzdial/25` and `http://coweb.cc.gatech.edu/cs2340/2147`

1 Description of the Course

CS 1315—An Introduction to Media Computation is an introduction to programming in the context of computer as a media creator and transformer.

The core ideas of the course are:

- All media are being published today in a digital format.
- Digital formats are amenable to manipulation, creation, analysis, and transformation by computer. Text can be interpreted, numbers can be transformed into graphs, video images can be merged, and sounds can be created. We call these activities *media computation.*

- Programming a computer is our most flexible tool for manipulating digital media. Knowing how to program becomes a communications skill. There are data structures that can be used for manipulating digital media (e.g., arrays, matrices), and there are algorithmic structures that support media manipulation.
- Programs can get large and cumbersome. Abstraction is our tool for managing program complexity and allowing programs to become even larger yet more flexible.
- However, the computer has limitations. There are some programs that cannot complete or cannot complete in our lifetime.

Students in this course will write programs to do media transformations, creations, and manipulations. We will also talk about digital media and what it means for media to be digital. Some of the content of the course will necessarily be on the media itself, e.g. that a video can be thought of as a collection of frames that are two-dimensional arrays of pixels, that sound can be thought of as a composition of sine waves, etc. Algebra and trigonometry will be necessary pre-requisites to this course.

1.1 Contrasting CS1315 with CS1321

A description of a sample offering of CS1315 appears later in this document. A contrast with CS1321[4] is useful.

The core idea of CS1315 is to focus on creating concrete artifacts that motivate the learning of abstractions, which come later in the semester. We will cover much of the same programming and *computer literacy* content as CS1321, plus content on how media are constructed plus some limited Computer Science content. For example:

- CS1315 will leave out the content on non-list/array data structures (e.g., Trees, BST)
- CS1315 will leave out sorts, but will include some material on searches.
- Some topics on abstraction and theory (per the above course issues) will be included in the class, but at the end. We want students to be doing interesting things with the computer and understanding what they're doing, *then* introduce abstraction as a solution to problems they may be noticing and introduce theory as defining the limitations of what these programs they're building can do.
- CS1315 will introduce Java and MATLAB (in lecture, not just in recitation/labs) to point out how the concepts introduced in class appear in these other, more common languages.

Overall, we want to go at a slower pace with a concrete, multimedia content focus and a strong focus on really learning to write real programs that the students understand.

[4]`http://www.cc.gatech.edu/classes/AY2002/cs1321_spring/reading.html`

The goal is **NOT** to *replace* CS1321! With its greater emphasis on abstract, complex algorithms, and advanced data structures, CS1321 has a role with our own majors and with majors from other units that desire a greater emphasis on computer science. CS1315 is more appropriate for students who plan to *use* computers, but want to understand what they're using and how Computer Science may inform their work in their own discipline.

That said, I would like to see this course be an entrance into Computer Science for students who might not have thought about CS as a career. In general, I want CS1315 to be an *attractive* class. It may also serve to make Computer Science attractive, too. There are several options here:

- CS1315 may just be a terminal CS course. If students want to go on to computer science, they can either take CS1321 or COE1361.
- We could think about a *CS 1316* (one to two credit hours) which might lead into CS1322. CS1316 might include more on abstraction, more on data structures, and more on Computer Science overall.
- An alternative approach may be to think about CS1316 as a form of CS1322 that follows in the CS1315 footsteps, so that CS-aimed students can merge into the mainstream during the second year, at CS2130.

This course would integrate well with the various content production courses in LCC's *New Media Center*. I would very much like to have them (and others) involved in setting up the content goals for the course.

1.2 Goals for the Course

The course is designed to be an introduction to computation for non-majors and to provide an alternative entrance into computer science for students who may find the current courses impossible. The point of the below goals are to be met eventually, and to design the course in order to achieve them, but not to expect them to be met in the *first* instantiation of the course.

1.2.1 Demographic Goals

I want to define some hard, quantitative goals for the course, along these lines:

- A drop-failure rate of less than 15%[5].
- The enrollment for the course will be at least 30% female, and the drop-failure rate will be no different for females then males. It is an explicit goal of the course to have assignments that support creativity (which the AAUW report called for as to make CS classes more attractive to female students [AAUW, 2000]), that are concrete and whose application is clear [Margolis and Fisher, 2002].

[5]It's not unusual for CS1 classes to have failure rates in excess of 30% [Roumani, 2002].

- Academic misconduct cases brought against less than 5% of the students, even while using "CheatFinder" (and its descendants). Our plan for addressing this (described in the next section) is to have the majority of assignments to be explicitly collaborative with a handful of assignments to be explicitly individual efforts[6].

1.2.2 Learning Goals

Some of the learning goals for the course will include:

- Students will create several real, concrete digital media artifacts, to create success that encourages retention [Margolis and Fisher, 2002] and learning [Resnick et al., 1996].
- Students will be comfortable using core computer tools: Microsoft Office, a web browser, and some email client.
- Students will understand the benefits and limitations of digitizing media.
- Students will meet ACM/IEEE CS1 learning goals, especially in algorithms, data structures, and programming.
- There are several published benchmarks for performance in CS1, including Soloway's Rainfall Problem [Soloway et al., 1982] and McCracken's Calculator Problem [McCracken et al., 2001]. Every term that CS1315 is offered, students will be asked to complete one of these problems (and others of a similar complexity that we will invent), with a goal of directly measuring learning in CS1315 in comparison with other CS1 courses.

  This last is a significant goal. There are lots of non-major CS1 courses (e.g., [Zimmerman and Eber, 2001] [Marks et al., 2001]), but there are *no* published papers demonstrating programming skills in non-major CS1 courses. This will be an important accomplishment when we achieve it.

1.3 Assignments of the Course

The idea for the assignments are to take less time than the CS1321 assignments, but while still going in-depth into how software works, giving students ample time to learn to program, providing good assessment opportunities, and providing time to work together. Here are some of my thoughts along these lines:

Here is the current plan:

- (15% of grade) I'm planning for bi-weekly brief (15 minute) quizzes whose focus is on *understanding* a given program, e.g., "At this point in the execution, what would the value of X be? What would be on the screen?" If quizzes are on Thursdays, a *pre-quiz* would be

[6]The plan for including collaborative activities is based on research supporting 'pair-programming' effects on retention in CS1 [McDowell et al., 2002] and performance [Williams and Kessler, 2000] in CS1, and on the positive impact of social learning situations for female students in CS1 [Margolis and Fisher, 2002][Barker et al., 2002]

distributed the Tuesday before. The pre-quiz would have very similar problems that students will be encouraged to work on *together.* The pre-quiz answers will be *written*, not submitted electronically[7] The pre-quiz will count for 20 points and the quiz itself would be out of 80 points, for a total 100 points per quiz.

- (15% of grade) There will be a bi-weekly *required* "lab" activity (opposite of the quiz weeks). The lab activity will be on computer applications use. The use of computer applications will be the focus of the recitations, and the lab activity will ask students to practice the applications material covered in recitation. "Labs" are quoted because the actual activity will probably take place on the students' own computers, for two reasons. First, CS1315 would place an enormous drain on any lab facilities on campus. Second, the point of teaching applications is for the students to be able to use them on *their own* computer that Georgia Tech made them purchase.
- (15% of grade) There will be weekly programming homework. Students will be *encouraged* to work on these collaboratively, perhaps with computer support for the collaboration [Guzdial, 2001].
- (20% of grade) I'm planning for four project assignments across the course of the semester, which are expected to be individual efforts. These will be checked with "CheatFinder."
- (35% of grade) I'm planning for two exams and a final.

1.4 Language for the Course

We are currently planning to use the programming language *Jython* for the course. Jython[8] is a variation of the Python[9] programming language which has been designed to be easily used by novices and to be a non-technical course. It's used extensively for Web programming, database programming, and "scripting" applications. Jython is a variation of Python written in Java—anything written in Java can be integrated with Jython, and Jython can be used for virtually anything for which Java can be used.

Originally we planned to use Scheme, but we've been conducting on-line surveys of students (e.g., `http://coweb.cc.gatech.edu/guzdial/25`) which suggest that non-majors perceive Scheme to be about "serious CS" and *not* about things that are important to them. Our original reasons for Scheme were the following.

- Scheme has a long history of success in CS1 courses, at places like Berkeley, MIT, Rice, and Brown.

[7]One of the problems that are being reported in CS1321 is (from students), "I type what you did in class, but it doesn't work!" Obviously, the students *didn't*, but probably because they didn't *see* all the detail of what the teacher typed: Non-programmers tend to gloss over quotes, commas, and case. By seeing what they *write* (not just copy-paste from a workspace), we can see what they are seeing—they can only write down what they see. This gives us a chance to focus on how they read programs and how they understand them.

[8]`http://www.jython.org`

[9]`http://www.python.org`

- Scheme allows us to be flexible with its syntax. We know a lot about what makes programming hard, and have known for a long time. For example, `If-Then` (conditionals) are hard for novices to learn [Green, 1977], and looping (iteration or recursion) is hard [Soloway et al., 1983][Miller, 1981]. We also know things that make it easier, e.g., providing a way to `break` out of a loop based on a conditional makes both easier [Soloway et al., 1983] and providing operations on sets and aggregates makes it easier to iterate before learning iteration constructs [Miller, 1974][Pane et al., 2001]. In Scheme, we *can* modify the language. We can use a construct like `(for-every x '(1 2 3 4 5))` before we teach more generic `for` or `while/do` loops or recursion.
- One of the high frequency errors that students make in programming is confusing precedence [Spohrer and Soloway, 1986]. In Scheme, this problem is reduced through use of parentheses to make precedence explicit.
- Scheme is interpretive: You can use it like a calculator to type in commands and get immediate feedback. We know that early success is important in learning [Blumenfeld et al., 1991], and we know that it's particularly important to encourage females in computer science ([Margolis and Fisher, 2002], p. 59). It's style of use is like MATLAB, which is important given the popularity of MATLAB in Engineering.

Jython has many of these advantages, and a couple more.

- It does have a flexible syntax. `for x in [1 2 3 4 5]` is similar to what we wanted to do in Scheme, and Jython includes structures like `break`. More importantly, Jython has a syntax that is designed for novices. For example, there is no `begin-end` or curly braces to define blocks—instead, indentation defines a block. The idea is "If it looks like a block, it *is* a block." Readability is important in Python and Jython.
- Jython is interpretive, so students can explore in the language.
- Jython *looks* like a traditional programming language — it doesn't have the parentheses of Scheme, so it has some face-level validity.
- Jython can be used for many tasks that students would recognize as useful. Python programming ability is a job skill that employers do advertise for.

We are also considering other languages, but find them less convincing:

- In none of MATLAB, C, or Java can we adapt the language as we wish and as the research suggests.
- All of MATLAB, C, and Java emphasize aspects of computer science referred to as "tedious" in the American Association of University Women report on women in computing [AAUW, 2000].
- C and Java are designed to emphasize speed and efficiency. While important goals, they are no longer the *most* important goals in most software development efforts, certainly not for non-CS-majors who write scripts and other kinds of supportive programs, and also not of many women in computing ([Margolis and Fisher, 2002], p. 56 and 120-121).

- MATLAB provides most of the kinds of multimedia supports we want, but with a difficult syntax. Java provides the multimedia supports, too, but requiring students to first master concepts like classes and `public static void main`. Neither are completely cross-platform, which is important for involving students who aren't interested in OS or speed/efficiency debates ([Margolis and Fisher, 2002], p. 65).
- Squeak directly would certainly be a possibility. However, it's syntax is quite complicated for rank novices[10].

1.5 Teaching Assistants

With the TA structures now implemented in CS1321 (after the review of the CS1321 Task Force), we plan to use a very similar structure in CS1315.

Teaching Assistants (TAs) will be hosting the recitation sections of CS1315, which *will* involve teaching new material—specifically, the application tools portion of the curriculum. We will want to cap the size of the recitations at 30, with a goal of *25:1* student:TA ratio. In addition, TAs will be responsible for grading, and more senior TAs will be involved in writing assignments, assignment solutions, and supervising/training TAs.

Because the goal of CS1315 will be to serve non-CS-majors, it's going to be important to have non-CS TA's involved:

- To reflect the concerns and perspectives of other departments,
- to serve as role models for younger students in the same discipline, and
- to help in creating assignments that meet the issues of other departments.

For all these reasons, we will aim to have 25% or more of our TAs be from outside the College of Computing. However, we will still want the majority of our TAs to be College of Computing students to provide a CS perspective.

As mentioned, we hope that CS1315 and CS1321 can share TA training, hiring, and organizational structure. Our standards (e.g., GPA standards, never been found guilty of academic misconduct, etc.) for TAs will be the same as CS1321.

1.6 Tour of a Sample Course

An elaborated syllabus is beyond the scope of this document, but a tour of what the class *might* look like would be helpful in explicating the story. I am currently thinking about the class as consisting of four stages.

[10] A brief example: The code `'abc' contains: $a ifTrue: [Transcript show: 'Has an A!']` actually wouldn't compile, because the string 'abc' doesn't understand the message `contains:ifTrue:`. Precedence is confusing in Squeak.

1.6.1 Stage One: First two to three weeks

The goal of this first stage is to get students programming things that they consider interesting and useful. The argument I'd like to make is based on their use of tools like Photoshop. I might show a before-and-after of an image processed by a Photoshop-like filter. Students could do this with Photoshop, but the application is expensive, you have to learn it, and you still don't really understand what happened, so if you want something more sophisticated or slightly different, you're out of luck. Or you can write these two lines of code (seen below). Of course, you have to know *how* to write these two lines of code, but once you do, you have a formidable tool in communicating what you want to communicate in the way that you want to communicate. You are not limited to what Adobe or Microsoft says you can say in the way that they want you to say it.

From the very beginning, lectures consist of demonstration to a great extent: 50-75% of class time is spent in the programming environment. Programming is a cognitive skill, and apprenticeship is an excellent way to teach cognitive skills [Collins et al., 1989].

In the first stage of the course, the core language constructs are introduced: defining variables and procedures (only a single user-defined procedure for all examples in these first weeks), function composition, simple `if` conditionals, and iteration as manipulation of a set. The language is introduced from the very beginning in the context of manipulating media.

For example, students might record themselves (using provided audio recording software), then play their sound back with `play(sound(‘‘my-sound.aiff’’))`. Variables might be introduced to hold some of those pieces:

```
my_sound = sound(‘‘my-sound.aiff’’)
play(my_sound)
```

To continue function composition, we might play the sound at different frequencies, e.g. `play_at_freq(my_sound` `440)`. The first example of iteration might be exploring the kinds of manipulations that a sampling keyboard does:

```
for freq in [110 220 440 880]:
   play_at_freq(my_sound, freq)
```

Another example might use image manipulation. Students might load in a picture with `my_pic = picture_file(‘‘me.jpg’’)`. Once students are told that a picture consists of pixels, pixel values might be changed by defining filters, not unlike Adobe Photoshop, with code like:

```
for p in pixels(my_pic):
   sethue(p, 2 * hue(p))
```

Note that we're explicitly making a choice here to focus on a function composition model of programming (e.g., `hue(p)`) as opposed to an object composition model (e.g., `p.hue`). Jython is flexible in supporting both models. The plan is to start with function composition because it can be used with *any* programming language, from Visual Basic to Java and MATLAB. Later in the class, we can offer the object notation (common in Java) as a way of doing the same kind of activity but in an object-oriented notation.

Recitations during this initial period will go over the critical infrastructure of the class: How to use email, how to use the Web, how to use the programming environment. In recitations, TA's will use the tools in live, projected displays to demonstrate the tools.

Student assignments in this period will mostly involve editing existing code. We want students to develop the skills of reading code, and modifying someone else's anything is always easier than starting from scratch. Pre-quizzes and quizzes will test student knowledge of their programs. For example, in the last example above, trace out the pixel hue values for a given set of sample data.

1.6.2 Stage Two: Up to Week Nine or Ten

The heart of the course is applying these basic skills in a variety of domains with only a handful of new programming concepts. We want students to feel that they are programmers, that they can do interesting things by combining concepts that they know well. We also want to demonstrate different mathematical concepts that may be useful in other disciplines in the context of programming. Some lecture time will need to be spent on media concepts (e.g., that a sound has a sampling rate, that a pixel has color values), but these are meant to be motivating and interesting.

The new computer science concepts to be introduced are:

- Arrays, in single and double dimensions. The above examples naturally extend themselves to single dimension arrays (e.g., `sound_buffer(my_sound)` could return the internal array of sound samples for the sampled sound) and doubly dimensioned arrays (e.g., referencing x and y values).
- Array-generation functions (really, list-generation functions). Iterating across x values of a picture might be `for x in width(my_pic)`
- Simple structures, e.g., the concept that a particular data element actually has several data associated with it. For example, a pixel has RGB or HSV values associated with it. A sound has a set of samples and a sampling rate associated with it. A MIDI note has a key and an instrument associated with it.
- Computer science concepts as they become useful. Somewhere during this stage, it's going to become cumbersome to copy-paste code to duplicate it. (We do want students to do this at least a few times.) That will be a natural time to introduce functional decomposition. Similarly, with all coding going on live, in lecture, bugs will undoubtedly creep in. Debugging techniques will be taught as they appear needed.

Lecture will cover a variety of media types, with new primitives introduced to make simple the manipulation of each media type. Research on novice programmers has found that students really don't have a problem with handling a variety of basic functions and figuring out which ones are needed (e.g., [Perkins et al., 1986][Spohrer and Soloway, 1985][Spohrer, 1989]). The problem is helping the students to understand how to use them in combination.

This actually becomes a human-computer interface problem, and there are solutions for such problems. By choosing primitives that are well-suited for students' tasks, we make it more like for the students to figure out what to use when [Pane et al., 2001].

There are lots of possible media examples to use in this part of the class. Some of the media examples we're considering are described below. The actual ones to be used in a given offering of CS1315 can certainly be dependent on instructor interest and student feedback. I would like for the course materials to include lecture and support materials (e.g., examples worked out, assignment suggestions) for *many* media types, so that a given instructor could easily pick the media types that interest her.

- **Sound**: Once we have the sampled sound as an array, we can shift its frequency by creating a new sound with only some of the original samples. If we take every other sample out of the original sound, we double the frequency. We can also change the volume of the sound by simply multiplying the samples by a constant (within the dynamic range).

  Once we have arrays, we can also start generating new sounds. We can generate a sine wave, then add sine waves (a technique called *additive synthesis*) to create wholly new sounds. (This example has been developed in an article from *Communications of the ACM* in press and available at `http://coweb.cc.gatech.edu/guzdial/17`.)

- **Image manipulation**: Besides Photoshop filters, we can create new pictures by rotating, magnifying, and shrinking a filter. Rotating a picture 90 degrees is fairly straightforward:

  ```
  new_pict := picture(width(my_pict),height(my_pict))
  for x in width(my_pict):
      for y in height(my_pict):
          old_pixel = pixel(my_pict,x,y)
          set_pixel(new_pict,y,x) = old_pixel  # Notice the x and y swapped
  ```

  Images can also be created using graphical primitives, like drawing lines, boxes, and circles. Again, this is still a matter of applying function composition and using the same computer science concepts.

  Combining these techniques, we can have students drawing graphics into images, like how Roger Rabbit is inserted into a movie with live actors.

- **Animations**: Animation becomes a simple matter of assembling a collection of images. We can provide functions for assembling sequences of images and playing them back. With animations, mathematical techniques such as interpolation can be used to fill in frames between key frames.

- **Movies**: Movies are now a sequence of images read in from a file. We can provide tools for converting MPEG movies into image collections. We can now explore concepts like implementing video effects by comparing between frames, modifying frames (e.g., inserting a graphical object), or computing differences between frames. Techniques such as *blue-screening* (e.g., where the weather announcer seems to be in front of the weather map) can be implemented without new CS concepts. If we have a background `my_background` the same size as a given frame `my_frame`, then replacing the bluish-parts (those colors within a `threshold` of a base blue color) can be replaced with the corresponding element from the background. (In the below example, we start to use the object-oriented notation, using the dot operator, as a transition from the function composition model.)

```
for x in my_frame.width:
    for y in my_frame.height:
        if (my_frame.pixel(x,y).color - color(‘‘blue’’) < threshold):
            my_frame.pixel(x,y) = my_background.pixel(x,y)
```

- **Text**: Text can also be seen as a medium, and its manipulation has often been used as a way to introduce procedures (e.g., [Harvey and Wright, 2001]). Procedures can be used to convert English into pig latin, to generate Haiku or other poetry forms, or to generate sentences randomly.

  ```
  noun = pick_random([Mark Dog Cat Fred])
  verb = pick_random([runs jumps sleeps eats])
  sentence = noun + verb
  ```

  More relevant to students, text-to-text translation is how we can automatically generate HTML, for reports or other sources.

- **Transforming Media: Text to Graphics**: Text has power as a medium for specification, and it's common to use text to drive other media, e.g., text programs themselves and text specification of graphics or sound. We might use text read from a file to define graphics to draw. For example, we might read two pairs of numbers to be start and end points (x,y) of lines, and then draw the specified lines. From there, it's an easy step to think about reading in values and plotting them by transforming the values into a graphing space.

  There are many Java charting packages, all of which are available from Jython. We might have students use these charting packages, as a way of doing sophisticated text-to-graphics translations, while also exploring how to reuse others' code.

Student assignments in this section of the course can actually be quite creative and open-ended. The specification of the assignment can define the media types and transformations to be applied, and students can define the details. Some examples:

- Magnify an image of your choosing four times. On the left side of that image, insert a copy of the image shrunk by half and rotated by 90 degrees counter-clockwise. On the right side, insert a copy of the image shrunk by half rotated by 90 degrees clockwise.
- Play a tune consisting of at least a dozen notes with at least two different instruments, where one instrument is a sampled sound of your choosing and the other instrument is synthesized by adding at least three sine waves at different frequencies and amplitudes.
- Record a sound of at least 10 seconds in duration, *but do not hard code the length of the piece!* Write a procedure to play that sound back and display three different images during the playback. You need to compute the duration of the piece by multiplying the length of the

sound buffer by the sampling rate. Display each image in succession *for the same amount of time*, and end the image display at the end of the piece.

- Find at this URL a provided movie of an actor working in front of a blue screen. Find an image of the same size (one of our provided ones, or take a picture yourself) and use it as the background instead of the blue screen.

Recitations in this section will focus on use of applications and on use of the development environment for the assignments. Students will be taught to use Microsoft Word, Microsoft PowerPoint, and Microsoft Excel (for both calculation and graphing). Additional tools might be introduced, and ties will be made between the programming in the class and the relevant media manipulations in the tools (e.g., "moving to back/front" in PowerPoint has a lot to do with the ordering of the graphics drawing primitives, which can be a useful example of statement ordering). TAs will also answer questions with example use in the development environment.

1.6.3 Stage Three: Up to Week Thirteen

Now that the students have done a good bit of (interesting) programming, we begin to introduce more advanced Computer Science concepts as a way of making programming easier and more robust. The question to students is how to handle lots of cases without writing code for each one, how to write code in fewer lines so that it takes less time, and how to write code that others can read and understand what you're doing (or you read others' code) so that they can use the same thing. We'll use the exact same media examples, and perhaps a few more sophisticated ones, to explicate the CS concepts introduced in this stage.

- **Recursion**: Recursion becomes another way of iterating, especially useful when manipulating lists of information, such as with text.
- **Functions as Parameters**: Treating functions as first class objects that can be passed around has a lot of applications in these kinds of examples. For example, in the image filtering examples, the "multiply-by-2" operation on the hue can be a function that is simply applied to all the pixels. Thought of that way, much more sophisticated filters can be created and encapsulated in procedures.
- **Interpretation**: Imagine in the text-to-graphics example if there were strings embedded in the text that gave commands like *line* or *circle.* A program that read in these commands and the numbers for the coordinates of those shapes could be assembled pretty easily using the concepts of this course. At that point, you're writing an interpreter, and Jython itself is only a more sophisticated version of that same concept.
- **Objects**: An object can be thought about as a structure that contains functions as well as data in an aggregation. Jython is naturally object-oriented, so it's a supported transition. As shown previously, we can think about objects as a way of doing the same kind of thing as we did previously with functions. Yes, you can do alot just with functions, but objects allow you to scale better. What if movie frame pixels and picture pixels are slightly different? Which would you be referring to when you write `pixels()`? What if you could separately have

`movie_frame.pixels()` and `picture.pixels()`? That's where the value of objects comes in.

- **Order of Algorithms/Complexity**: Processing across all the frames in a movie takes much longer than processing a single image which takes longer than processing sound. Why is that? Talking about the complexity of algorithms flows very naturally here. (And might, in fact, be easier to do back in Stage Two.)
- **What Computers Can't Do**: The complexity of algorithms can reach the point where the algorithm takes forever. The computer just getting faster has its limits when facing some of these algorithms, like trying to optimize across many variables. There are some problems, like analyzing other programs (see interpretation) that computers really can't do.

Student assignments will involve using these new concepts to manipulate media as previously. There will be an emphasis on creating the same *functionality* as in the Stage Two assignments, but using these new concepts.

Recitation will emphasize re-iterating the lecture concepts in a small group setting with plenty of examples and time for questions. These are highly abstract concepts. Students will need a lot of time to grok these ideas.

1.6.4 Stage Four: Last couple weeks

Some of our students may actually work in Jython — it is a serious programming language with commercial viability. However, some will want to work in languages like Java. We want the course concepts to transfer out into new contexts [Bruer, 1993]. In Stage Four, we explicitly aim at that transfer into useful contexts.

In lecture, we will introduce Java. Direct comparison will be made between the core programming concepts introduced in Jython (e.g., variables, procedures, function composition, structures, arrays, iteration and conditional constructs, recursion, objects) in the new language, which is made easy since Jython is close to Java in core concepts already. The goal here is not to introduce language constructs specific to Java (e.g., object references or constructors in Java). Rather, we want students to be able to do the *exact* same kind of programming that they were doing in Jython in this new, more common, but harder language.

Student assignments will involve doing similar kinds of programming as what they were doing previously, but in Java.

Recitation will focus on use of Java. In particular, installation, debugging skills, and practical issues will be the focus.

1.7 Comparison to Other Approaches

The approach that is closest to the one described here is the *Exploring the Digital Domain* text and course by Abernethy and Allen [Abernethy and Allen, 1998]. The multimedia content of CS1315 is very similar to the Digital Domain course, but the Digital Domain course is *without* programming.

Most other non-major or Scheme-based approaches to CS1 emphasize computer science concepts without a more concrete focus. The *Great Ideas in Computer Science* text and course by Biermann [Biermann, 1998] *does* introduce programming (in small pieces), but doesn't expect the students to program much. The Berkeley non-majors course is in Scheme [Harvey and Wright, 2001] and does offer some fun projects (like tic-tac-toe and Eliza-style pattern matching/interaction), but is mostly about CS abstractions. For example, the course talks a lot about meta-functions like `map` and `reduce` which are quite complex. The text currently used in CS1321 [Felleisen et al., 2001] emphasizes the design of programs, which could be useful for Engineering students. However, the content of the CS1321 text is clearly Computer Science abstracts: It's not clear how to connect the design of programs to any other kind of design.

There have been other multimedia-focused CS1 courses, mostly for non-majors (e.g., [Marks et al., 2001] [Zimmerman and Eber, 2001]). There was little emphasis on programming in those courses. Probably the closest approach to the one proposed here was the ACSE environment and class in the late 80's [Miller et al., 1994], where undergraduate students learned programming in the context of manipulating multimedia simulations in order to learn about science and programming. (A similar focus was used in Emile [Guzdial, 1995], but with high school students.) While the literature on ACSE says that students "built" a lot, we don't really know much about learning in this environment, either about programming or science.

This course will be unique. There are virtually no courses aimed at non-majors, with a focus on concrete applications, that also attempt to teach programming. If successful, it will be an important accomplishment for our students and for CS education.

2 Implementation Issues

This section describes the time frame and costs of this course.

2.1 Time Frame

I am planning this course to take on a serious number of students from the first offerings.

Here is the implementation time frame that I'm thinking about:

- **Spring/Summer 2002**: Start planning out details of the course, developing the programming framework and environment for Jython, defining lectures, and writing lecture slides and course notes. We want to involve non-CoC faculty in defining the course.

- **Fall 2002**: Working with one or two Senior TAs, begin creating assignment definitions and some instances of these definitions.

  Present the course as defined so-far to the various units and to the Undergraduate Curriculum Committee for approval as meeting Core Area B.

- **Spring 2003**: Offer a prototype of the course to 75-100 volunteer students.

- **Summer 2003**: Improve the course based on the prototype offering and train up a cadre of TAs for full-scale implementation of the course.

  We also create the organization of the course at this point: The analogue to the *CS 1321 "Machine"* that keeps creating assignments, training TAs, supervising TAs, etc.

  At this point, we would offer it to the Institute Undergraduate Curriculum Committee as a permanent course.

- **Fall 2003**: Offer two sections of the course, of 250-300 students each, with me teaching both sections.

- **Spring 2004**: Again offer two sections of the course, of 250-300 students each, but with someone else teaching one of the sections (faculty or instructor) to make sure that we've created a system that can be adopted and taught by others than the original developer.

References

[AAUW, 2000] AAUW (2000). *Tech-Savvy: Educating Girls in the New Computer Age.* American Association of University Women Education Foundation, New York.

[Abernethy and Allen, 1998] Abernethy, K. and Allen, T. (1998). *Exploring the Digital Domain: An Introduction to Computing with Multimedia and Networking.* PWS Publishing, Boston.

[Barker et al., 2002] Barker, L. J., Garvin-Doxas, K., and Jackson, M. (2002). Defensive climate in the computer science education. In Knox, D., editor, *The Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education, 2002*, pages 43–47. ACM, New York.

[Biermann, 1998] Biermann, A. W. (1998). *Great Ideas in Computer Science: A Gentle Introduction.* MIT Press, Cambridge, MA.

[Blumenfeld et al., 1991] Blumenfeld, P. C., Soloway, E., Marx, R. W., Krajcik, J. S., Guzdial, M., and Palincsar, A. (1991). Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational Psychologist*, 26(3 & 4):369–398.

[Bruer, 1993] Bruer, J. T. (1993). *Schools for Thought: A Science of Learning in the Classroom.* MIT Press, Cambridge, MA.

[Collins et al., 1989] Collins, A., Brown, J. S., and Newman, S. E. (1989). Cognitive apprenticeship: Teaching the craft of reading, writing, and mathematics. In Resnick, L. B., editor, *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, pages 453–494. Lawrence Erlbaum and Associates, Hillsdale, NJ.

[Felleisen et al., 2001] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2001). *How to Design Programs: An Introduction to Programming and Computing.* MIT Press, Cambridge, MA.

[Green, 1977] Green, T. R. G. (1977). Conditional program statements and comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50:93–109.

[Guzdial, 1995] Guzdial, M. (1995). Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments*, 4(1):1–44.

[Guzdial, 2001] Guzdial, M. (2001). Use of collaborative multimedia in computer science classes. In *Proceedings of the 2001 Integrating Technology into Computer Science Education Conference.* ACM, Canterbury, UK.

[Harvey and Wright, 2001] Harvey, B. and Wright, M. (2001). *Simply Scheme, Second Edition.* MIT Press, Cambridge, MA.

[Margolis and Fisher, 2002] Margolis, J. and Fisher, A. (2002). *Unlocking the Clubhouse: Women in Computing.* MIT Press, Cambridge, MA.

[Marks et al., 2001] Marks, J., Freeman, W., and Leitner, H. (2001). Teaching applied computing without programming: A case-based introductory course for general education. In McCauley, R. and Gersting, J., editors, *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, pages 80–84. ACM Press, New York.

[McCracken et al., 2001] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulletin*, 33(4):125–140.

[McDowell et al., 2002] McDowell, C., Bullock, H., Fernald, J., and Werner, L. (2002). The effects of pair-programming on performance in an introductory programming course. In Knox, D., editor, *The Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education, 2002*, pages 38–42. ACM, New York. Pair-programming (social) reduces attrition rates.

[Miller, 1974] Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies*, 6:237–260. Participants strongly preferred to use set and subset expressions to specify the operations in aggregate.

[Miller, 1981] Miller, L. A. (1981). Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215. Languages require iteration where aggregate operations are much easier for novices.

[Miller et al., 1994] Miller, P., Pane, J., Meter, G., and Vorthmann, S. (1994). Evolution of novice programming environments: The structure editors of carnegie-mellon university. *Interactive Learning Environments*, 4(2):140–158.

[Pane et al., 2001] Pane, J. F., Ratanamahatana, C., and Myers, B. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54:237–264.

[Perkins et al., 1986] Perkins, D. N., Martin, F., and Farady, M. (1986). Loci of difficulty in learning to program. Technical report, Educational Technology Center, Harvard.

[Resnick et al., 1996] Resnick, M., Bruckman, A., and Martin, F. (1996). Pianos not stereos: Creating computational construction kits. *Interactions*, 3(5):41–50.

[Roumani, 2002] Roumani, H. (2002). Design guidelines for the lab component of objects-first cs1. In Knox, D., editor, *The Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education, 2002*, pages 222–226. ACM, New York. WFD (Withdrawl-Failure-D) rates in CS1 in excess of 30

[Soloway et al., 1983] Soloway, E., Bonar, J., and Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860.

[Soloway et al., 1982] Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1982). What do novices know about programming? In Badre, A. and Schneiderman, B., editors, *Directions in Human-Computer Interaction*, pages 87–122. Ablex Publishing, Norwood, NJ.

[Spohrer, 1989] Spohrer, J. C. (1989). *MARCEL: A Generate-Test-and-Debug (GTD) Impasse/Repair Model of Student Programmers*. Ph.d., Yale University. YALEU/CSD/RR 687.

[Spohrer and Soloway, 1985] Spohrer, J. C. and Soloway, E. (1985). Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume March. Tucson, AZ.

[Spohrer and Soloway, 1986] Spohrer, J. G. and Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. In Soloway, E. and Iyengar, S., editors, *Empirical Studies of Programmers Workshop*, pages 230–251. Ablex, Washington D.C. Operator precedence errors were among the high frequency bugs observed in novice programs in a traditional programming language.

[Williams and Kessler, 2000] Williams, L. A. and Kessler, R. R. (2000). The effects of 'pair-pressure' and 'pair-learning' on software engineering education. In *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*, pages 59–65.

[Zimmerman and Eber, 2001] Zimmerman, G. W. and Eber, D. E. (2001). When worlds collide! an interdisciplinary course in virtual-reality art. In McCauley, R. and Gersting, J., editors, *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, pages 75–79. ACM Press, New York.