Music in Introductory Object Oriented Programming

Viera K. Proulx, *vkp@ccs.neu.edu*
College of Computer and Information Science, Northeastern University, Boston, MA , USA

Abstract

Our Java-based *idraw* library has been designed to give a novice Java programmer the tools to design a simple interactive animated graphics-based game. The programmer focuses on the design of the game behaviour and representation of the game scene in terms of simple shape-based graphics. It has been used by hundreds of students over the past several years. This paper presents the Java *isdraw* library that extends our *idraw* library by giving the programmer tools to add musical effects to their game.

The new library provides the opportunity for students to practice working with sequences of data, designing loops, and designing classes that represent musical phrases, melodies, chords, and effects. It also illustrates the connection between the information and its representation as data. Of course, the addition of musical effects to an interactive game is a great motivator, making the learning more concrete and challenging.

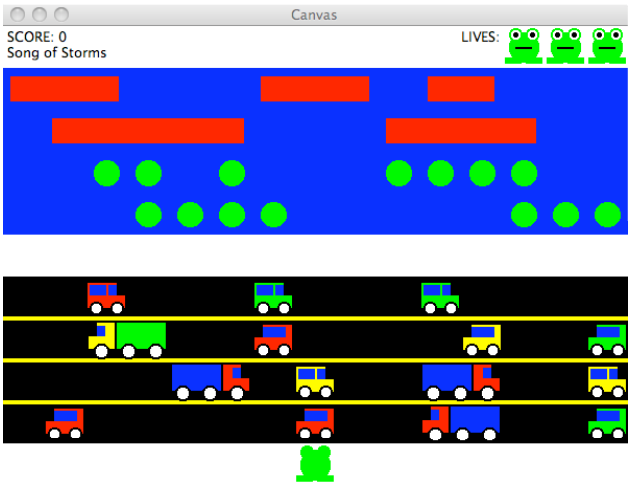

Figure 1. Frogger game designed with isdraw library.

Our pedagogy enforces systematic unit testing from the beginning. The design of the library that supports musical and sound effects makes it possible for students to play the tunes before embedding them in the game, and design tests for their sounds prior to playing them. Our goal is to combine constructive exploration with structured design discipline.

Keywords (style: Keywords)

Introductory computing; object-oriented programming; games and music; constructionism

Introduction

The *TeachScheme/ReachJava* curriculum focuses on introducing students to solid program design principles from the beginning. Several software artefacts that allow the student to focus on the key design concepts support the curriculum. At the beginning the teaching languages within the *DrScheme* programming IDE allowed student to begin programming without learning complicates language syntax, and with the ability to evaluate small program segments interactively. Over the years, the support for the first part of the curriculum, known as *TeachScheme!* has grown to include support for unit testing, support for the design of graphics-based interactive games, and, most recently, support for distributed program design with several clients communicating with a server. The most important feature of all these *teachpacks*/libraries is the ease of programming --- requiring only the basic programming skills.

Over the past eight years, we have been developing the next part of this curriculum, focusing on the program design in object-oriented style for class-based languages. Our curriculum starts with a simple Java-like language *NeuJava*, and progresses to the standard Java language as students see the need for language support for increasingly more sophisticated design of abstractions and libraries. Besides providing the language environment, over the years we have designed a *tester* library that supports unit testing within the constraints of the language knowledge of a novice student. This library is now used extensively in introductory courses, even when the instructors do not follow our curriculum. We have also designed and used extensively libraries that make it possible for students to design graphics-based interactive games, using only the very basic language skills --- working only in the mutation-free *NeuJava* language (the *draw* library). The more advanced version of these libraries uses imperative style and allows students to convert their games into Java Applets by adding with only a small wrapper class to initialize the applet (the *idraw* and the *adraw* libraries). But except for using the imperative programming style the library still asks the programmer to only provide the game model behaviour and the representation of the game's graphics as a simple shape-based graphics drawing.

Today's games nearly always include musical background and sound effects. We describe our new library that allows the beginner programmer add musical background and sound effects to their games. Additionally, the new sound library also provides a context for experimentation with musical phrases, and combines learning about music with learning to design programs that deal with sequences, and a variety of ways sequences can be manipulated and combined. The two versions, *isdraw* and *asdraw* are again targeted to Java Applications or Java Applets

The Tune Bucket and The World Library

The *idraw* library defines an abstract class `World` that builds a frame with a `Canvas` for the game display and declares several abstract methods that the student needs to implement. There is `onTick` method that represents the game action on each tick of the clock, the `onKeyEvent` method that represents the action in response to the different key presses, the `draw` method that defines the game display from the current state of the user's world, and a `bigBang` method that defines the initial world, the clock speed, and the size of the game `Canvas`. In a typical game, some objects move on each tick, some objects are controlled by the key presses, and the game ends when some object collide, or run out of lives.

The *isdraw* library adds support for sound effects and music exploration. An English saying describes musically inept people as those who cannot carry a tune. The whimsical response is that I can carry a tune - in a bucket. So, for the programmer with only a limited musical knowledge, we provide two buckets (instances of the class `TuneBucket`) for carrying the tunes: the `keyTunes` bucket for defining which notes are to be played in response to the key presses and the `tickTunes` bucket for defining the tunes to be played at each tick of the timer.

Originally, the library provided a list of constants that represents the pitches over three octaves. A more sophisticated musician would use the actual MIDI pitch codes directly. The two buckets get filled with an arbitrary collection of notes to be played during the event handling method. If the programmer wants to make sound when a key is pressed, he includes in the `onKeyEvent` method a call to `keyTunes.addNote` method that consumes as arguments the instrument to play and pitch of the note to play. Each key press plays the tune for one-quarter-note duration. In a similar manner, invoking the `tickTunes.addNote` method within the `onTick` method plays the given note when the `onTick` method is invoked. The notes added to the `tickTunes` bucket are played until the next tick event. We expected students to compose the tune as a list of notes and select the next note in the list on each tick.

Initially, this seemed to be a very primitive design, having no provisions for the duration of the tunes, little provision for repeating a theme, or for playing more sophisticated sounds. We let our students use the first prototype of the library in the fall 2009. A programming pair that included a musically talented student produced a *Frogger* game with an exciting jazzy background music that made the game quite impressive. Their code included several extensive lists of tunes; with one form each list to be played at each tick. They figured out how to represent the tempo, they defined a method that transposed the tune and/or scaled, it. Their work has been an inspiration to add tools and abstractions to our library so that both, musically talented, and musically challenged students would be able to add sound effects to their games.

During the Spring 2010, we were still using still the basic version of the library, but we gave the students several examples of what kind of sounds and music they can construct from the given building blocks. Many students explored the ways of generating polyphonic melodies, adding sound effects at the critical points during the game, annotating their code with auxiliary representation of the music data. A student with a serious interest in music constructed a sequencer application. The composer could add notes to a grid display where the height of a coloured marker represented the pitch, the horizontal axis represented the time, and the colour of the marker defined the instrument that should play the note. When the play mode was turned on, a thin vertical line moved across the grid and all notes that the line crossed were played.

This confirmed our prediction that a tool like this will inspire students to explore and construct interesting sound sequences and music --- all while learning the basic skills of writing programs that combine sequences of data into more complex structures.

Making Music: Evolution of the Library

Playing notes

The harpsichord music from the Baroque era had no provision for playing the note for extended duration. The playing of a sequence of quarter noted followed by a half note was accomplished by the timing of the initial key presses. Initially, we have adopted a similar technique for representing the note duration. We illustrate this on an example. The following sequence represents the first four lines of the Frére Jacques tune:

```
noteC,0,0,0,noteD,0,0,0,noteE,0,0,0,noteC,0,0,0,
noteC,0,0,0,noteD,0,0,0,noteE,0,0,0,noteC,0,0,0,
noteE,0,0,0,noteF,0,0,0,noteG,0,0,0,0,0,0,0,
noteE,0,0,0,noteF,0,0,0,noteG,0,0,0,0,0,0,0
```

Three silent ticks follow each quarter note; seven silent ticks follow a half note. etc. The constants `noteD`, `noteG`, are the names for the corresponding pitches. That means that the number of silent notes after the note is played represents the duration of the note. Surprisingly, this allows for constructing interesting and amusing musical sequences.

Representing notes

After our initial experiences with students it became clear that this tool could be extended to support extensive exploration of musical features and constructions without compromising our design-driven pedagogy of programming instruction. Additionally, we believe, the design of the library can serve as a model of different techniques of program design.

Our first challenge was to design a clean and robust representation of the MIDI notes. From students; point of view this is a wonderful example of multiple representations of information as data. One of the representations consists of the `pitch` and the `duration` of the note, another one specifies the note name (e.g. F), its `modifier` (sharp, flat, or natural), the `octave` on the piano keyboard, and the `duration` of play. Rather than defining methods that perform the conversion between the different representations, we defined a `Note` class with several constructors, each accepting a different representation of the note, but each of them initializing not just the `pitch` and `duration` field, but also the note name field, the `modifier`, the `duration`, and a field `snote` that records its representation as a `String`. So, a note representing the middle C, playing for 2 beats can be defined in either of the following ways:

```
Note c4n2V1 = new Note("C4n2");

Note c4n2V2 = new Note(60, 2);
```

The need for multiple representation of the same information and the use of constructors to make this possible is illustrated here in a compelling way.

Extending the Tune Buckets

Initially, the `tickTunes` and the `keyTunes` tick buckets limited the duration of each note to one tick. To make it possible for a note to be played over several time ticks, we needed to modify the design and the behaviour of these `TuneBucket`-s. Furthermore, in the initial version the programmer had to add the notes to the `TuneBucket` one at a time.

Music is a wonderfully complex time sequence. The MIDI synthesizer allows us to play up to 16 instruments at a time, and play on many of the instruments a polyphonic melody (a chord). Our goal in extending the library was to allow the musically gifted student to work with as many features of the MIDI interface as possible, yet keep intact the original simple setup for those that are musically challenged, or do not want to create elaborate musical structures.

Our new note representation includes the `duration`. We decided to limit the granularity to 1/16th note playing for one tick. So, a note of `duration` 2 is 1/8th note, note of `duration` 4 is a ¼ note. Of course, the actual time needed to play one note is set when we start the timer and specify the rate at which the time events should happen.

To capture the timing information and act on it, we added two new `TuneBucket`s to our `World`, and added method `nextBeat` to the `Note` class that simulates playing of the note for one beat by decreasing its `duration`. The `currentTickTunes` and `currentKeyTunes` `TuneBucket`s represent the list of notes currently playing. Instead of stopping the playing of all notes that started on the previous tick, only those whose `duration` has decreased to 0 are stopped. All notes added to the tune bucket on a tick or key event start playing, and are then moved to the current buckets. These are advanced to the next beat on each tick; the notes that fell silent are stopped and removed from the current buckets.

Designing Tunes

One can think of music as being a collection of scores, one for each instrument that needs to be played in a synchronized sequence. To model this, we designed a `Tune` class that represents one time event for an instrument. It includes a field that identifies the channel on which to play (or the instrument we wish to play) and a `Chord` --- a collection of `Note`-s to play. We then allow

the programmer to add notes to the `TuneBucket` in any of the following ways: a single note, a single note given only by its name as a `String`, a single pitch (which then plays for one beat), a `Chord`, a `Tune`, or an `Iterable` collection of `Tune`-s, or an entire `TuneBucket`. This provides the flexibility for how the programmer organizes the musical sequences.

The `TuneBucket` now contains a collection of 16 `Tune`-s --- one for each channel in the current MIDI program. New notes, chords, and tunes are added to the `Chord` associated with the `Tune` for the corresponding instrument.

Playing the notes and instruments

To allow playing the music apart from the interactive game and to promote further exploration of the musical structure, we added a `MusicBox` class. It initializes the MIDI synthesizer to a default program, or to the program given by the programmer, and provides the method to play or stop the given `Tune` or a collection of `Tune`-s. The game `World` class then uses an instance of the `MusicBox` to play and stop the `Tune`-s in its `TuneBucket`-s.

Combining notes and instruments

Rather than providing a specific structure for the melodies students compose, we suggest exercises that gradually lead the student to understanding how music is structured and how various components can be combined and used to generate the next collection of `Tune`-s to play. So, the student may start with a simple sequence of `Note`-s, compose a canon as a sequence of `Chord`-s, create an inversion, transposition, or glide reflection from the original sequence, and compose those into a new sequence of `Chord`-s. The `Chord` sequence can then be played on several different instruments.

We use this to motivate the design of iterators that deliver the next collection of `Tune`-s to play at the next tick, and use circular iterators to create a musical sequence in the style of piano roll, that repeats a melody sequence indefinitely.

Unit testing

The programs students design define the behaviour of the program in response to the tick and key events. Adding the desired music data to the appropriate TuneBucket before each event is invoked generates all musical effects. During the program design stage, students can design complete unit tests that verify that the expected music data has been generated for each tick or key event. To make this possible, wee add methods that allow the student to examine the current state, and the effects of advancing to the next beat for each of the classes we designed. The students can then test that the `nextBeat` method modifies the note sequence appropriately, and that when building a collection of notes to be played on the next tick (e.g. a `Chord`), the collection consists of the expected notes.

We believe strongly that developing a solid design discipline that included systematic unit test design and test evaluation is essential to making students confident and competent programmers, Additionally, our tests-first approach forces students to think through the problem carefully and understand the underlying issues before writing the code.

The Pedagogical Perspective

Our *draw* and *idraw* libraries have been designed to give the students an environment in which they can construct their own worlds and practice on an open-ended problem the basic program design skills. Besides providing a motivation for learning and exploration, it serves a pedagogical purpose as well. Students learn to design fairly complex collection of classes that interact in a number of ways: colliding objects, object aware of the locations of other objects, objects whose behaviour depends on other objects. By focusing on the game model we can insist on proper design, code that is well organized, documented, and tested.

Students built games such as *Frogger*, *ConnectFour* board game, the classic *Snake* game, *Tetris*, traffic simulation, space invaders, and a number of others. However, while the class interactions were quite complex, only a few of these required extensive manipulation of multiple collections of objects.

Of course, adding music and sound effects to the game makes the game more exciting and motivates students even more. But that is not the main reason we decided to extend our libraries. Working with the music sequences provides a rich environment for practicing programming with arrays, `ArrayLists`, and loops. Our exercises ask students to combine several melody sequences together, generate chords, cord sequences, transpose the music to a different key, or construct a canon. They build new classes to represent the complex musical sequences with the built-in iterator to generate the next set of instructions for the `tickTunes TuneBucket`.

Our experiences have been great. Students are eager to add musical effects to their games and get motivated to manage complex loops so their music sounds just right. Furthermore, after working with similar games in their first semester, students start loosing motivation when the new game designed in the object oriented style differs little from the one they have designed in the functional style. Adding the sound components makes the whole game design more interesting.

Acknowledgements and Further Plans

This work has been inspired by Erich Neuwirth, especially his pedagogical use of music with spreadsheets, by Uri Wilensky and his team's use of music within NetLogo, and Jenny Sendova's exploration of the music phrase composition as examples for working with sequences of data. Erich Neuwirth's work on spreadsheets and music was especially influential. His spreadsheet language is used to create musical effects by manipulating music representation within the spreadsheet. Our project emulates some of this work in the context of a standard introductory object-oriented programming environment. The author wishes to thank Erich Neuwirth for his continued support and his encouragement for experimenting with music-supported pedagogy.

We plan to extend this work to provide tools for the display of the music in a variety of ways and to leverage the key event handling to allow students to enter the musical sequences by playing the computer keyboard. We will add the library to our website at *http://www.ccs.neu.edu/javalib/* and include tutorials, sample code, as well as downloads for both the library files and the source code.

References

Felleisen, M., Findler, R. B., Flatt M., and Krishnamurthi S. (2001) *How to Design Programs*. MIT Press.

Holbert, N., Penney, L., & Wilensky, U. (2010). *Bringing Constructionism to Action Gameplay*. Paper to be presented at Constructionism 2010, Paris

Neuwirth, E. (2010) *Music and Spreadsheets*. http://sunsite.univie.ac.at/musicfun/.

Proulx V.K. (2009a) *The pedagogy of program design*. In Proceedings of DIDINFO 2009. Brusno, Slovakia, pp. 65 − 74.

Proulx V.K. (2009b) *Test-driven design for introductory OO programming*. SIGCSE Bulletin 2009. 4191), pp. 138-142.

Sendova, E. (2001) *Modelling Creative Processes in Abstract Art and Music*, Eurologo 2001, Proceedings of the 8th European Logo Conference 21-25 August, Linz, Austria.