Mobile Music Development Tools for Creative Coders

Daniel Stuart Holmes
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_dissertations

Part of the Audio Arts and Acoustics Commons, Composition Commons, Graphics and Human Computer Interfaces Commons, Interdisciplinary Arts and Media Commons, Music Education Commons, Music Performance Commons, Software Engineering Commons, and the Systems Architecture Commons

Recommended Citation
https://repository.lsu.edu/gradschool_dissertations/4935

This Dissertation is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.
MOBILE MUSIC DEVELOPMENT TOOLS FOR CREATIVE CODERS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Music

by

Daniel Stuart Holmes
B.M., Southeastern Louisiana University, 2007
M.M., University of Alabama, 2012
August 2019
Acknowledgments

First, I’d like to thank Dr. Stephen Peles. He had a reputation for having unusually difficult classes, but he was also an excellent pedagogue (and had a soft side once off the topic of music). He accepted me as a masters student and TA in music theory. His tonal structural analysis classes were the most challenging I’ve taken throughout all of my degrees, but they were also extremely rewarding. He also helped me gain a foundation for hearing musical structures that I might never have achieved otherwise. I have an auditory processing disorder that makes it more difficult than normal to hear differences in individual events, which made undergraduate ear training something I barely passed (and made acceptance into my masters program probationary). After finding out, Dr. Peles stopped giving graded atonal dictation quizzes and began focusing on larger constructs. I then found that while I struggled with short exercises and dictation, I could learn to hear more complex musical relationships and structures. Ultimately, I switched from music theory to electronic music composition without consulting him. He passed away before I ever apologized, which I will forever regret. It was hard being his assistant but extremely rewarding to be his student, and I’m thankful for having had the opportunity to study under him.

I’d like to thank Patrick Kerber who was my first music teacher. I had just started high school, and my parents bought me a guitar and lessons for Christmas. It turns out Mr. Kerber was the guitar instructor at the local university but taught on the side at this music shop my parents had visited. I was learning power chords and grunge rock, and then I walked in to a lesson a little early. Mr. Kerber was playing classical guitar, and I said I wanted to learn to do that. I studied with him for eight years—all through high school and then as an undergraduate. Even when I lost interest in pursuing guitar as a career, he proof-read my (really bad) first theory paper and wrote a recommendation to Alabama.

I’m really not sure how to even begin to thank Dr. Stephen Beck and Dr. Jesse Allison. They gave me a chance even though I applied to the EMDM program late. I originally came
to LSU without funding, but they had secured an assistantship for the following semester. They consistently brought opportunities to my attention, including my positions working for Louisiana Sea Grant doing mobile web development and as a full-time adjunct teaching music theory and music technology at Southern University.

I might have never pursued mobile development if Dr. Allison hadn’t asked me to join their iOS bootcamp before I had even started at LSU (don’t worry, it’s all turned out to be a good fit), and Dr. Beck regularly showed and voiced support for my teaching and creative ideas. They’ve been beyond patient even when I haven’t expressed my struggles in detail, and the confidence they’ve shown in my ability, even when my own confidence has been lacking, has meant more to me than any particular lesson or class. Jesse Allison has been thoughtful and supportive beyond my experience with most teachers, and Steve Beck always has just the right thing to say when catching up in the hall, especially during those nervous moments before a show. More than anything, I appreciate Dr. Allison giving me the space to work on things at my own pace and in my own way. I have occasional manic periods interwoven with really long downs, and there’s no way I would have survived grad school if I had been forced to always keep very strict time tables.

I’d also like to thank my entire committee: Drs. Jesse Allison, Stephen Beck, Edgar Berdahl, Brett Boutwell, and Kevin McCarter. There was a three and a half year period between my general defense and getting in touch about finishing my dissertation. I never really planned to not finish, but things just kept getting pushed back. I’m extremely thankful for everyone’s willingness to continue serving and saving me from hassles I may not have had the will to overcome.

I’d like to thank Dr. Linda Cummins for being amazingly thoughtful and supportive in my early grad school years. She influenced how I think about non-tonality, and she was always available to give advice on any matter—musical, personal, or otherwise. It was late in the season when I had decided to apply to PhD programs, and, without prompt, she found out about EMDM at LSU, pointed me towards Steve Beck, and helped set up
an interview. She genuinely cared about her students and colleagues and was an under-appreciated, behind-the-scenes force of nature.

Finally, I’d like to dedicate this work to Dr. Marvin Johnson.

I originally moved to Tuscaloosa, AL in order to study music theory, but Dr. Johnson convinced me to take his course on electronic music as an elective. There were only two of us in his class that semester. Before that, I hadn’t had any real exposure to computer music or coding, but it didn’t take long to realize electronic music was an area in which I was very comfortable. I was on a half-time assistantship, but the following semester Dr. Johnson had also acquired a small fellowship for me to serve as his TA in the electronic studios–funding for which he continued to fight and secure for the duration of my time at UA.

As a mentor, he was always patient and interested. He listened to and considered new ideas no matter how ridiculous they might have turned out, and he spent time with my music. I might have ended up on a completely different path if he had not made this known. One day, he casually mentioned that the more he listened to a recent composition of mine (which I had unconvincingly explained in a previous lesson) the more he liked and understood what I had done. We had a discussion where he articulated things I had tried to explain but failed to do. The fact that he took time to explore my music and consider my ideas has never left me. He helped shape the way I think about music and the way I interact with students.

I’m sad to say Marvin Johnson passed away before I finished my education. The last time we communicated was a couple years after I had left Alabama, but, after he passed, his wife shared with me that even though he continued teaching after I graduated, I had been his last true student. I think he would have been proud to learn of my recent work and accomplishments, and I wish I had had the chance to show and thank him in person one last time.

Dr. Johnson, I will forever be grateful for your guidance, support, and friendship.
# Table of Contents

ACKNOWLEDGMENTS ................................................................. iv

LIST OF FIGURES ................................................................. vi

ABSTRACT ................................................................................. ix

CHAPTER

1 INTRODUCTION ......................................................................... 1
  1.1 Accessing Technical Fields of Study ........................................... 2
  1.2 Targeting Creative Coders ......................................................... 4

2 KDKIT ..................................................................................... 7
  2.1 Self-Perceived Understanding of Context .................................... 7
  2.2 Native Language Library .......................................................... 9
  2.3 User-Centered Design for KDKit ............................................... 10
  2.4 Hierarchical Class Structure ..................................................... 18
  2.5 Dot Syntax Scripting ............................................................... 20
  2.6 Technical Descriptions for Select Classes ................................. 22

3 EXAMPLE MOBILE APPLICATIONS DEVELOPED WITH KDKIT ................. 43
  3.1 In Sen ................................................................................. 44
  3.2 Lights ............................................................................... 47
  3.3 Grain ............................................................................... 49
  3.4 Minim .............................................................................. 52

4 CSPD NOTATION ........................................................................ 57
  4.1 The Score as a Musical Object ................................................. 58
  4.2 Free Composition .................................................................. 60
  4.3 CSPD Notation Specifications ............................................... 62

5 COMPOSITION .......................................................................... 69
  5.1 Robot develops anxiety, dissociates (27') ................................... 69

6 CONCLUSION ............................................................................ 73

REFERENCES ............................................................................. 74

APPENDIX, ROBOT DEVELOPS ANXIETY, DISSOCIATES (SCORE) ............. 79

VITA ............................................................................................. 84
List of Figures

2.1 Example starter program in Swift. Since Swift is a type-safe language, the double and integer can not be operated on together. .................. 8

2.2 Complete code for a keyboard synth app with KDKit. ......................... 9

2.3 Cocoa MVC as per Apple documentation. ........................................... 13

2.4 Functional SAM according to KDKit. ............................................... 15

2.5 Text on the thumb of KDSlider moves when touched. This is a mobile-first microinteration to ensure a user can see the text that would normally be under their finger. ................................. 17

2.6 KDKit makes it simple to create a scrolling view out of multiple conjoined views. KDScrollView.append() calculates the geometry to place consecutive, connected views in the scrolling direction. ......................................................... 18

2.7 Comparison showing how KDKit’s dot syntax scripting reduces code verbosity, improves visual organization, and makes text searches more efficient. ................................................... 21

2.8 Adding a gesture recognizer with delegate method in Swift. The delegate method must be its own separate function, and the UIGestureRecognizer instance must reference it as a selector. The selector name can not be filled in with code completion. .................. 23

2.9 Adding a gesture to a KDView with callback function. The class handles adding its own gestures with .add functions, and the action is defined in the .onTouch callback. ..................................... 23

2.10 Test for multiple gestures using basic Swift solutions.......................... 24

2.11 KDColor provides a robust selection of modern design colors. .............. 25

2.12 KDKit UI classes conform to the KDColorTheme protocol. .................... 25

2.13 KDColorTheme was implemented to aid in app-wide color updates. This example shows how views in the view stack can be updated with a forEach loop. ............................................. 26

2.14 KDRoll is a random number generator as a dice metaphor. .................. 27
2.15 Instead of Swift’s standard random number functions which return a single number, KDRoll uses the GameKit class GKRando
domDistribution which manages random distribution tables. ............... 27

2.16 Multiple dice with independent tables can be rolled at once. ............... 28

2.17 KDMotion makes it simple to use motion data. ......................... 28

2.18 KDLable functions to aid with text label geometry. ...................... 29

2.19 Text styles added to KDLable. ............................................. 30

2.20 Basic example using KDKeyboard. ........................................ 30

2.21 Example customizing KDKeyboard. ....................................... 31

2.22 UI animation example using UIView.animate(). ....................... 33

2.23 UI animation example using KDAnimate. ............................... 34

2.24 Basic example using KDEasyPD to manage libpd. ....................... 35

2.25 Basic example using KDKit’s AudioKit classes. ......................... 37

2.26 KDFXChain creates a signal chain from a KDInstrument through any number of KDFXUnit. ................................................. 38

2.27 KDFMOscillatorBank is nearly as simple to use as KDOscillator. .... 39

2.28 KDScore can be used to create a timed, ordered script of events or repeat a command at regular intervals. ......................... 40

2.29 Example using KDPlsit to store and load data in persistent storage. .... 41

3.1 The In Sen UI includes several sliders that affect sound in real time. ..... 44

3.2 Lights features a rippling water effect when touched. ..................... 47

3.3 Grain is designed so it can be played on multiple devices at once........ 50

3.4 Minim’s pads can be individually customized. ............................ 52

4.1 Example of CSPD notation dots on a staff. ............................... 64

4.2 CSPD dots should be arranged to show relationships. .................... 65
4.3 When necessary, specific rhythms can be notated in the legend or next to the CSPD staff. In this example, the notated rhythm should be featured prominently throughout the section.

4.4 This is an excerpt from *Cello Pie*, a piece for three performers on Grain. While left to right represents the passage of time, duration is non-specific in CSPD. Vertical lines can be used to denote landmarks or points of arrival. In some cases, it may be appropriate to mark a target time above a vertical line.
Abstract

This project is a body of work that facilitates the creation of musical mobile artworks. The project includes a code toolkit that enhances and simplifies the development of mobile music iOS applications, a flexible notation system designed for mobile musical interactions, and example apps and scored compositions to demonstrate the toolkit and notation system.

The code library is designed to simplify the technical aspect of user-centered design and development with a more direct connection between concept and deliverable. This simplification addresses learning problems (such as motivation, self-efficacy, and self-perceived understanding) by bridging the gap between idea and functional prototype and improving the ability to contextualize the development process for musicians and other creatives. The toolkit helps to circumvent the need to learn complex iOS development patterns and affords more readable code.

CSPD (color, shape, pattern, density) notation is a pseudo-tablature that describes performance interactions. The system leverages visual density and patterns of both color and shape to describe types of gestures (physical or musical) and their relationships rather than focusing on strict rhythmic or pitch/frequency content. The primary design goal is to visualize macro musical concepts that create middleground structure.
Chapter 1
Introduction

Throughout recorded history, new technology has influenced the musical arts. Something new is discovered, and musicians explore how it can be used to improve on their existing practices or invent new ones. Steel was used to create piano strings, valves enabled a full range of chromatic horn notes, and computers now blink and beep on a concert stage. Since these technologies were not invented specifically for music, musicians either had to create the change themselves or have the foresight to ask for and direct it. People with some stake in the musical arts had to invest time and resources—beyond what they invest in their primary musical work—towards making those things happen at all.

Over time, this general process has not changed, but modern technology has had a significant impact on how it happens. The prevalence of digital computing devices has empowered musicians to create in new ways [1], not just because computers can be used as musical instruments, but also due to how and what we can learn. New information is readily available at near instant speeds, and electronics have become a frontier for introducing new concepts into the musical repertoire. This isn’t to say electronics will definitely be the source of the next great musical movement, but young creators are exploring computer technology more than new ways to build acoustic instruments. In fact, some study in creating electronic music is often a requirement for student composers. Unlike their studies in acoustic composition, this requires them to invest in a technical component that may not have other practical musical applications outside of the creation of electronic music. This is similar to their musical predecessors exploring the technology of their time but on a different scale. For these students, having knowledge in both art and technology is a requirement rather than a special case.

Some embrace this study and some do not. Either way, there is both a practical and personal limit—what we expect from students and of what they are capable—to how much they can invest, but those limits are malleable [2]. As the what and how we learn changes,
so do our practical and personal limits. What we can learn is changing relatively fast, but it is available to nearly everyone at the same rate. How we learn is different for different people, and that is an area that can be influenced with more immediate personal effect [3].

With digital musical technology, the leap from idea to manifestation is relatively small. There’s no need to source raw materials or manufacture anything. Digital musical ideas can be realized nearly as fast as the creator can learn something new. Unless specialized equipment is required, there is no physical barrier keeping us from acting on new knowledge or skills. Today, time is a resource of increasing value, and our ability to and rate at which we learn are real commodities. As such, we can create value by finding ways to improve our learning capacity.

This improvement is often left to the student and often attributed to dedication and willpower [4], but perhaps the standard way these technical skills are taught isn’t the best way to teach everyone [5], [6], [7].

1.1 Accessing Technical Fields of Study

Current educational strategies in technical fields favor a certain type of student, and other students, even if they have perceived potential, are not able to perform as effectively in that environment [8]. Studies in computer science and engineering education have identified problems such as self-perception of understanding and poor standardized teaching strategies [5], [9], [10], [11]. Each came to similar conclusions: formal education in technical fields will benefit from tools and strategies that address and respect the experiences and perspectives of a diverse body of students, or contextualization. These studies suggest that many students are either failing or not reaching their potential because of the way they are taught and the way the field is represented, and this trend is actively working against technical fields by creating educational barriers that discourage diverse groups from pursuing study.

Bush and Mott make the case that technology has been pursued incorrectly as an educational tool. They suggest the focus of educational tools should be from helping teachers teach to helping learners learn, or learner-centricity. They also explain tools benefit from being “open, modular, and interoperable.”
Du and Kolmos, who studied gender diversity in engineering, suggest formal education that replaces technical content with interdisciplinary knowledge, focuses on practical skills as well as problem-based learning (PBL), and employs non-technical assessments [8]. Moesby discusses the importance of *perception of context* and concludes that PBL targeting personal skills is effective in engaging more students [4]. Sheppard et al. emphasize not favoring theory over practice [9]. Lahtinen, Ala-Mutka, and Järvinen find that actually creating a functional program, rather than studying abstract examples to demonstrate a theoretical concept, “is a major positive force on students’ traditional programming work [6].” All of these solutions are related in that they are student-oriented, but the changes they are championing do not only apply to currently enrolled students. They suggest normalizing contextual education such that it is more personally meaningful to a particular student will not only improve the quality of graduating students but will also attract more diverse groups to study technical fields.

This is a complex, long-term goal for formal education, but a 2017 study of developers showed 90% of respondents identify as being at least partially self-taught, and 54% felt that their formal education in computers or engineering was not entirely important to their current careers as software developers [12]. This may be related to the previously mentioned shortcomings in formal education, but perhaps developers are taking the problem into their own hands. Even so, the tools developers of all experiences are using to teach themselves are still most often technically oriented [12]. While this sort of informal learning is attractive to knowledgeable developers learning a new skill, it does little to offer access to new or non-developers who are interested in programming but are not personally oriented to learn from technical documentation.

The studies referenced above focus on gender, racial, and socioeconomic diversity, but, given the similar findings in all regards, perhaps these strategies could be effectively applied to any group attempting to access technical study in a non-traditional way. The ways we educate demographically typical engineering students may not be the most effective or
attractive methods to teach musicians interested in or compelled to learn programming and other technical skills in support of their art.

The popularity of graphical programming environments with artists and designers already supports this difference [13]. Desktop applications such as Max and Pure Data are favored in media arts circles, but many computer science students have never heard of these programming tools.\(^2\) Similarly, the rise of drag-and-drop front-end web design programs such as Froont, Webflow, Squarespace, and some Wordpress plugins shows how these visual tools help designers enter the role of programmer. These solutions hide much of the abstract technical requirements that accompany most textual programming languages and make it much easier for artists to quickly learn to make products that are relatable and personally meaningful. A beginner gets to skip discussions of bits and compilers and jump right into making interfaces, sounds, and visuals. These creative coders have a different stake in the material, different goals, often a different educational background, and the tools they are gravitating to are different from those of their math, engineering, and computer science counterparts.

### 1.2 Targeting Creative Coders

It’s unclear where the term originated, but John Maeda, once the director of the MIT Media Lab Aesthetics + Computation Group (ACG), settled on *Creative Code* to title the anthology documenting his and ACG’s work [15]. Maeda impresses on designers to view the computer as an artistic entity, a medium worth attention. Rather than simply relying on graphic editors, audio workstations, and text processors, Maeda makes the case for artistic design that further leverages and even integrates computers. As such, *creative code* refers to programs that serve an artwork or aesthetic in a more significant role than simply being the means for creating files and formats, and *creative coders* are those who envision and

\(^2\)Max has become the go-to commercial solution for media arts and physical computing, but the concept of “the patcher” has been cloned many times over and remains popular with digital artists across media. Miller Puckette described the patcher at the International Computer Music Conference in 1988 [14], and, since then, Max, Pure Data, Touch Designer, and Open Frameworks have become staples of digital media programming. Even so, these graphical environments have not found a standard place in computer science curricula.
create that work. As an umbrella term, creative coders include artists and designers of any media, background, or experience as long as they create computer programs as a part of their art.

While it’s possible for trained programmers to be creative coders, those who primarily view themselves as artists drawn towards incorporating computer technology in their work are the focus of this project. For these people, the potential solutions for addressing diversity in engineering education seem significantly appropriate.

While some people thrive when reading technical documentation, studies have shown that many students will fail to progress when this is their primary resource [5], [7]. The popularity of the aforementioned graphic programming environments shows us that media artists tend towards technical solutions that have a more direct and obvious connection to meaningful goals, which is in line with observations concerning the attraction and retention of diverse groups of technical students. Ultimately, though, this tendency to only use these graphical environments does not encourage exploration beyond the platforms and capabilities specific to those tools. The connection between patching languages and textual languages is not always obvious, and the self-efficacy of these artists often lacks when confronted with what seems to be more technical abstraction [2], [3].

In response, this project provides a support system designed to emulate a learning experience similar to graphical programming environments—one that is better contextualized by having a more direct connection to a mature product and favors practical skills over theory—that remains relevant and useful at all learning levels by affording production-quality development. It is meant to serve as an example of an educational tool that embodies the previously discussed solutions for attracting and supporting non-traditional students in engineering education, but specifically targeting musicians, composers, and other sound artists as a demographic. The project focuses on the relatively immature yet globally consuming world of mobile computing [16], and it includes a code library for creating musical iOS applications, a graphic notation system designed for gestural performance with mobile
devices, and example applications and scored composition demonstrating these tools. It is intended to facilitate the creation of new art by encouraging musical creatives and other media artists to explore technology beyond their typical by improving their self-efficacy.
Chapter 2
KDKit

KDKit is a library of high-level classes for the development of mobile audio applications. It’s designed to be accessible to non-coders yet still useful to experienced programmers. KDKit is written in Swift for iOS and organized into three primary categories: user-interface (UI), audio synthesis, and utility (e.g. data management, networking).

KDKit’s governing design philosophy is to simplify the technical aspect of user-centered design and development with a clear connection between concept and deliverable [17], [18], [19]. This simplification addresses learning problems such as motivation, self-efficacy, and self-perceived understanding of context by lessening the gap between idea and functional prototype and improving the ability to contextualize the development process (see chapter 1.1).

2.1 Self-Perceived Understanding of Context

For any developer, but especially casual or secondary programmers, it can be difficult to self-motivate when it’s not clear how to reach an end goal that is personally significant. Recreating something that is clearly instructed can be interesting and gratifying at first, but the drive to continue learning is often diminished if there isn’t an obvious personal use for the lesson [8].

Early programming lessons often include simple data types and basic math. Starter programs of these sorts are such small building blocks, and, for the uninitiated, it’s not always clear how these exercises lead to feature-rich applications. This is a basic example of self-perceived understanding of context [10]. A thorough grasp of data types will eventually be necessary, but initially there’s no context for their significance beyond printing some words on a screen or creating a command line calculator. Even though an experienced developer knows it to be an important lesson, the perceived simplicity and lack of future context can detract from the value and impede a student’s motivation and progress,

¹The URL https://notnatural.co/kdkit/ is the persistent homepage for KDKit.
especially if the student has not already made a long term investment in the study.

```swift
func helloworld() {
    let helloString = "Hello World!"
    print(hello)
}

func datatypeError() {
    let five:Int = 5
    let seven:Double = 7

    let sum = five + seven // Binary operator '+' cannot be applied to operands of type 'Int' and 'Double'
    print(sum)
}

func datatypeWorking() {
    let five:Int = 5
    let seven:Double = 7
    let sum = Double(five) + seven
    print(sum)
}
```

Figure 2.1: Example starter program in Swift. Since Swift is a type-safe language, the double and integer cannot be operated on together.

### 2.1.1 Granularity of the Learning Object

KDKit addresses this problem with a top down approach. Rather than beginning with the simplest building blocks and working up towards more complex entities, KDKit decreases the granularity of the learning object by allowing complex but self-contained components to be created without prior programming knowledge [20].

Figure 2.2 is an example of how a first time coder can create a working keyboard synth on their mobile device.

This approach is similar to how graphical patching languages are structured, and it is this hierarchical granular organization of the object-oriented programming paradigm—not just the graphical nature of those environments—that is an effective and attractive learning tool [13], [20], [21]. The decreased granularity of the learning object aligns with necessary skills and information that more closely connect to a product the learner already values,

---

2The term *granularity* is borrowed from Bukvic et al. to describe how fine or coarse the learning object. A coarse object has less granularity than a fine object. This is a sliding infinite scale for the purposes of grouping and comparing objects, but there is no precise measurement. For our purposes, the granularity of a learning object references how high or low level the programming entity. In developer jargon, a low level entity is less complex which coincides with a finer granularity.
and their self-perception of context is improved. As a learning tool, the example still affords introductory discussion on several accessible technical topics (e.g. classes, data types, view hierarchy and geometry, functions), but the simplicity and readability of the code and the perceived value of the product leads to increased self-efficacy. What’s being challenged is the idea that all programming education must begin with a technical understanding of algorithms and control structures. Instead, KDKit creates a more accessible environment for learners with non-technical backgrounds by offsetting any need to first discuss abstract concepts far removed from relatable products.

2.2 Native Language Library

Many developmental tools have strict access-control over the internal code. This is often necessary to ensure the application works as intended, but it detracts from the ability to extend a given object and, in the case of educational tools, limits the learning potential. For example, in Max it is possible to inject independent textual code into a program but affecting a particular UI object beyond the public parameters requires technical experience.

Figure 2.2: Complete code for a keyboard synth app with KDKit.

```swift
func keyboardExampleApp() {
    // Create a keyboard and add it to the view hierarchy
    let keyboard = KDKeyboard(self.view.bounds)
    keyboard.addSubview(self.view)

    // Create some sounds to play
    let sines = KDOscillatorBank()
    let orchestra = KDOrchestra(withInstrument: sines)
    orchestra.onStage()

    // Play a note while the key is held
    keyboard.onKeyDown = { key in
        sines.play(key.value)
    }

    // Stop a note when the key is released
    keyboard.onKeyUp = { key in
        sines.stop(key.value)
    }
}
```

3Data-safe classes with controlled public access are a major aspect of the object-oriented programming paradigm in general. In this case, the comparison is between how the environment presented by many other learning tools is not accessibly extensible while KDKit is designed to be hacked as a part of the learning experience.
outside the realm of the patching environment. This is acceptable for a program like Max, but KDKit is designed to scale from beginner’s tool to commercial development library.

In this regards, KDKit is built as a native solution and can take full advantage of the capabilities of Swift and Xcode. While its classes present a simplified, more readable syntax, every object can be extended using native class extensions and, as the user gains more familiarity with Swift, the code comprising each class becomes accessible for further customization or templating. KDKit is meant to be both used and explored. A novice can relate to and use its coarsely tuned objects, with more proficiency a user can look at how objects are built to find example code or create extensions, and an adept can use the classes as templates for building their own solutions, all with compounding knowledge of the same programming language and development environment.

Another benefit of KDKit being written in Swift is due to the language capabilities of the Xcode integrated development environment (IDE). Xcode can compile Swift, Objective-C, C++, and C, and all of these languages can exist in a single project. KDKit does not inherently limit or impede access to the lowest levels of iOS programming. In fact, KDKit can serve directly as a step towards the full range of iOS development.

2.3 User-Centered Design for KDKit

User-centered design is a process where the tasks and workflow of a product and its components are evaluated, throughout all stages of design and development, with concern for usability goals [17], [18], [22]. There are two levels to the user-centered design paradigm concerning KDKit: it is designed for a specific user base, but it is also designed to aid its users in their own user-centered design.

---

4The term native refers to a language that compiles directly to an executable that can run on the operating system in question. Swift is native to iOS. An application written in a non-native language requires an interpreter level or emulated environment. This can be acceptable and useful in many cases, but it also comes with limitations and quirks that may sometimes be severe.

5Xcode also includes support for Objective-C++, Java, Python, Ruby, Applescript, and ResEdit. Third-party modules add support for Pascal, Ada, C#, Perl, and D, and JavaScript functions can be called using WebKit.

6KDKit has not yet undergone rigorous usability tests as would be expected with commercial UCD processes. To clarify, the current version is the product of multiple iterative development cycles informed by the experiences of a few knowledgeable developers. It also leans heavily on existing research and practices
KDKit’s objects and their use syntax were created to reduce or remove unnecessary non-contextual language or jargon. In order to instill a balance between usability and customizability, high-level objects can be functionally instantiated with sensible defaults in minimal code, but they also can be created by passing in custom parameters. Unlike many core Foundation and UIKit classes, KDKit initializers also prompt for parameters that may provide useful or even necessary functionality. KDKit also reduces or removes common redundancies in order to improve code readability and make iterative development more time-efficient.

Another of the design goals behind KDKit is a dedication to making it look easy while still leaving room to learn. Project-based learning (PBL) and undercover design were highly relevant in this regards [23]. A learner isn’t meant to look at how the toolkit works and think, “I can do that.” They should look at the toolkit and think, “This is what I am going to make.” It was more appropriate to design KDKit to play to the strengths and expectations of its target audience rather than focus on underlying code finesse and efficiency that might appeal to technical-leaning students [4], [5]. Even so, the tools are meant to have an extendable life-cycle, and they should also afford robust, scalable products. Unlike many other entry-level learning tools, it was important that KDKit not be technically limited and that its potential for future use be apparent.

Finally, KDKit aids its users with their own user-centered design. Learning to address usability is a separate topic altogether, but the design process itself is afflicted with problems similar to the technical educational hurdles previously discussed. It’s one thing to re-design an aspect of your app, and it’s another thing entirely to have to code each new solution. KDKit supports rapid iterative design and development in order to alleviate negative effects on self-motivation and self-efficacy.

For KDKit, sensible defaults are values that can be seen or heard without surprising or jarring effect. Foundation and UIKit are the collections of Apple-provided functions and classes that make up the bulk of iOS user-interface development.
2.3.1 Rapid Iterative Development

Since KDKit is primarily for interactive musical applications, user-centered design looks a little different from that of a reference, utility, or productivity app. In many ways, it’s more like designing a game. We’re not always talking about workflow efficiency, but we focus on physical interaction, engagement, and information visualization. One of the biggest challenges is finding a balance between ergonomics, audibility, and accuracy of control [24]. KDKit aids in this by supporting and encouraging rapid iterative development (RID).

RID is the idea that a product or component can generally be improved after a comprehensive evaluation, and quickly producing iterations with approximate adjustments can coincide with a better overall design [17], [25]. The problem is that it can be a real bummer to tell any coder, especially a new one, that they should start over and do better. The simplified syntax, encouraged code organization, and improved accessibility to advanced features all help address the issue, but KDKit’s primary benefit in this regards remains the granularity of contextually useful objects.

The earliest iterations of KDKit began with a collection of objects independently created with little or no unifying structure. As KDKit evolved to support learning as well as making, it became necessary to develop better support structures in order to ensure it continued to afford RID. Simply put, the original objects were easy to use individually, but they didn’t always mesh well when put together. In response, recent iterations have adopted a structural paradigm and informal protocols to ensure individual components retain integrity as granularity decreases.

2.3.2 Programming Paradigm: Avoiding the Delegate Pattern

KDKit avoids Apple’s suggestion to use the delegate pattern in iOS development. Their suggestion is meant to support the Model-View-Controller (MVC) programming paradigm by providing a remote connection between the interface and the data it controls.\(^9\) This

\(^9\)Apple documentation explicitly endorses their version of the MVC paradigm (Cocoa MVC) and the delegate pattern. This has lead most tutorials and other learning material to embrace only these concepts in their instruction.
delegate can be beneficial in the case where there are many instances of a class that should be connected to independent data sources, but it can be a hindrance or annoyance when only a few or even one instance is required.

Figure 2.3: Cocoa MVC as per Apple documentation.

The abstract nature of the delegate pattern can be especially confusing to new programmers with no exposure to theoretical development paradigms. As such, instances of KDKit classes own their own data and references, perform potentially unsafe mutations internally, and communicate with other classes using callback functions.\textsuperscript{10} This is more in line with object-oriented programming afforded by graphical programming environments. Here, it is adapted for a native, mobile textual language, and the patch cable metaphor is replaced with a protocol-driven application programming interface (API) designed for inter-object functionality with minimal abstraction.

A core concept of the MVC and delegate pattern is to decouple data models from the objects they affect and the things that affect them. Since KDKit classes own their own data, there is no need for a remote interface connecting views to models (the delegate).

\textsuperscript{10}A callback function is a responding function call after the initial request is completed. In KDKit, callback functions are properties owned by the target object. The callback tasks are passed in the requesting parameters as a completion block.
KDKit classes do decouple entities internally, but it more closely aligns with a hybrid of State-Action-Model (SAM) and functional programming concepts.

Under the SAM paradigm, a model should receive a request for an action, accept or reject that action, and then update the state if accepted [26]. This is a different approach from MVC in that the model has a more active role and the view is reactive.\textsuperscript{11} For the KDKit protocols, this paradigm is useful because some classes own views, but some are themselves a view window. Rather than adopting a different task hierarchy for these different types of classes, giving the model the active role and having the view remain reactive means state-affecting and data-mutating functions can remain consistently similar throughout the library. For example, those functions will reference input parameters or instance properties instead of \texttt{self}, and the view state does not require direct attention from the user.

Functional programming is essentially the antithesis of object-oriented programming in that it suggests code patterns avoid state-affecting mutable data altogether [27].\textsuperscript{12} It is attractive because it helps ensure code reliability and reduces the potential for invalid data mutations, but, concerning the development of applications with visual interfaces, it is an extremely abstract paradigm to present to new coders, especially considering the iOS API is already object-oriented.

Instead of adhering to any single paradigm, KDKit \textit{encapsulates} the SAM paradigm and \textit{limits} state-affecting mutable data by encouraging only immutable data be passed between objects.\textsuperscript{13} Each object owns its own data, state, and action functions but encapsulated in a wrapping class. Valid data mutations can be enacted using class functions with type-safe parameters, but any external mutations must occur on a differently referenced data model.

\textsuperscript{11}The term \textit{view} is used to refer to both the view as a component of MVC and the view as a visible graphical interface. While it is possible for both cases to refer to the same entity, they can also be mutually exclusive terms.

\textsuperscript{12}Functional programming is not a new concept, but it has recently been championed as an important but oft overlooked alternative to object-oriented programming.

\textsuperscript{13}SAM guides the internal design of KDKit classes but does not govern how objects interact with external entities. Instead, external interactions are influenced by functional programming concepts which replaces the patch chord metaphor.

---

14
and then explicitly set as the new data for the object. This means there is no need for the model to accept or reject a mutating request since the request is always known to be valid (its origin is always itself), but external changes and data sharing can still occur as long as its on a copy that is passed back as an immutable version.

![Functional SAM according to KDKit](image)

**Figure 2.4:** Functional SAM according to KDKit.

This pattern was adopted for the benefits it presents a new coder in addition to improving the integrity and interoperability of high-level classes. Internally, models are still decoupled and tightly controlled in order to ensure the data is always valid and, since required mutating functions are owned by the objects themselves, matches the view state requirements. Externally, the view controller and other KDKit objects have no inherent knowledge of or control over each others’ state and must request access or action. KDKit objects can be “patched” together in a more conceptually accessible manner because they expose usable data, self-validate incoming data, and provide an interface for direct connections via callback rather than a remote delegate. This reduced level of abstraction leaves less room for user error, and the protocols and internal structures help ensure the
compatibility of new objects.

2.3.3 Mobile-First Design

On the surface, this may seem like an obvious solution for native mobile applications. Mobile-first design is a term usually reserved for web development, and it is the idea that a web page or app should be visually and interactively designed for mobile devices before expanding it to the desktop [28]. The mobile-first strategy primarily acknowledges the overwhelming shift towards media consumption on mobile devices. Since it’s generally more difficult to condense or reduce functionality from the desktop to mobile devices, mobile-first strategies help to ensure that mobile experiences aren’t lacking.

For our purposes, it’s still important to recognize that KDKit is designed primarily for mobile audio applications. Concepts from mobile-first web design are non-arbitrary concerning the granularity of interface objects. The level of detail, gestural interaction, default values, and sometimes even the actual function of high-level UI objects is designed mobile-first rather than relying on what might be the expectation in other audio applications [29].

For example, it’s typical to display the value of a slider in a static position if at all. It’s what we would expect on a physical interface, and it’s what we’re used to with other digital interfaces. Placing the value next to the slider requires valuable mobile screen real estate and hiding the value entirely is not always ideal for digital audio applications. Placing the value on the slider itself may seem attractive and could work well for a desktop application, but, on a mobile device, the changing value is often blocked by your hand. KDKit sliders come with a built-in function to slide the text label out from under your thumb as the slider is being touched. This is an example of how KDKit leverages microinteractions as a major aspect of mobile design [30].

Another example of mobile-first design can be found in the collection of layout functions included with KDKit. Creating a complete interface typically involves positioning UI elements in the coordinate spaces of various container views. Learning how to manage
Figure 2.5: Text on the thumb of KDSlider moves when touched. This is a mobile-first microinteration to ensure a user can see the text that would normally be under their finger.

these spaces is unavoidable in most cases, but it’s also true that many mobile interfaces feature the same kinds of elements such as scrolling, full-width lists. Since this is such a prevalent way to organize menus and information throughout the mobile design world, KDKit simplifies the process for new coders by providing a pre-built, reusable solution.

If we were to create a full-width scrolling list using Apple’s UIScrollView class, we would need to manage the exact coordinates of each item or write a function to determine the coordinates as needed. While this is something a new coder should eventually learn to do, that does not mean it has to be a barrier keeping them from creating such a menu. KDScrollView includes the ability to append this content simply by managing the order views are added rather than coordinates and geometry. In figure 2.6, KDKit is used to create a scrolling window with three differently colored full screen views.

Even though KDKit’s process is more accessible than the typical method (using UIScrollView), it still affords complexity and learning potential when building this into an actual list or menu. The individual items would likely be more complete entities than a solid color block, and a new coder would need to organize and manage code for the geome-
func scrollViewExample() {
    // Create a vertical scrollable view. */
    let sv = KDScrollView(self.view.frame, .vertical)

    // Have some content that will be larger than the scrollable view. */
    let f1 = KDView(sv.frame)
    let f2 = KDView(sv.frame)
    let f3 = KDView(sv.frame)

    f1.setBackgroundColor(KDColor.blue)
    f2.setBackgroundColor(KDColor.green)
    f3.setBackgroundColor(KDColor.red)

    // Add the content to the scrol view. */
    sv.append(f1)
    sv.append(f2)
    sv.append(f3)

    sv.addToSuperview(self.view)
}

Figure 2.6: KDKit makes it simple to create a scrolling view out of multiple conjoined views. KDScrollView.append() calculates the geometry to place consecutive, connected views in the scrolling direction.

However, the overall complexity of learning to create this prevalent mobile interface element is reduced and redistributed with KDKit. The code for individual elements can be grouped independently without need to reference the size and position of a previous element in order to place them consecutively, but it is still necessary to understand view geometry and its hierarchy in the view stack.

2.4 Hierarchical Class Structure

A major aspect of object-oriented programming is class inheritance. KDKit conforms to this expectation but with some quirks. Since KDKit is built on top of Apple’s UIKit in order to ensure it is compatible with anything iOS, there is no single entry point for creating the entire UI library.

One of the goals of KDKit is to ensure objects with similar purpose and visual style should have similar features and can be invoked with similar code syntax. Theoretically, this is a primary goal of class inheritance, but UIKit classes have branched so thoroughly that it’s not always clear how to accomplish a design goal if you are not already aware
of all potential solutions. For example, even though a UIButton inherits from UIControl which inherits from UIView, customizing the visual style of a UIButton and UIView can be different in ways that may muddle the surface level benefits of the inheritance. While the hierarchy is technically efficient, learning to customize either is a different experience. Also, adding to and managing physical interactions on a UIView is different from a UIButton which conforms to the UIControl delegate protocol. KDKit addresses this kind of issue with an expanded low-level class hierarchy and its own informal protocols.

KDKit approaches class inheritance with the path of least resistance rather than primarily prioritizing theoretical code or performance efficiency. For example, KDToggle inherits from KDButton which inherits from KDView because each subclass fully functions by only adding to or overriding properties from its superclass. There’s no structural difference, only featured differences. Unlike Apple’s UIButton and UIView which are familial but functionally far-removed, KDButton and KDView have a more direct and obvious relationship. Everything we know about customizing the visual style of a KDView applies directly to both KDButton and KDToggle, and both also use onTouch callbacks just as KDView does when invoked to add gesture recognition. In opposition, UIView requires us to add touch interaction either using the UIGestureRecognizer classes or the global touch functions, UIButton prefers we use or override the UIControl delegate functions, and visual customization requires familiarity with UILayer and UIButton’s non-obvious properties.

While it would be superb if KDKit classes could all follow this pattern, a problem arises when branching out to classes with more specific uses. UIKit classes all inherit from UIView. While it would be theoretically optimal if KDKit classes could have a similar structure, the path of least resistance requires a different approach. For example, it would be possible to recreate the functionality of UIImageView by having KDImageView inherit from KDView with added custom solutions that mimic the UIKit version, but this method is redundant in a way that is more prone to UIKit incompatibilities. In order to ensure a scalable learning experience that is fully compatible with UIKit, KDKit UI elements that do
not neatly inherit in the same way as KDView, KDButton, and KDToggle instead inherit from their UIKit counterpart. For example, KDImageView is a subclass of UIImageView, KDScrollView is a subclass of UIScrollView, and KDLlabel is a subclass of UILabel. Since these classes are not descendants of KDView, KDKit is designed with an informal protocol to ensure similar use and functionality can be expected of each.

Various elements may have different requirements to achieve appropriately similar behavior, so, rather than enforcing a strict protocol that may then include properties and functions that are not descriptively named, KDKit classes conform to a set of expectations with names that are descriptive but predictably identifiable such that Xcode’s code completion fills in the blanks. Less granular objects are more likely to have more specific function names, but they always begin with an expected identifier such as set or onTouch, and they always share the same data types where appropriate to ensure interoperability.

2.5 Dot Syntax Scripting

In Swift, an instance or class must usually be expressed immediately before each function call. This is typical for many languages, but the verbosity of this sort of code can make it difficult to visually search through large code bases. It also means that automated code searches will return several matches—perhaps dozens, hundreds, or even thousands—for every class instance. This can make debugging a tedious process, especially for new programmers.

KDKit objects define their functions as @discardableResult and, even when it is not a necessary aspect of the function, always return a copy of itself. This allows each successive instance function to be called with dot syntax scripting simply by stacking the dots in vertical alignment (or successively in one line). Since each function returns a copy

---

14 Unlike Objective-C where some types such as Float, Double, and CGFloat are interchangeable, Swift will throw an error if the type does not exactly match expectations. In accordance, KDKit includes some higher-level custom data types in order to reduce the potential for error. For example, KDSHadow is a type structure (struct) that provides a single entity to ensure value types for the properties of a reusable shadow are always correct. This is a pattern an experienced programmer might use on their own, but KDKit exposes the concept to new coders as a feature of the toolkit rather than as a lesson to be learned, recalled, and implemented.
Figure 2.7: Comparison showing how KDKit’s dot syntax scripting reduces code verbosity, improves visual organization, and makes text searches more efficient.

of itself, each function is aware of the instance and type being referenced, and, since the functions are labeled as `@discardableResult`, the compiler allows the final function in the chain to terminate without error by throwing out the reference.

In figure 2.7, since the local variable name `view` could potentially exist for dozens of views throughout the app, KDKit’s dot syntax makes it easier to visually identify that instance in any given code block, and code searches will provide significantly less matches to navigate.

This feature has a very small processing footprint since every function call requires an extra decision to determine what to do with the returned result. Given the power of modern devices, it is rare this trade-off will have an actual performance impact, and the benefits of the code readability, navigation, and organization of dot syntax scripting was favored in KDKit.

```swift
func swiftVersion() {
    let view: UIView = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
    view.backgroundColor = UIColor.red
    view.layer.borderColor = UIColor.black
    view.layer.borderWidth = 3
    view.layer.cornerRadius = 12
}

func kdkitVersion() {
    let view: KDView = KDView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
    view.backgroundColor(KDColor.red)
    .setBorderColor(KDColor.black)
    .setBorderWidth(3)
    .setRoundness(12)
}
```
2.6 Technical Descriptions for Select Classes

Functional descriptions and technical use for KDKit objects can be found in the official documentation. The following section will provide further insight into the design and development of select classes, but these details are not required knowledge to use KDKit.

2.6.1 KDView

KDView is the most basic UI object in KDKit and defines the base protocol for all others. It is a subclass of UIView and has all of its uses and capabilities. One feature of KDKit objects is to add two new optional properties to categorize and identify instances of a class. It can be useful to be able to group individual instances of UI objects for future reference, but avoidable lists of global variables aren’t efficient practice and managing large sets of key-value pairs or multi-dimensional arrays can be daunting for new coders. Since all UI objects must be added to the view controller’s array `self.view.subviews` in order to appear on the screen, KDKit simply adds an extra integer identifier called `idNum` and a string identifier called `name` that can be used along with the `tag` property available to all UIKit objects. Then, it’s a much simpler task to loop through the single view controller array and match identifiers to locate specific objects.

These properties may be the most basic benefit of KDKit, but, since the feature can be expected for all KDKit classes, they provide new coders with an accessible and recyclable tool for problem solving. Even though they might be given an overview of classes and class extensions, it requires a more abstract understanding of their own code structure to create similar solutions. These simply named properties can be used to organize collections in a more familiar manner.

KDView also owns functions to add and manage gesture recognizers without the need to define them separately or rely on delegate methods. Any of Apple’s provided UIGes-

\footnote{The URL https://notnatural.co/kdkit/documentation/ is reserved for KDKit’s official documentation.}

\footnote{In Swift, an optional is a data type that can be `nil`. Swift data types can not be `nil` unless wrapped as an optional, which allows for the `Optional.none` (or `nil` literal) and the `Optional.some` (has a wrapped value) cases. The compiler always assumes a data type can not be `nil` unless it is explicitly declared as `optional`.}
featureRecognizer’s can be added to the instance simply by calling the appropriate function, and the action method is set to the `onTouch` property as a callback. Since an instance of KDView can have multiple gesture recognizers but owns a single callback method, we can use normal Swift solutions to test the sender’s `gestureRecognizer.type` property and identify specific gestures.

```swift
func uvivewVersion() {
    let view: UIView = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
    let tap: UITapGestureRecognizer = UITapGestureRecognizer(target: self, action: #selector(didTap(sender:)))
    view.addGestureRecognizer(tap)
}
func didTap(sender: UITapGestureRecognizer?) {
    print("Tap")
}
```

Figure 2.8: Adding a gesture recognizer with delegate method in Swift. The delegate method must be its own separate function, and the UIGestureRecognizer instance must reference it as a selector. The selector name can not be filled in with code completion.

```swift
func kdkitVersion() {
    let view: KDView = KDView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
    view.addTap().onTouch = { this in
        print("Tap")
    }
}
```

Figure 2.9: Adding a gesture to a `KDView` with callback function. The class handles adding its own gestures with `.add` functions, and the action is defined in the `.onTouch` callback.

The KDView protocol also organizes access to style properties for more convenient use of Xcode’s code completion. Instead of having to know or search for differently named properties such as `backgroundColor`, `borderWidth`, or `shadowRadius`, all visual style properties can be set with functions that begin with the identifier `set`. With an instance of KDView named `view`, simply type `view.set` and code completion will list all available style properties.

Finally, the KDView protocol includes functions to simplify and expedite geometry changes. These include simplified access to individual aspects of the view origin, size, and
Figure 2.10: Test for multiple gestures using basic Swift solutions.

center point, and functions to add margins (similar to CSS for a web browser). This is meant to aid rapid prototyping by allowing UI objects to be placed using flush coordinates, and then a container can be iteratively resized simply by adding fixed-width margins.

2.6.2 KDColor

KDColor extends UIKit’s UIColors with an extensive collection of RGBA color codes popular in modern design. Currently, there are twenty-one base colors each with nine shades. KDColor also includes functions allowing hex codes in place of RGBA and to invert a color to find its literal complement.

KDColor helps avoid workflow interruptions with quick, intuitive access to these modern colors. This can be useful for prototypes that may change significantly, especially since the perceived success of a trial or example can often be influenced by its visual appeal or polish. Something as simple as changing a bright solid red to a flat shade can have long lasting effect on students or clients [2].

Accessing the base color works just as Apple’s UIColor, but shades of each base color
can be created by calling the related function (named after the plural of the base color) and passing an integer 1-9. The shades are graded on a scale from lightest to darkest. Calling the base color alone is a shortcut that returns the value for shade 5.

```swift
func kdsColor() {
    let olive: KDColor = KDColor.oives()
    let lightAzure: KDColor = KDColor.azures()
    let azure: KDColor = KDColor.azure()
    self.view.backgroundColor = azure
    // KColor is a struct whose functions return a UIColor
}
```

Figure 2.11: KDColor provides a robust selection of modern design colors.

### 2.6.3 KDColorTheme

KDColorTheme is a data type and protocol for managing and changing colors for a collection of UI objects. A KDColorTheme struct holds colors for background, text, border, shadow, contrast, and accent. Each is optional and passing nil will result in the color being ignored when the KDColorTheme is passed to a UI component.

KDKit UI objects all conform to the KDColorTheme protocol, but some objects do not use all six colors. For example, a KDView can only have a background, border, and shadow and will ignore contrast, text, and accent whether they are nil or not. More complex UI objects such as the KDKeyboard use all six properties. Passing a KDColorTheme to a UI object will update both the internal and visible state.

```swift
func kdsColorTheme() {
    // KDColorTheme holds UIColors (KDColor functions return custom UIColors)
    let theme: KDColorTheme = KDColorTheme(background: KDColor.steel(),
                                           text: KDColor.steel().invert(),
                                           border: KDColor.blue(),
                                           shadow: KDColor.blue(),
                                           contrast: UIColor.white,
                                           accent: UIColor(red: 255, green: 156, blue: 156, alpha: 1))

    // Create a KDView and set the color theme
    let view: KDView = KDView(frame: self.view.frame).withColorTheme(theme)

    // Create a keyboard, set the margins to 20 points on all sides, and set the color theme
    let keyboard: KDKeyboard = KDKeyboard(style: self.view.frame)
    .setMargin(20)
    .setCenter(self.view.center)
    .setColorTheme(theme)

    // Add the view to the view stack
    self.view.addSubview(view)
    keyboard.addSubview(self.view)
}
```

Figure 2.12: KDKit UI classes conform to the KDColorTheme protocol.

---

17 In that case, the white keys are set as the contrast color, and the black keys are set as the accent color.
This feature was added to KDKit in order to aid rapid iterative design and development. Objects in the view stack that are categorized using the tag, name, or idNum properties can be looped through and a KDColorTheme applied en masse. This enables quick, global color changes throughout the app. Also, by understanding proper code patterns to take advantage of KDColorTheme, a new coder is exposed to practices that reduce potential for error and encourage clean, organized code.

```swift
func kdcolorthemeloop() {
    /* Loop through all views in this view controller’s stack */
    self.view.subviews.forEach { view in
        /* Declare the target view as a KView if possible */
        guard let target: KView = view as? KView else { return }
        /* Set the color theme for all targets named "button" */
        if target.name == "button" { target.setColorTheme(KDColorTheme(background: .black, border: .red)) }
    }
}
```

Figure 2.13: KDColorTheme was implemented to aid in app-wide color updates. This example shows how views in the view stack can be updated with a forEach loop.

### 2.6.4 KDRoll

KDRoll is a random number manager. The most basic function allows a user to generate random numbers using a die rolling metaphor. The class function Roll.D(6) will generate a random integer between 1 and 6. This metaphor is used to help clarify the number range since normal dice do not have a zero face. Any valid integer value can be passed to the Roll.D() function, and the result can then be scaled and converted to any appropriate range and data type. KDRoll also includes quick access functions to generate random numbers of various types with normalized values.

KDRoll also utilizes Apple’s GameKit in order to provide advanced random number features. Calling Roll.D() actually starts or adds to a random distribution table. Roll.uniformD() will manage a uniform distribution table, and Roll.gaussianD() will manage a gaussian distribution table. In each case, the Roll singleton globally manages the tables. If we were to desire different independent tables, we could simply create instances of KDRoll and call the same functions on each instance. Also, since the values are stored in a table, the last rolled value can be retrieved by calling Roll.getDie(),
Figure 2.14: KDRoll is a random number generator as a dice metaphor. Roll.getUniformDie(), or Roll.getGaussianDie(). The standard deviation and mean can be retrieved by adding .deviation or .mean respectively.

Another KDRoll capability is to handle multiple distribution tables of the same type with a single instance. For this, we can call Roll.addDie() to add a die to the metaphorical dice bag. Then, when Roll.dice() is called, an array containing a new random value for each die is returned. Dice in the dice bag can have different numbers of sides, but each form of distributed randomness is managed separately.

```swift
func kdrrollTables() {
    /* Roll the dice 10,000 times with gaussian distribution */
    (0...9999).forEach { _ in Roll.gaussianD(100) }

    /* KDRoll dier are instances of GKRandomDistribution which
     * self compute standard deviation and mean with each roll */
    let deviation: Float = Roll.getGaussianDie().deviation
    let mean: Float = Roll.getGaussianDie().mean
}
```

Figure 2.15: Instead of Swift’s standard random number functions which return a single number, KDRoll uses the GameKit class GKRandomDistribution which manages random distribution tables.

2.6.5 KDMotion

KDMotion manages Apple’s CMMotionManager class and provides quick access to tuned device motion data. CMMotionManager enables and processes data from the accelerometer, magnetometer, and gyroscope in order to produce Euler angles (pitch, yaw,
Figure 2.16: Multiple dice with independent tables can be rolled at once. roll) and user acceleration, but its data must be polled and tuned in order to make it useful to media applications. KDMotion handles initialization, creates and manages a polling timer, provides a callback function rather than a delegate to access the data, filters the data flow to reduce jitter, and scales the raw data to a normalized range (0-1). A major benefit of using KDMotion is that this entire process is automated and can be instantiated with just a couple lines of code.

```swift
func kdrollbag() {
    /* Add ten 20-sided die to the bag */
    (1...10).forEach { _ in Roll.addDie(20) }

    /* Roll all ten dice and return the results in an array */
    let results = Roll.dice()
}
```

Figure 2.17: KDMotion makes it simple to use motion data.

In addition, KDMotion includes some customization and convenience utility. The double precision of the polled data can be set along with the polling interval. KDMotion can also manage a set of views to be rotated along with the device. Simply pass a KDView or UIView to `motion.addToRotatedViews()` and KDMotion handles all of the geometry.
transformations for the managed views. This feature was originally meant to provide visual feedback that confirms polled data is properly synced with the physical device orientation, but it could also be useful for stylistic visual effects or gamification.

2.6.6 KDLabel

Creating visible text is a fairly basic requirement for mobile apps. Basic formatting would be accomplished with the Swift UILabel and String classes, but fitting text in the visible frame requires trial-and-error and adding text styles is an involved process necessitating NSMutableString instead of String applied to the UILabel property `self.attributedText` instead of `self.text`. KDLabel was designed with the assumption that whatever benefits the UILabel design may afford are not often worth the hassle. All we probably want is to put an underlined title on the screen.

KDLabel provides the resizing functions `fitTextToFrame()` and `fitFrameToText()` to simplify geometry. `fitTextToFrame()` will automatically resize the text so it fits the given frame. `fitFrameToText()` will wrap the text to fit the frame width and expand the frame height to fit the wrapped text. Both functions can be included in KDLabel initializers.

```swift
func simpleLabelExample() {
    // Create a label where the text auto-resizes to fit the given frame. /*
    let fittedLabel = KDLabel(fitTextToFrame: CGRect(x: 0, y: 0, width: view.width, height: 50),
        fontName: KDfont_regularFontName,
        text: "Text this label out with a whole sentence.",
        alignment: .center)
    
    // Create a label where the text auto-resizes to fit the width of the frame, but the frame height expands to accommodate the wrapped text. /*
    let expandedLabel = KDLabel(fitFrameToText: CGRect(x: 0, y: 20, width: view.width, height: 50),
        fontName: KDfont_titleFontName,
        fontSize: KDfont_fontSize1L,
        text: "Text this label out with a whole sentence.\n\nAdd another sentence to make it longer.\n",
        alignment: .center)
}
```

Figure 2.18: KDLabel functions to aid with text label geometry.

KDLabel also includes data structures for each type of attributed text style and manages the use of String or NSMutableString as appropriate (see figure 2.19). Adding attributes to an entire label is as easy as calling `set` functions similar to other KDView properties, and the attributes can be shown or hidden independently for the cases where the styles are used for toggleable emphasis. KDLabel also provides a function to add individually styled attributes to substrings in the label.
2.6.7 KDKeyboard

KDKit is designed to provide useful and accessible high-level classes, and, for music, the most obvious UI component is probably a piano keyboard. Using KDKeyboard is as simple as creating any other view. It’s also programatically drawn when initialized—there’s no need to include any images or other media—and is appropriately touch interactive with just six lines of code.

```swift
func keyboardSimpleExample() {
    let keyboard = KDKeyboard(self.view.bounds)
    .addSubview(self.view)

    keyboard.onKeyDown = { key in
        print(key.value + " on.")
    }

    keyboard.onKeyUp = { key in
        print(key.value + " off.")
    }
}
```

Figure 2.20: Basic example using KDKeyboard.

![Figure 2.19: Text styles added to KDLabel.](image)

![Figure 2.20: Basic example using KDKeyboard.](image)
KDKeyboard is subclassed from KDScrollView and conforms to the KDView protocol, so normal KDScrollView and KDView properties can be accessed as expected, but unique properties can also be styled. Figure 2.21 shows how to customize touch properties, octave range, and key size and color. Touch interaction is handled directly with the UITouch class rather than UIGestureRecognizer and aftertouch or glissando are available as optional modes. KDKeyboard owns callback functions for pressing down on a key (.onKeyDown), letting go of a key (.onKeyUp), and aftertouch (.onAftertouch).

```swift
func keyboardExtendedExample() {
    let keyboard = KDKeyboard(CORect(x: 0, y: 40, width: self.view.frame.width, height: 120))
    .setHorizontalMargins(15)

    // Change the color of the keys.
    keyboard.setKeyColors(forKeytype: .white, background: KDColor.azure, border: KDColor.amber)

    // Customize the range and size of keys.
    keyboard.set(lowestOctave: 2, numberOfOctaves: 4, numberOfVisibleWhiteKeys: 6, blackKeyWidthRatio: 0.33, blackKeyHeightRatio: 0.7)

    // Start the view focused on a middle octave.
    keyboard.scrollToOctave(3, animated: true)

    // Require two fingers to scroll and turn on aftertouch detection.
    keyboard.setNumberOfTouchesToScroll(2)
    keyboard.touchMode = .aftertouch

    // Optionally, switch to glissando mode which lets you slide across keys without lifting your finger.
    keyboard.touchMode = .glissando

    // Customize KDScrollView properties.
    keyboard.setTextColor(KDColor.amber)
    .setLabelFont(KDFont.defaultFont())

    // Customize KDView properties.
    keyboard.setBorderColor(KDColor.amber)
    .setRoundness(keyboard.getRadius())
    .addToSuperview(self.view)
}
```

Figure 2.21: Example customizing KDKeyboard.

### 2.6.8 KDAnimate

There are multiple methods for handling animations with Swift, but the recommended method for most UI components is with the `UIView.animate` class function. For simple animations, this syntax is fine, but figure 2.22 shows how creating more complex animations or chaining together a series can quickly become cumbersome and difficult to decipher as

---

18 Recently, Apple has added UIViewPropertyAnimator as the successor to `UIView.animate`. KDAnimate will be updated when appropriate.
animation blocks are embedded within each other.

KDAnimate provides a less verbose, simple syntax that hides redundancies and is more human-readable. Animation options are stored as mutable instance properties rather than having to pass the same values into every animation block. Also, since UI animations block the main thread, KDAnimate automatically wraps each call to be queued for asynchronous background execution with grand central dispatch (GCD).\footnote{Grand central dispatch is the thread manager for iOS. Swift includes several functions for creating, managing, and assigning tasks to various queues. KDAnimate places all calls in the global background queue in the order they are made. This works well for individual UI animations, but it may not be controlled enough for constant, recurring, or overlapping/interrupting animations. There are future plans to add the option to use a more precisely ordered and timed serial queue to KDAnimate.} The example in figure 2.23 is the KDAnimate version of figure 2.22. Even though it adds a couple more steps to the animation, it still only requires five statements in eight lines of code to display the view with a pop animation, move it diagonally, rotate ninety degrees, hide the view with an inverted pop, and finally remove it from the view stack.

KDAnimate conforms to the KDKit protocol allowing function calls to be chained using dot syntax, but it uses the syntax in a different manner than other KDKit classes. Instead of functions being named with the same expected identifiers, KDAnimate functions are named in a manner meant to aid the organization of compound or complex animations. The name of the function is descriptive of the animation, parameters are labeled with active language, and the completion handler is simply named then. Under this naming convention, chained functions read as a sentence-like series of events rather than abstract code commands.

KDAnimate includes functions for common basic animations such as fade, scale, rotate, and translate. Each of these animations are performed on the animation layer using core animation transformations rather than changing the actual geometry of the object. This means animations can be reversed or reset by setting an object’s animation layer to its identity rather than having to backtrack the animation. In the case we may want to animate a position change of an object’s actual geometry, KDAnimate includes move as
func UIViewAnimation() {
    /* Create a frame with 50 point margins */
    let margin: CGFloat = 50
    let rect = CGRect(x: 0, y: 0,
                      width: self.view.frame.size.width - margin,
                      height: self.view.frame.size.height - margin)

    /* Create a view with auburn background color and center it in the main view. */
    let square = UIView(frame: rect)
square.backgroundColor = UIColor(red: 255, green: 87, blue: 34, alpha: 1)
square.center = CGPoint(x: self.view.frame.width / 2, y: self.view.frame.height / 2)
square.view.addSubview(square)

    /* Animations are thread-blocking, so run them on a background thread instead of main. */
    DispatchQueue.global(qos: .background).async {
        DispatchQueue.main.async {

            /* First animation will make the view fade in and grow, so we have to set the initial state. */
            let t = CATransform3DScale(square.layer.transform, 0.1, 0.1, 1)
square.layer.transform = t
square.layer.opacity = 0

            /* UIView.animate blocks with options. */
            UIView.animate(withDuration: 0.5,
                            delay: 0,
                            usingSpringWithDamping: 1,
                            initialSpringVelocity: 1,
                            options: .curveEaseOut,
                            animations: {

                            /* Animate the view returning to normal size and fade in. */
                            square.layer.transform = CATransform3DIdentity
square.layer.opacity = 1

            }, completion: { (finished: Bool) in

                /* Once the previous animation is complete, start a new one that moves the view. */
                UIView.animate(withDuration: 0.5,
                                delay: 0,
                                usingSpringWithDamping: 1,
                                initialSpringVelocity: 1,
                                options: .curveEaseOut,
                                animations: {

                            let t = CATransform3DTranslate(square.layer.transform, 20, 25, 0)
square.layer.transform = t

            }, completion: { (finished: Bool) in

                /* Continue embedding animations as needed. */

            })

        })
    }
}

Figure 2.22: UI animation example using UIView.animate().
Figure 2.23: UI animation example using KDAnimate.

an alternative to `translate`. KDAnimate also includes a function for custom animation blocks so it has all the capabilities of `UIView.animate()`.

Finally, KDAnimate also features a growing collection of pre-built, complex animations such as tilt, jiggle, shake, and flash. The nature of these animations means there is no standard method for implementing them, but each function is a custom implementation using KDAnimate. Essentially, they are created by having KDAnimate create child instances of itself that reference the parent properties. Each of these advanced animations have pre-designed defaults to ensure they are simple to use, but they can also be customized with optional parameters.

### 2.6.9 KDEasyPD

KDKit includes two audio solutions, the first of which is a wrapper for the third-party library libpd [sic]. With libpd, Pure Data (PD), a graphical programming environment comparable to Max, can run as an embedded application on other platforms. KDEasyPD simplifies the setup of libpd for iOS and helps manage advanced features. Also, libpd is written for Objective-C and does not have a documented Swift implementation. KDEasyPD resolves this issue by wrapping libpd functionality in Swift-compatible functions and exposing them to the rest of KDKit.

Currently, KDEasyPD requires libpd to be added as a standalone dependency. This
means a user will still have to import the libpd source project into their KDKit project and refer-
ence the correct path in the project build settings.\footnote{KDKit documentation comes with simplified step-by-step instructions for handling this process.} Once libpd is properly included and a PD patch added as a resource, KDEasyPD manages all other aspects of setup, validation, and error handling with a single line of code.

```swift
func pdExample() {
    // Initialize with a guard statement to avoid fatal errors. */
    guard let pd = EasyPD(patch: "EasyPD_Demo.pd") else { return }

    // Pass values to receive objects in the patch. */
    pd.sendBang(toReceiver: "click")
    pd.send(350, toReceiver: "freq")
}
```

![Figure 2.24: Basic example using KDEasyPD to manage libpd.](image)

In order to facilitate efficient polyphony, libpd is capable of dynamically loading multiple instances of the same patch. Objects named with a \$0 variable are assigned a number identifier in its place, allowing communication with individual patches whose objects are nominally the same. With KDEasyPD, this feature is made more accessible by having the \$0 assignment stored when patches are loaded rather than having to sort and access them individually. After loading patches in the desired order with `pd.addPatch(toPatches: "patch_name"), \$0 identifiers can be accessed with `pd.dollarZeroForPatch(at: index)` which returns the \$0 value as a string to be incorporated with send and receive names.

### 2.6.10 KDOrchestra and KDInstrument

The second audio solution in KDKit consists of protocols and classes for the third-party library AudioKit, an implementation of Csound for iOS.\footnote{Csound is an audio synthesis library written in C and modeled after Max Mathews’ MUSIC N series. Since C code can be compiled and run on iOS, AudioKit is able to provide Swift implementations of its various instruments.} AudioKit is already available in Swift and has detailed online documentation, but it is not designed for inexperienced coders. There is a live coding playground version which has more beginner friendly syntax, but its
code can not be compiled verbatim within actual apps. In order to improve accessibility and better afford rapid prototyping, KDKit instruments conform to a protocol that reduces or removes odd discrepancies between similar entities, facilitates AudioKit’s per instrument audio initialization and signal chains, simplifies instrument initialization and setup code, and ensures AudioKit compatibility with other components of KDKit.

Instead of extending AudioKit classes and unlike KDKit UI classes, all KDKit instruments and effects inherit from the superclass KDInstrument. Even though KDOscillator may look and function similar to AudioKit’s AKOscillator, KDOscillator is actually a managing class and AKOscillator is one of its properties. The KDKit protocol was designed in this manner because AudioKit sound generators do not all have the same type of output nodes. KDInstrument enforces that each sound generating entity’s signal chain terminates with AKAmplitudeEnvelope passed to AKMixer. This ensures that every KDKit instrument will have a `self.adsr` that affects the entire instrument and `self.volume` that is the master level. It also standardizes the output node type and expectations concerning what happens when a sound generator is turned on or off in order to facilitate automation.

AudioKit requires initialization with the `do`, `try`, and `catch` error handling pattern. This requirement isn’t terribly complex, but it’s also not likely to be the first topic in your standard beginner’s guide to Swift. In the simplest cases, initialization code is basic enough to be copy/pasted between projects, but with multiple signal chains it’s also necessary to mix the proper outputs into a single node to assign to `AudioKit.output`. KDOrchestra manages all aspects of this process.

Since all KDInstruments’ terminating output nodes are the same Swift type, KDOrchestra is able to reliably automate the creation of the final mixed node. Once we’ve defined any number of instruments of any type, we can create an instance of KDOrchestra by passing each instrument in an array. Then, calling `orchestra.onStage()` will connect all instruments to the KDOrchestra mixer and handle proper AudioKit initialization.

Once activated, KDInstruments can be played with `instrument.play(midiNote:)`
func kdkitAudioKit() {
    // Create a couple oscillators. */
    let osc1 = KDOScillator(.sine)
    let osc2 = KDOScillator(.square)

    // Create an orchestra that includes our oscillators, and get it ready to play. */
    // This handles all AudioKit setup and signal routing with KDInstrument. */
    let orchestra = KDOrchestra(withInstruments: [osc1, osc2])
    orchestra.onStage()

    // Start playing some notes. */
    osc1.play(midiNote: 36)
    osc2.play(midiNote: 42)

    // Wait 2 seconds and then stop all orchestra instruments. */
    // *waitFor(duration:then:) is a KDKit helper function for GCD. */
    orchestra.waitFor(duration: 2, then: {
        orchestra.stop()
    })
}

Figure 2.25: Basic example using KDKit’s AudioKit classes.

or, in some cases, `instrument.play(frequency:)`. This will start the ADSR envelope and maintain the sustain level until `instrument.off()` is called. `orchestra.off()` calls `instrument.off()` for all of its referenced instruments.

KDOrchestra also inherits from KDInstrument, so we can expect this master output to respond to `self.adsr` and `self.volume`. This means that KDOrchestra can be used to manage groups of instruments and that these groups can be passed into a master KDOrchestra as individual instruments themselves. Since `orchestra.onStage()` attempts to initialize AudioKit but continues KDOrchestra initialization even if unable, the only requirement is that the master KDOrchestra be the last to make the call to ensure the initialization includes all of the correct outputs.

### 2.6.11 KDFXUnit and KDFXChain

In addition to sound generating instruments, AudioKit includes several different effects classes. KDKit implements these effects with KDFXUnit and KDFXChain, which are also both subclassed from KDInstrument. KDFXUnit essentially manages effects in the same manner KDInstrument manages instruments, but it adds a function to create an intermediary signal chain. KDFXUnit classes also declare their parameters with their own
custom data type. This encourages code patterns that group related values and ensure they are descriptively named based on their function.

Even though KDFXUnit classes inherit from KDInstrument, they do not generate sound on their own and are not added directly to a KDOrchestra. Instead, KDFXChain is used to create a new instrument entity via a signal chain from an instrument to an effect or series. The combined signal chain is added to KDOrchestra, but play commands are still enacted on the base instrument.

```swift
func delayExample() {
    /* Create a KDFXUnit for the AudioKit delay effect. */
    let fx = KDDelay(0.05)

    /* Create an audio source by playing a sound file. */
    let player = KDPanner("testAudio.wav")

    /* Create a new compound instrument chaining the player to the delay. */
    let fxChain = KDFXChain([player, fx])

    /* The orchestra instrument is the KDFXChain, not the sound generator. */
    let orchestra = KDOrchestra(withInstruments: [fxChain])
    orchestra.onStage()

    /* But sound is still controlled via the sound generator. */
    player.play()
}
```

Figure 2.26: KDFXChain creates a signal chain from a KDInstrument through any number of KDFXUnit.

### 2.6.12 KDFMOscillatorBank and Other Generators

AudioKit includes several unit generators in both mono and polyphonic versions. KDKit provides an implementation for each one, and they are all modeled in similar fashion. For example, creating a polyphonic frequency modulation (FM) synthesis instrument is nearly as simple as creating a monophonic sine wave oscillator. The only differences are the parameter requirements necessary for FM (see figure 2.27).

### 2.6.13 KDScore

Similar to KDAnimate, KDScore provides a more readable syntax for managing an ordered series of asynchronous events. There are two methods of using KDScore: `score.on()`
Figure 2.27: KDFMOscillatorBank is nearly as simple to use as KDOscillator.

or `score.start().then(after:)`. Calling `score.start()` begins a series of events with individual timing, and each successive event is chained by calling `then(after:)`. Both include a parameter to pass a callback function, and `then(after:)` also requires a duration in seconds. KDScore takes the entire series and queues the commands in the background thread for asynchronous execution.

Calling `score.on()` starts a repeating loop. `score.on(time:)` can be used to pass a duration in seconds, but `score.on(beat:)` requires an instance of KDMeter. In either case, the callback function is run after every interval until `score.stop()` is called. `score.on()` can be used to wrap a timed series created with `score.start().then(after:)`.²²

Currently, `score.start().then(after:)` queues in a background thread using the GCD function `DispatchQueue.main.asyncAfter()`. This means the whole series is placed in the queue at the same time, but a system timer will appropriately delay each later command. Being in the background thread leaves these timers at the mercy of thread priority, and there can be noticeable rhythmic discrepancies over long periods. GCD only guarantees that the block will be executed after waiting at least as long as the provided duration. `score.on()` is timed with its own scheduled timer and has a lesser but similar problem. Since the accuracy of the Swift Timer class is dependent on run loop availability, a very short duration may have a noticeable timing issue due to its repeating nature.

Potentially more reliable options are being explored for both cases—including ones that

₂²`score.start().then(after:)` can not be interrupted in its current implementation.
Figure 2.28: KDScore can be used to create a timed, ordered script of events or repeat a command at regular intervals.

may allow `score.start().then(after:)` to be interrupted—but it is important that KD-Score remain simple and accessible. For now, it is recommended that KD-Score be used to create shorter series (e.g. an arpeggio or ostinato) rather than a complete piece.

2.6.14 KDPlist

KDPlist provides a simple persistent data management solution. NSUserDefaults and Core Data are the recommended data solutions, but Core Data has a steep learning curve. KDPlist is similar to NSUserDefaults, but KDPlist is designed to improve workflow around large sets of simple data such as synthesizer settings.

When an instance of KDPlist is created, it searches for a `.plist` file in the app’s main bundle that matches the string passed as a parameter. If it finds a match, the object can then be used to manage that data, and if it doesn’t find a match, KDPlist will create the file. Each instance of KDPlist then represents a mutating data set which is operated on as the instance itself. Rather than writing to disk every time a change is made, the
instance will hold changes until explicitly told to save or load. KDPlist holds data as a serialized, mutable key-value dictionary. Setting or changing data is as simple as passing in a dictionary using `plist.set([:])`. Calling `plist.save()` will then write the data to file. Since the state of any given instance of KDPlist and the file it references can be different, calling `plist.load()` will copy the file state to the calling instance and overwrite any differences.

```swift
func plistExample() {
    /* Create a data store and save some key-value pairs */
    let store = KDPlist(name: "demoStore")
    store.set(
        "testKey": "testValue",
        "numberTwo": 2
    )
    /* Save the store to a local plist. */
    store.save()

    /* If we change the data, this instance is affected, but the persistent file isn't unless we call `save()` again. */
    store.set(
        "testKey": "1111111111",
        "numberTwo": 4, "four",
        "extra stuff": "this will be deleted when the last store is loaded"
    )
    /* Calling `load()` will restore this instance to the data we originally saved to file. */
    store.load()

    /* Currently, KDPlist only works with strings, but we can use normal Swift methods to cast to appropriate types when necessary. */
    guard let two = Int(store.data["numberTwo"])! else { return }
    print("\(two) as \(typeof two)"")
}
```

Figure 2.29: Example using KDPlist to store and load data in persistent storage.

### 2.6.15 Other Classes of Note

The specific classes described in this chapter are only meant to give an overview of KDKit’s design and capabilities. New classes are still being added, but many others already exist including but not limited to:

- **KDSliderView** creates and connects several sliders to a multi-point data source such as the properties of a KDInstrument.

- **KDAActivityIndicator** manages a custom activity indicator.

- **KDWarningLabel** manages reactive pop-up information.

- **KDFlowView** is a re-orderable, editable list with its own managed datasource.

- **KDClock** displays an interactive stopwatch.
• KDMediaPicker allows quick access to sound files in the device music library or the app local storage.

• KDWebUploader wraps and provides a pre-built UI for a third-party solution letting users upload remote files directly to an app using a WiFi connected desktop browser. The browser page is served directly from the mobile app via an IP address connection, so there’s no need to host or access a remote site. Any files can be uploaded to or downloaded from the app local storage with this interface, but the device will only be able to interact with iOS compatible file types.
Chapter 3
Example Mobile Applications Developed with KDKit

Throughout the development process, KDKit was tested for usability by actually using it to create several musical mobile apps. KDKit’s design evolved along with these applications. Each app presented different problems and increasing complexity, and ultimately it was this developmental experience that shaped the technical design of KDKit. Even though KDKit supports the development of various kinds of musical apps, the most obvious mobile music comes in the form of a standalone musical instrument or synthesizer [24], [31]. The specific apps described in this chapter all belong to that category and each marks a major iteration of KDKit.

When creating these apps, it was natural to make comparisons between potential mobile interface designs and extant computer music controller designs. While many of the problems that have existed are similar [32], [33], it was important to consider that the solutions may be very different. Since a mobile device is compact, self-contained, does not require external hardware or sensors, and comes with prescribed non-musical expectations for interaction, as a computer music controller it is not quite like most of its predecessors. This became a guiding principle for the gestural and interactive designs of these KDKit instruments.

Another goal was to ensure these instruments had inherent musicality. This is closely linked to gestural intent, but there was also a focus on the potential for virtuosity [34]. The instruments should be simple to use but should also require time and dedication to really learn to control and play. Once familiar, they should also afford the potential for advanced ability. While there have been examples of using all of the various sensors and capabilities of a mobile device to create performable music [35], [36], [37], [38], [39], [40], the following examples focus on the most obvious—touch screen and device motion via accelerometer and gyroscope—and attempt to expand how we think about the gestural interaction they afford.
3.1 In Sen

In Sen served as a launching point for the development of KDKit. It was the first full featured musical application developed using the toolkit, and the development process was as useful to the development of KDKit as KDKit was for developing In Sen.

The app is named after a traditional Japanese scale typically used for tuning the koto [41]. In Sen features multiple methods for creating and shaping sound, but its basic operation allows a user to play in a scale using pseudo-automated rhythmic playback.

The overall design for In Sen was governed by the idea that smart phone touch screens are inherently different from mouse/keyboard sliders or physical analog inputs [42]. Even though digital inputs are often modeled after physical devices such as buttons, knobs, and faders, it’s clear that using a mouse to manipulate a virtual slider is different from touching and moving a physical fader. The same is true when comparing touch screen sliders and mouse sliders. Even though they are both digital representations, the act of using a touch screen is more tactile than a mouse but differently tactile from an analog fader.

Figure 3.1: The In Sen UI includes several sliders that affect sound in real time.

---

1The koto is a traditional Japanese stringed instrument.
3.1.1 Sound Design

In Sen features a synthesized pitched percussion sound. The instrument was intended to have a generally familiar sound quality at first but then could be manipulated to drastic effect. Pitch is selected using a slider, and sound is played only while the pitch slider is being manipulated. Pitch selection is limited to various scales predefined in the app. The pitch slider plays through the notes of the selected scale.

The tempo and rhythmic pattern are selected in real time using separate sliders. In Sen allows for the tempo to be increased beyond a point where discrete attacks can be heard resulting in more complex waveforms. The sound becomes more like a distorted drone, and the complex waveform can be shaped and manipulated by continuing to change the tempo. Another change can be created by selecting different rhythmic patterns. This leads to various timbre and pitch changes as the spacing of the attacks further shapes the waveform.

Finally, the filters used to create the pitched percussion sound can be manipulated using device motion. This effect is subtle. It wasn’t meant to be a filter sweep or other electronic effect. It works more as a subtle timbral effect similar to vibrato.

3.1.2 UI Design

The UI is a collection of sliders and a couple of knobs meant to use as much of the screen real estate as possible for real time control [43]. Most of the controls are used to manipulate the sound live during performance with only a couple controls set aside for static settings. This was to take advantage of the properties that make a touch screen slider different from other similarly familiar controls since, with a touch screen, there is more than one way to manipulate the slider.

A physical fader is touched with your finger and pushed or pulled to a new position. Similarly, a digital slider/fader controlled with a mouse is clicked and moved. With a touch screen mobile device, there is the option to treat the slider in the same manner as those examples, but consider that you could also leave your finger stationary on the control and
move the device instead. This may be seem overly burdensome at first, but it becomes very useful when creating precise changes. When trying to set the brightness of your screen this amount of precision may be negligible, but, when manipulating sound, it can be important, even musical. By adjusting the pressure on the touch screen, and rolling or twisting the device rather than explicitly pushing or pulling your finger, you can gain more control over the rate and shape of the change of the values provided by that slider which can improve the visual connection between a gesture and the sound [44]. With In Sen, the complex waveforms created at high tempo can be audibly shaped using this technique in a way that would be near impossible with a physical fader or mouse/keyboard control. Also, the act of rolling and moving the device even small amounts will cause the accelerometer to move the filter values. This is an example of how these instruments can create many-to-many mappings via usability design rather than only in programming [42].

3.1.3 Performance and Use

In Sen has been used in performance both as a solo and ensemble instrument. As a solo instrument, it lends well to improvisations that begin reminiscent of traditional Japanese melodies but builds activity until the tempo threshold for creating complex waveforms is broken. Ultimately though, the instrument is well suited for creating subtle dynamic and timbral changes within its sound pallet, but the overall versatility is limited by its scale locking.

*Cherry Blossoms* is an ensemble piece written for In Sen. It is performed by 6-10 players reading off a score with timing and setting markings as well as several colorful cherry blossom graphics of various shapes and sizes. Performers are instructed to play the pitch slider as if they were tracing the outlines of the cherry blossoms and traversing the score from left to right. While the actual selection of pitches is interpretive, the idea was to create familiar sonic shapes and visible gestures that bounced between players.

In Sen was an informative study. Its development cemented the first version of KDKit and gave direction for the second iteration. Concerning usability and performance potential,
it set the stage for designing KDKit objects that serve and integrate with digital sounds to better promote the potential for touch screen devices as musical controllers.

3.2 Lights

Lights is built on much the same principle as In Sen but with emphasis on trying to create a more visually appealing live performance. While it does away with most of the sliders, it is still scale locked with preset scale options. Lights was meant to make more use of the screen real estate for controlling pitch and timbre. In order to create more visual appeal, the UI background color changes via device motion. This creates a color shift effect that coincides with changes in sound.

![Figure 3.2: Lights features a rippling water effect when touched.](image)

3.2.1 Sound Design

Lights is designed to sound similar to a distorted electric guitar. The base sound is generated with phasors and distortion is created using a feedback loop which is then layered with a classic chorus effect. Touching the screen in any position will begin playing a note. A new attack is created either by lifting and touching in a new position or by sliding to the
next note region (top to bottom). Moving left or right within a region changes the amount of distortion by adjusting the gain and duration of the feedback loop. The device motion creates a rough vibrato effect along with changing the background color.

A pop out menu on the side reveals a couple optional sliders which control the root note of the scale and the output gain. The gain slider has a special effect: moving sideways will bend the slider like a string which in turn bends the note, a nod to bending a guitar string.

3.2.2 UI Design

The UI for Lights features a solid colored background with a smoky particle effect overlay. As a user touches the screen, it ripples as if you are touching water. This visual effect is meant as much for the user as it is for potential onlookers [45]. Loading Lights automatically sets the background glow to the highest setting, and the background color shifts as the device is moved. In a dark setting, this creates a shifting light show as the glow reflects off of the user or is projected for an audience. The ripple effect makes touch position and motion clear.

Learning to play Lights requires some effort in order to get a feel for note positions for a given scale. There are optional guide lines, but the process and effort required to learn note positions is similar to learning any other pitched instrument. For some, this may seem an unusual deterrent for an electronic instrument, but, given practice and locking the scale to the chromatic setting, it’s possible to play any number of melodies.

The “bending” string visual is a bit of a surprise effect. Even though the controls are being represented in familiar styles, the reality is that every button, knob, slider, switch, picture, text, etc., is just a defined region of a touch screen. As such, a slider can function as an activator (like a button) as well as have two-dimensional aftertouch, not just one axis of control. In the case of Lights, this is visually represented as a bending string. In early versions of KDKit, this effect was built in as an option for any slider. Currently, the latest version doesn’t have this option, but it is planned in a future release as a unique animation.
to stretch or bend any control.

3.2.3 Performance and Use

Lights has only been used in concert performance once, but it has been a kid-friendly favorite at various fairs and demonstrations. As a performance instrument, it works well in a setting that might suit an electric guitar (especially playing “solos”). For a practiced musician, this may not seem particularly useful, but Lights has no learning curve to start playing. Set up a backing track and the app turns into a fun jam tool for musical beginners, especially kids.

As a demonstration tool, Lights is easy to understand given the visual feedback and familiar sound. Adults and children alike have commented on how the combined effect is fun and even soothing. It has served as a great tool for drawing interest and opening discussion at various events.

3.3 Grain

Following development on In Sen and Lights, KDKit underwent its first major overhaul. Rather than just providing basic familiar controls with advanced properties, a major goal for KDKit became more advanced forms of control and more unified class structure. This expansion of the granularity of KDKit coincided with the development of Grain, a granular synthesis app.

Grain is designed to be more sonically robust than In Sen and Lights but still have unique quirks and qualities that make it sound like a recognizable instrument. The app allows you to access your music library or upload files directly to it via wireless connection. These sounds can then be loaded into playback buffers to be layered and manipulated. In this regards, it is similar to a looping app, but it doesn’t have discrete tempo or meter control.

3.3.1 Sound Design

Grain features three sound buffers with independent granular controls. It also features independent delay lines and a spectral freeze effect. You can set the granular playback
speed, pitch, window shape, and window size for each track, but you can also set the reverb amount and duration of the spectral freeze window.

A sound file is manipulated by scrubbing a visual representation of the waveform. Touching any location in the waveform will jump the file to that position and begin playing back via the spectral freeze. Then, as you move your finger the frozen position is updated in real time. Depending on your settings, the effect ranges from a noisy drone or quick stutter to short looping sections. It’s also possible to scrub through the file with varying playback speed depending on how fast you are moving your finger, similar to record scratching.

The delay lines add another layer of looping control. The pitch, yaw, and roll of the device are mapped to the feedback gain of the delay line for each track. When the device is at rest, the delay lines are set to zero gain, but moving the device allows the delay line buffers to fill with rather precise control. This can lead to complex loop layering, reverb, and distortion effects.

3.3.2 UI Design

The screen is divided into three sections, each assigned to a start/stop button and waveform visualization for one of the three buffers. Grain comes pre-loaded with default sound files and always loads the last settings, so the buffers and visualization are never empty. Tapping on the start button or waveform begins playback and turns the start button into a stop button. At this point, touching and holding the waveform will hold that
position on repeat until let go, which will resume playback. This allows you to explore the waveform and hear little slices of sound. The Y-axis controls gain which is represented as a background glow behind each waveform.

A side menu gives access to the other track settings previously mentioned, as well as access to the wireless upload interface and wireless device connectivity. In order to change the sound file for each buffer, long press on the playback button to open the menu giving access to the sounds stored in the app or your music library.

The wireless upload interface lets a user connect to the app via a desktop browser and upload media directly to the app local storage. This function is meant to be a convenient work around from having to sync your music library with iTunes. The wireless device connectivity allows a second mobile device running Grain to connect and be used as a secondary controller. The concept here was to enable more points of gestural articulation for controlling sounds from a single source [42], [46]. When in controller mode, the accelerometer and X/Y touch screen coordinates provide real-time control over the reverb time and freeze duration for the tracks playing on the main device. Since Grain was designed to be able to be played with one hand, it’s possible to use two devices as a main/controller pairing, but it’s also possible to use a main/main combination giving access to six buffers.

3.3.3 Performance and Use

Grain has been used in multiple solo pieces and a small ensemble work. The app has proven much more versatile than both In Sen and Lights. Even though each piece is characterized by the particular sounds chosen, the delay line and freeze functions give the app its own nuance. Grain can be learned in a manner similar to other instruments, and it can be identified by sound even when playing different sound files.

On phones, the touch screen use isn’t visibly enticing, but once the accelerometer is employed it becomes much more engaging to watch. Body motion makes visible connections to sound control in a way that doesn’t require knowledge of the instrument to understand that a performer is moving to manipulate the sound. With a tablet, Grain can be much
more visibly expressive if performed with that intent. Rather than twisting and turning
the device, cradling the device in one arm allows a performer to control the accelerometer
with larger body motions. The increased touch screen size also allows a user to make larger
arm gestures with a more obvious connection to the sound manipulation.

3.4 Minim

Minim is a microtonal synthesizer featuring a grid of pads, and settings for each row
of pads can be customized independent of the other rows. This allows different sounds to
be available on one screen, but it also lets a user tweak the timbre of each pitch within
a range of the same sound type. The synthesizer options are limited in a way meant to
create instrumental character but be robust enough to create a variety of different sounds.

Minim was the last KDKit app to be written entirely in Objective-C, and it would
eventually lead to the latest complete overhaul of KDKit. During development, it became
clear the library was becoming unmanageable and needed a more solid class hierarchy. This
led to KDKit being rewritten entirely in Swift and for the class structure to evolve into the
latest iteration.

Figure 3.4: Minim’s pads can be individually customized.
3.4.1 Sound Design

The synthesizer is essentially a single click passed through a series of filters and feedback delay lines. The driest settings result in hearing the short click, but the range of sounds include synthesized drums, distorted leads, and formant-shifting drones. Tapping on a pad will initiate a sound, but they also include an aftertouch feature where the X/Y axes will adjust the filter ranges and, on supported devices, pressure will manipulate gain for that particular pitch.

The app also includes four recording buffers. When activated, any sound produced by a pad will be recorded until recording is stopped. This feature is meant to help layer complex textures with repeating elements throughout a performance. While it can be used to create backing elements on persistent loop, it is designed to be a more active part of playing the app.

While each row of pads can be independently set to produce a different sound, there is also a global effects chain through which all sounds pass. These effects are fairly standard and include a global delay, reverb, distortion with delay, and bit crush. In the signal flow, the effects chain occurs after the main synth but before the recording buffers.

Frequency control in Minim is on a sliding scale. There are two tuning options: adjustable equal-tempered and natural tuning. In the adjustable equal-tempered mode, the frequency slider value adjusts the frequency distance between successive pads. For example, setting the value to 100 would mean the first pad would be 100 plus the base frequency, the second pad would be 200 plus the base frequency, and so on. The base frequency is controlled by a separate slider on the main interface.

When at the base setting, the natural tuning mode sets each pad to be a successive natural harmonic of the base frequency. Adjusting the tuning slider in this mode detunes the natural harmonics in cents. The base frequency setting functions the same in both modes.
3.4.2 UI Design

The main interface primarily consists of a grid of circular pads. The design consideration was to enable expressive gestures via prolonged touch in a similar but different manner to PitchCanvas [47]. After an initial touch, the acts of rocking the device, moving a finger in circular or side to side motions, and changing pressure all affect the sustained sound. Similar to Grain, using your full upper body in performance is potentially more engaging for the audience, but the practice could also be more musically beneficial if the interface and sound were designed specifically to encourage this body motion. While this may seem artificial to some, arguably, music teachers of all kinds make just those same suggestions to play acoustic instruments. Swaying, leaning in, head motion and so on are often suggested to increase musical expression and audience engagement. Building on lessons from Grain, Minim is designed such that the same sort of motion and interaction can have an audible effect with direct connections to visible gestural intent.

There is a menu icon which opens the pad settings, but the settings for each row of pads can be opened individually as well. This option was included so settings can be changed during a performance without interrupting playback on other pads. One of the issues with Lights and In Sen was the lack of versatility when it came to the types of sounds they could produce versus the inherent potential for musicality. With Minim, it is intended that the pad settings can change throughout a performance. Settings can also be saved and recalled with just a couple taps using a quick access menu.

In order to use space efficiently, the controls for the recording buffers have been condensed to a single button for each buffer. Tapping the button for an empty buffer will start recording and begin a rocking animation as visual feedback. Tapping it again will stop the recording and change the inner icon to show the buffer is not empty. Tapping the button for a filled buffer will start the loop and begin a pulsing animation that is timed to the loop duration. Tap again to stop the loop. In order to empty a buffer, press and hold on the button until you feel the haptic feedback click. Just below the recording buttons is a
The last slider on the main interface controls the base frequency of the pads. This has multiple uses depending on the settings and situations. The basic function changes the frequency of the lowest note and adjusts the other pads based on the other frequency settings. Using this slider and a single pad, it is possible to play melodies or create glissando effects on sustained notes. With some sounds, the slider can be used as a manual low frequency oscillator to modulate the carrier.

The accelerometer is an option that can be toggled on the effects page. When turned on, device motion manipulates a vibrato effect. Device angle controls the rate and size of the vibrato, and the device acceleration handles a crossfade that introduces the vibrato into the signal flow. Having a one-to-one angle-to-vibrato mapping would have required cumbersome and constant motion to create anything but a slow vibrato effect. Since the vibrato is controlled by an acceleration-activated oscillator, a shaking gesture is still visibly tied to a vibrato effect, and it can create more dynamic and sustained vibrato without impeding other actions or looking as if the performer is seizing.

3.4.3 Performance and Use

Minim is featured in two movements of Robot develops anxiety, disassociates. The app is clearly more gesturally expressive than both Lights and In Sen, and the potential for numerous works is more obvious. The variety of sounds and ability to quickly change the synth settings also means the recording function does not have to be leveraged in every work to be able to create density and momentum.

Even though it’s possible to open the app the first time and begin effortlessly playing, Minim requires dedicated practice to really grasp the nuances. For example, dynamics and subtle timbral shifts are relatively easy since they are mapped to common tactile mobile gestures (X/Y motion and pressure), but accurately controlling the base frequency slider is a difficult but powerful tool for expanding Minim’s vocabulary. With an understanding of the frequency spread in the equal-tempered tuning setting, it’s even possible to play mono-
phonic melodies in tune with common equal-temperament. As a microtonal instrument, it's more challenging to create musical motive since it requires an ear for the frequency language as well as practiced accuracy moving the slider.
Chapter 4  
CSPD Notation

CSPD (color, shape, pattern, density) notation is a graphic score system designed for notating pseudo-improvisatory compositions for mobile instruments. CSPD is informed by concepts and features borrowed from both western standard notation and other graphic scores. It is flexible, scalable, and adaptable to instruments of any kind. Generally speaking, CSPD does not explicitly describe pitch and rhythm as we would expect from western standard notation. Instead, CSPD focuses on describing collections of either musical gestures or physical gestures that produce the appropriate sonic characteristics of a middle-ground structure [48].

Alternative systems for scoring avant-garde and other experimental music have depended on the composer to devise and describe their own notation or graphic imagery [49]. This has proven successful in different ways, but generally the score then serves as either just a tablature or specifically a literal representation of the sounds that should occur [50]. In the former case, the system serves well when the performer is expected to take significant musical freedom in their interpretation, but it doesn’t always serve a composer with more specific intentions. In the latter case, the problem is reversed. The system is an authoritative representation of the composer’s intentions, but, given unfamiliar instrumentation or avant-garde modes of performance, the notation may be near impossible for a player to accurately decipher and perform. Both options are fine in many cases, but neither fits all potential composer-performer relationships and are often not accessible or flexible enough to be used for anything other than the piece for which they were conceived. CSPD is meant to support and encourage creative coders by addressing those issues and providing a means to sketch and communicate musical ideas for any experimental instrument with relative convenience.

1Ian Pace makes an argument that scores do not just have to tell us what to do to recreate the music, but rather they may also be viewed as what not to do. This is an important concept influencing the design of CSPD. It informs readers of the types of things that should happen, sometimes emphasizing that things should definitely happen, but it ultimately does not give explicit instructions for specific sounds.
The design of CSPD notation considers what information should be in a score as well as how that information can be incorporated in a culturally sensitive manner [51], [52], [53], [54]. Rather than being a system that empowers composers over performers, CSPD specifically acknowledges musical practice where performers are given a role as co-creator [55], [56]. Even so, it is still important that composers be able to communicate intent. Where do we draw the line? How much control is sufficient in a notation system for this purpose? In this regards, CSPD was influenced by the concepts of the musical object and organism.

4.1 The Score as a Musical Object

In his article “The Score as a Musical Object,” A. Cutler Silliman argues that scores for western music are concrete representations of a composers musical intent [57], [58]. He suggests that any interpretation or addition beyond what is clearly notated is “noise.” In this regards, he makes a clear point that aleatoric, improvisatory, or process music does not fall under this umbrella whether notated or not. The intent of his argument becomes more clear when he quotes Krenek discussing total serialization: Krenek insists that serial works are the product of a process [57]. Even though the score could technically and predictably describe sounds that would occur, he tells us that such compositional practice would be premeditated but the audible result would not.

While their presented views of serial music may not hold true, the reference makes an important distinction to clarify his position. At least for him, a score could be the ultimate musical authority, superior even to a recorded performance, as long as it is the direct translation of how the composer intends to hear their own composition. He discusses how

\[\text{This is in agreement with Patricia Carpenter who wrote “The Musical Object” in 1967. In one example, Carpenter makes a clear distinction between Varese’s “Ionization” as a piece and Cage’s “Variations for Orchestra and Dance” as a process. For her, a musical object is a representation of art. Form and function must exist and neither can be fully comprehended without the entirety of the work. Some of her contemporaries were quick to rebut this position, but the idea that the intricacies of temporally bound art require a complete form to be fully appreciated does not originate with Carpenter.}\]

\[\text{While total serialization may be conceived from a process, there are certainly serial works that are composed and structured on par with similarly intricate tonal works. Krenek’s statement is included here to give more context to Silliman’s argument.}\]
the subjectivity of how an audience actually experiences a performance is of no consequence.

Within the strict parameters he provides, this can seem to be a compelling and noble ideal, but as soon as we begin considering the vast world of music beyond western tonal common practice, as Silliman points out, the argument loses steam. Given their position on serial music, it’s even questionable whether some tonal contrapuntal practices would pass the test. If we were to embrace this view, though, what is the purpose of a score? Silliman acknowledges the idea that a musical score is simply a set of performance instructions, but he seems to be suggesting that the ultimate purpose of a score is to be the music. The assumption is that a score should be wholly communicative of the complete, exact musical experience. He points out that other forms of composition that can not be adequately notated in this manner can only exist as a performance, but he also recognizes that even western standard notation relies on imprecise tools, such as dynamics and rubato, and these should simply be understood within the bounds of whatever was common practice at the time of the composition. If this is the case, why is it imperative to exclude other non-precise forms of notation?

Consider another quote Silliman references, this one by Langer: “The commanding form of a piece of music contains its basic rhythm, which is at once the source of its organic unity and its total feeling [57].” This suggests the score isn’t the ultimate authority simply because it painstakingly describes every sound, but rather because it embodies and communicates the musical structure, relationships, and intent as determined by the composer. Silliman further elaborates that a score is a musical object because, given an understanding of the composers era and culture, it allows for a reader to reconstruct the musical message. An educated listener does not require an actual performance to “hear” and appreciate the art.

For our immediate purposes, it’s not important whether or not a score is philosophically the embodiment of musical art, but Silliman’s argument for that possibility is of interest.

---

4Silliman viewed scores as an important artifact but relied on the assumption that a reader would have a contextual understanding for the composer as well as the work.
For a reader with proper knowledge, a score can contain musical information that may not be apparent in any given performance, and the potential to study and discover such compositional intent could be as artistically important—perhaps more so if we are to believe Silliman—as listening to a performer’s interpretation.

4.2 Free Composition

Musical coherence can be achieved only through the fundamental structure in the background and its transformations in the middleground and foreground.

— Heinrich Schenker
Der freie satz

Heinrich Schenker had a fairly strict view of what musical works could be considered art. His writings focused on common practice tonality, but, if we take a little liberty with the context, perhaps we can see why his theories have heavily influenced the past century: his generalizations and arguments for *organicism* are relatable to artistic worlds beyond his own intentions. While his universally applied ursatz may be a little too reductionist for some, the underlying reasoning for its existence is important.

With the ursatz, Schenker tells us that stylistically different musical works spanning a couple centuries of development were fundamentally bound by a similar abbreviated structure. Even though melodic and harmonic thematic material in the foreground could vary greatly, the ursatz suggests that all true tonal art works had a recognizable temporal form governed by the unfolding of the tonic chord. For Schenker, this is a necessary component to support his primary theory. Having intimate structural development connecting that background and the surface musical features in a bidirectional, hierarchical relationship is what Schenker calls *organicism*.\(^5\)

At its core, even when taken out of a tonal context, his is a theory that art is dependent on this organic coherence. The idea that there should be an identifiable, traceable, and audible developmental path that permeates multiple levels of articulation and connects the

\(^5\)Schenker is clear that he specifically means structural tones and bass arpeggiation in the background, strict counterpoint and voice leading in the middleground, and ultimately the appearance of surface features, the result of free composition, in the foreground. Schenker was an educator, and his entire theory is based on the belief that composition should be taught on this basis: thoroughbass as per Fux/Schenker, counterpoint as per J.S. and C.P.E. Bach, and free composition as per Schenker.
governing or boundary structural concept with surface musical features. Again, Schenker’s apparent beliefs on the limits of “true art” may be questionable, but the evidence and methods he presents to support his position do not necessarily suffer the context he originally intended. In conjunction with another of his beliefs, his observations may be found quite relevant to contemporary musical practice.

Schenker’s analytical methods and theories were rooted in the concept that understanding and finding musical art requires in-depth study. Der freie satz specifically tells us we should even study the autographs, sketches, fantasies, and other improvisatory work from the master composers [59]. He suggests there is great value in understanding the evolution of a composition and that these records of improvisatory musings can be windows into the minds of geniuses. Schenker tells us that, even in their working state, these composers had such command over their tools that the necessary traits for musical art emerged, even over extended periods of improvisation, a phenomena he called “aural flight.” For Schenker, the ursatz and organicism are both products of free composition governed by harmony and counterpoint, not the result of a formula or recipe and not the starting point for composition.

Consequently, one can understand that the layman is unable to hear such coherence in music; but this unfortunate situation obtains at higher levels, among musicians of talent. Even they have not yet learned to hear true coherent relationships... Technology enables distant parts of the world to be connected at a rate of speed which is approaching the point of frenzy. This has also conditioned our attitude to art. Today one flies over the work of art in the same manner... This contradicts not only the historical bases of the work of art but also—more significantly—its coherence, its inner relationships, which demand to be ‘traversed’ [59].

Two important things to note: Schenker recognized that experimentation and improvisation had value in the right hands, and he insisted that even the greatest musical art can not be fully appreciated without time-invested study. These observations are strikingly relatable to contemporary practice [60], [61]. The value we place on improvisation is different from Schenker [55], [62], but, aside from increased emphasis on improvisation as
a performance art, perhaps it’s because what Schenker would consider steps in a composition’s evolution are now often presented as mature works. Likewise, our fascination with novelty means we often take little time to really explore any particular new piece.

Regardless of whether we agree all tonal works share a fundamental structure, Schenker’s discussions on organicism share much with Carpenter’s and Silliman’s description of a musical object. Both are concepts describing intricate structural relationships that are inferred to belong to musical art, but both must be constructed and also discovered via a process. It’s these processes that are of concern. Both theories rely on a rich musical context and history to support their analyses, and neither would be possible without sufficient record. For most of us, we must study and learn to truly hear the music before we can even begin to make a strictly aural analysis, and the required effort compounds as unfamiliar sounds and organizational techniques are introduced. Likewise, a purely visual representation of the music will not suffice without experience hearing the sounds [63]. If cultured listening requires an investment of time and thoughtfulness in both hearing and reading, then we must have some manifestation of the composer’s intent for detailed study. In this regards, it is undeniable that a written score aids in the process.

4.3 CSPD Notation Specifications

CSPD notation is meant to encourage composers to experiment with new media by providing a means to document and build on a collective and collaborative record of work. Its design acknowledges that new instruments require experimentation and that the culture surrounding the modern avant-garde often emphasizes improvisation and exploration as a mode of performance rather than as a strictly compositional tool [48]. With some works, the art is truly in the process, but that doesn’t mean every aleatoric or improvisatory piece lacks the traits to be a musical object [64]. CSPD is meant to be musically communicable to knowledgeable musicians, even if they have never encountered the system, such that the evolution of a composition can be tracked and studied and perhaps give us the means to better analyze, critique, and improve on new work.
4.3.1 General Description

With western standard notation, educated readers can look at a series of dots and other shapes and ascertain musical information on many different levels. On the surface, standard notation describes pitch and rhythm first and foremost, but an educated reader can view the contour and order to discover phrases and formal structure. They look for patterns and deviations, and they use knowledge and experience to decipher grey areas or moments when artistic license is necessary or preferred. CSPD notation emulates these experiences also using dots and other shapes but without a strict emphasis on pitch and rhythm.

An underlying theory governing CSPD notation’s design is that background and middle levels of musical meaning are not entirely instrument specific. For example, while we may have to take some technical liberties playing a string quartet piece on a piano, it can work and be recognized because the musical meaning is affected but not entirely defined by the technique of the stringed instruments. On the surface and aside from standardized pitch and rhythm, we expect playing technique, cultural expectations, and common practice to shape the sound of the music. These things can be greatly affected by which instrument we are playing, but deeper levels of musical relationships are affected less.

CSPD is designed with that concept as a starting point. In essence, CSPD is an instrument agnostic gestural performance tablature that describes a piece just below the surface. Rather than attempting to represent specific sounds—something which may be impossible to standardize for experimental music—CSPD features collections of dots describing the gestures that make up various musical collections or sections [64]. Gesture may be physical action or musical moment since the meaning of dot shape and color must be defined per instrument or piece. It is equally appropriate to define “green dot” as middle C, in the key of C, explore the noise button, wave the instrument slowly, quickly raise and lower the

---

6Edwin Roxburg identifies this sort of indeterminacy in the scores of Witold Lutoslawski and Giorgi Ligeti. Roxburg’s provided examples show clear instructions on what to do, but the exact nature of the sound could depend on a variety of performance factors.
reverb, or quack like a duck.

### 4.3.2 Dots and Staff

CSPD dots have various shapes and colors and are arranged into stacked, horizontal rows. There can be any number and types of shapes and colors but each should be distinct enough to quickly identify when placed in close proximity. The number of rows may also vary as necessary, but there must be sufficient space to visualize macro shapes and patterns. These horizontal collections are treated in the same manner as the western standard notation staff.

![Example of CSPD notation dots on a staff.](image)

When creating a score, CSPD dot colors and shapes should be independently and clearly defined. It may not always be necessary to have both multiple shapes and colors. Then, the dots should be allocated shape and color and arranged on the staff in a manner that creates clear, identifiable collections. The contour, pattern, and density of both colors and shapes, not excluding but not limited to the dots as a whole, should correlate with middle ground musical relationships such as theme, variation, phrase trajectory (voice leading), register, linear ascent/descent, etc.

In the following example, notice the growth of the red shape and density at the beginning. Within that section, the blue dots create a rising contour that could be understood as a leap then step. Between the red beginning and green middle is something akin to *subito piano*, but also notice the patterned shape the red and blue dots make in the middle. In the final section, the red and blue dots might be viewed as variations of the same gesture.\(^7\)

\(^7\)Red low, then blue high. Red high and blue low together. Red and blue high. Red and blue low, but a little different. Red high and blue low again.
The opening is a section of red and blue with individual shape and contour. The middle features green, but red and blue are still present and patterned. The ending is black, but the interplay between red and blue continues to develop beyond that of the middle section. Musically speaking, we might expect this section to lead into a further developed red and blue section, perhaps with an interplay between green and black.

![Figure 4.2: CSPD dots should be arranged to show relationships.](image)

### 4.3.3 Rhythm

Surface level rhythm is not explicitly notated in any specific manner. When an exact rhythm is desired, standard notation symbols can be used to define the rhythm as a shape or color, or they could be placed as a performance note above or below the staff.

![Figure 4.3: When necessary, specific rhythms can be notated in the legend or next to the CSPD staff. In this example, the notated rhythm should be featured prominently throughout the section.](image)

Overall timing and formal structural boundaries are dependent on the clear vertical alignment of sections. Since there is no need to depict meter and measures on a staff, vertical lines are used to mark sections which may include a precise or approximate textual time or duration (e.g. “about 30 seconds”, “four times staggered then arrive together”).

---

8 Since this notation system describes music beneath the surface, it may be advisable to avoid terribly specific rhythms unless they are thematic or otherwise important to understanding the piece. One possible rhythmic solution could be to describe a color as a rhythmic characteristic (e.g. “variations of long-long-short”, or “occasional dotted rhythms”).

65
or “2min 20sec”). For ensemble scores, it is important to ensure moments of arrival are clearly marked, and performers should be instructed to use cues, if not a conductor, to synchronize events. Repetition and coda can be notated as standard notation. Unless otherwise notated, horizontal timing and duration in CSPD are non-specific and open to performer interpretation.

Figure 4.4: This is an excerpt from *Cello Pie*, a piece for three performers on Grain. While left to right represents the passage of time, duration is non-specific in CSPD. Vertical lines can be used to denote landmarks or points of arrival. In some cases, it may be appropriate to mark a target time above a vertical line.

### 4.3.4 Dynamics, Tempo, Articulation, and Other Marks

Dynamics, tempo, articulation, and other performance or technical marks are handled as standard notation. Text or symbols are both acceptable and should be placed above or below the relevant area of the staff as expected.

For some instruments, it may be necessary to describe events that are not covered by standard notation, such as changing the settings for an electronic instrument or describing an action or articulation that is non-typical. If it doesn’t seem appropriate to define a shape or color to represent these events, we can borrow from the myriad of other examples
where non-standard notation marks and text are incorporated into contemporary scores for those same purposes.

### 4.3.5 Pitch

Generally speaking, CSPD should handle pitch and frequency by describing the characteristics of a section. For example, the definitions for color and shape along with the contour, pattern, and density could correlate to variations of a D major arpeggio or frequencies between 200 and 500 hertz in intervals of 25.

Even though CSPD is not intended to describe specific pitches or frequencies on a note by note (or event by event) basis, that doesn’t mean the information can’t be incorporated. If it’s necessary to have precisely notated pitch (or rhythm) for long periods or an entire piece (e.g. an equal-tempered experimental instrument or one in ensemble with traditional instruments), then a CSPD staff can be incorporated alongside a standard staff and the performer instructed to respect meter. If non-equal-tempered frequencies are required, they should be described as colors, shapes, or performance marks next to the staff.

### 4.3.6 Scaling Information

CSPD is meant to be flexible. There are few hard rules or specifications, and the primary decision driver should be to clearly articulate musical events and relationships. As a proposed reusable solution, the specifications described in this project are intended to help musicians communicate in a manner where experience with standard notation and analysis can be leveraged but removed from components that restrict instrument capabilities. This means the general description provided here serves only as guidelines for creating scores similar enough to build familiarity. Its acceptable and appropriate to extend the system in creative ways as needed [65].

One example of how to scale the system could be to add a secondary dot to the middle of the other dots. Perhaps a small black dot found on a purple triangle could mean to articulate a gesture differently. We may go as far as to have a black and white secondary dot mean something different, or to have multiple secondary dots on a single primary dot.
denote surface level repetitions of an event.

Another example could be to use a successive color gradient to show timbral shifts (e.g. light green to dark green represents bright to dark timbre on a scale). Perhaps a bounding shape could be used to draw visual attention to important collections similar to segmentation analysis. Any legible solution is possible as long as it is well defined in the score instructions or legend.
Chapter 5  
Composition

5.1 Robot develops anxiety, dissociates (27’)

*Robot develops anxiety, dissociates* is a solo work in four movements for mobile devices. Two movements are performed with Grain, and two movements are performed with Minim. There is an over-arching theme composed around beating frequencies and polyrhythms. This piece demonstrates instruments created with KDKit and scored with CSPD notation, and it was first performed in recital on February 6, 2019, in the Digital Media Center Theater at Louisiana State University.

Beating frequencies are used to represent conflict and tension, but also uncertainty. There is a contrast between the themes of the first and fourth movements (conflict) and the second and third movements (resolution), but the meta-theme (tension, uncertainty) is developed throughout. The piece is pseudo-improvisatory, but the CSPD score guides form and collections that create motive as well as beating effects. Still, it is up to a performer to embody the character of the work in their interpretation.

5.1.1 Introducing the Consumer (6’)

The first movement, “Introducing the Consumer,” is minimal in nature but perhaps not in effect. The primary musical building block is overlapping ostinati of straight rhythms. With Minim, the dynamic of the repeating pattern can be controlled with pressure, and a performer should be creating polyrhythmic patterns (by accenting beats) as well as general dynamic swelling.

“Introducing the Consumer” relies on the performer to create tension. The score guides the characteristics of each section, but there is no indication of specific combinations to create proper effects. Texture and motion are created by building chords and moving voices, but tension and release should be created with beating frequencies, consonance/dissonance, and polyrhythm.

This movement is meant to feel fast-paced as extra voices arise from the rhythmic
accents, but changes to actual voiced notes should be slow. The robot is thoughtful but conflicted as it first develops consciousness, and it lacks control or direction. Voices should wander, and, while rhythms and frequency beating should resolve, there should be little or no harmonic resolution.

5.1.2 hold on, asphyxiate (8’)

“hold on, asphyxiate” represents deep, conflicted feelings. Our robot is becoming more aware and is clearly unhappy with its station in the world.

The piece is composed of five 8-bit video game samples (think Mario picking up a coin) and one clip of factory-type industrial sounds, all processed with Grain on two mobile devices. One device should be left stationary on a table while the other is held to be used with either one or two hands as needed.

For this piece, the loops are often allowed to run on their own while the performer explores dynamic shapes amongst them. Sonically, it is composed to express inner turmoil rather than direct conflict, but it is also composed to maximise the physical musicality of the actual gestures. There’s a lot of space in the sound for a performer to make repeated gestures that are obviously connected to audibly identifiable events, as well as grand gestures at moments of climax.

The score is the only one in this work to use individual dot shapes as well as color. The colors roughly represent each sound file, but they are named to represent the character of the sound since there is potential for overlap between sources. The shapes represent dynamic intent or direction.

The provided settings will leave opportunities for harmonic resolution. This is meant to be a key tool to create phrase closure. Much of the texture revolves around incessant repeating sounds that are hidden as other sounds build, but the background is then revealed to have never left as the texture dissolves. This “pedal” revelation should often coincide with some semblance of harmonic resolution.
5.1.3 Berceuse (8’)

The third movement features three pre-composed melodies on two devices with Grain. One of the melodies is on four tracks with different settings. For this piece, a performer should hold a device in each hand, but it does not use Grain’s WiFi controller feature. Both devices should be on the main player setting.

“Berceuse” is a pseudo-release for the previous two movements. The characteristics of the composed melodies are meant to be somewhat soothing, but overall the piece should still prominently feature unresolved tension in a subversive way.

Similar to the second movement, the performer should let some loops run while exploring moments of other sounds. Rather than creating sweeping gestures like in “hold on, asphyxiate,” Grain’s spectral freeze is used to capture jittering moments of a sound, and the device motion is used to build those moments in the various delay lines for feedback.

The melodies are composed to overlap and run with consonant effect, so it’s up to the performer to use the tools at their disposal to create tensive dissonance with feedback. This gives opportunity for physical expression to clearly map to the tension. The most conflict will be created with maximum feedback in the delay lines. In turn, that effect will coincide with exaggerated twists as the arm reaches for the maximum values on multiple axes of motion.

Subversive resolutions are created when tension is released as the delay lines are emptied and the consonance comes back to the forefront. The nature of the spectral freeze lends to hearing it as a new voice though, not as a timbral shift of the consonant sounds. As such, the “release” of the delay lines does not always sound like a resolution of the tension, especially when the release is very slow.

5.1.4 Inner Monologue (5’)

The fourth movement should be characterized as declarative. The settings on Minim allow for a short, nearly western equal-tempered descending motive comprised of three notes. The piece is composed around this motive, which should be used as a theme throughout.
Counter motive and variation can be created with pitch as well as contour.

“Inner Monologue” requires the use of Minim’s loops as well as multiple presets. For most of the piece, each section switches presets and builds on the previous. The score guides the textural density and registral contour, but, aside from the primary motive, it’s up to a performer to create related phrases. Out of all four movements, this is probably the most challenging concerning musicality. The fast moving nature doesn’t lend well to Minim’s aftertouch feature, so it’s important to ensure phrase gestures are practiced. Visible musical intent will likely rely on accurate technical ability rather than reactive dynamic interaction.

Even though the construction of the piece is fairly straightforward, this movement should mirror the tension of the first movement. It represents declarative statements made by the robot, but it shouldn’t be a convincing monologue. It should leave unresolved questions even if counter motive seems to create answers to the theme.
Chapter 6
Conclusion

*Robot develops anxiety, dissociates* is meant to be a collaboration between the composer and the performer. There are characteristics and some rules, but much of the musical realization is up to the performer. Given the nature of these electronic instruments, the musical success of a performance relies heavily on visible gestural intent [44]. A performer must compose musical interactions in real-time, and be able to perform them in a manner that the audience understands. As a purely auditory experience, the important role of the performer may be lost due to the electronic sounds [66].

The tools created for this project are meant to help musical creatives explore and address such problems. KDKit is designed to help creative coders break into mobile development, and CSPD notation is meant to temper the edge of uncertainty concerning how to relay non-traditional musical ideas as compositional intent. Together, perhaps they can open doors for curious and inventive minds.

In Sen, Lights, Grain, Minim, and *Robot develops anxiety, dissociates* demonstrate that these tools can be used to some effect. They exemplify one approach to musical experiences with mobile devices, but there are any number of other possibilities. Here we have mobile apps developed as musical instruments and used in a traditional recital format. Other possibilities that have been explored are games, social experiences, travel/location, interactive installations, and multi-media visual art. There are certainly other modes and media to discover and experience. Perhaps the tools featured in this project can be used directly or serve as examples for other different or better tools. Ultimately though, the goal of this project is to be a stepping stone towards new, productive creativity, and, even though it has been a mostly personal experience, the development process itself has been an example of creative evolution.
References


Appendix
Robot develops anxiety, dissociates (Score)
Introducing the Consumer

solo for Minim

Minim

\[ \begin{align*}
&| \text{cresc.} \quad 1' \\
&| \text{mp} \quad \text{mp} \quad \text{mf} \quad \text{mf} \quad \text{mp} \\
&| 5' \\
&\text{quick bursts, sparse, isolated} \\
&\text{fast beating, chords} \\
&\text{steady, straight but with accents} \\
&\text{low and contrasting, slow dynamic pulse} \\
&\text{registral polyrhythms} \\
&\text{stuttered, assymetrical accents}
\end{align*} \]

- The distance above and below the center line approximates register.
- Vertical dotted lines mark general sections.
- Estimated points of arrival are marked for 1 and 5 minutes.
- Short, solid vertical lines represent rests. Multiple consecutive rests simply mean take a longer rest.

80
hold on, asphyxiate

solo for Grain on two devices

~8'

- Horizontal lines represent sustain. Let the sound continue looping with no input activity.
- Dotted vertical lines represent a clear break or rest. Sustained sounds carry through the rest.
- Vertical position within the staff has no explicit meaning, but a change in position represent some change in the characteristic of the gesture/sound.
- Contour resulting from vertical changes represents similar characteristics for symbols in similar positions.
- Colors correspond with each Grain track on two devices. The color key describes the sound effect, but each sound file is described below and recommended settings are provided. It's not important to have the exact sound files for this piece, but any substitutions should have similar characteristics.

- *industrial.wav* features metallic factory sounds
- *impact1.wav* and *impact2.wav* are 8-bit sound effects for character collision.
- *coin1.wav, coin2.wav, and coin3.wav* are 8-bit sound effects for collecting a coin.
Berceuse

solo for Grain on two devices

III

- This piece is for three precomposed melodies. The colors correspond with the matching track in the color key. `lullaby1.wav` is used four times, but with different settings for each track.

- The density of a given color represents the prominence of the delay line for that track.

- The vertical position of a given color represents the prominence of that voice.

- Once introduced, even if a color isn't present it should be left to loop with no interaction and low presence.

- Timing is very free. Search for overlapping contour and shape, and treat each with care.

1 - lullaby1.wav
2 - lullaby2.wav
3 - lullaby1.wav
4 - lullaby1.wav
5 - lullaby3.wav
6 - lullaby1.wav
IV
Inner Monologue

*solo for Minim* ~5’

- This movement is a development of the short motive shown above. It’s featured prominently with counter motive and variation in all voices. Counter motive and variation should include microtonal elements.

- Vertical position represents approximate register.

- Bracketed sections are recorded in the corresponding loop buffer.

- The numbers below the staff represent when the associated loop should start and stop. A number with an arrow means to start the loop, a number with an X means to stop the loop.

This piece uses four of Minim’s default presets. Each color matches a preset name. For Crystal - *detune*, use the Crystal preset but slightly detune using the frequency slider.

- Crystal - main theme
- DevBass
- Beep Boop
- Crystal - detune
Vita

Danny is a composer, sound artist, researcher, software developer, and music educator. He held a private guitar studio for several years before teaching music technology and music theory at Southern University in Baton Rouge, LA. He has also released several mobile apps to the Apple App Store.

Danny received a BM in Guitar Performance from Southeastern University and MM in Electronic Music Composition from the University of Alabama. He has presented work at various conferences including New Instruments for Musical Expression (NIME), Society of Electro-Acoustic Music in the United States (SEAMUS), International Computer Music Conference/Sound and Music Computing (ICMC/SMC), and the Texas Music Educator’s Association (TMEA). This dissertation is in pursuit of a PhD in Experimental Music and Digital Media from Louisiana State University.

Danny’s major research areas include mobile music, gesture and performance, and music technology in education.