T.C.
ANKARA MÜZİK VE GÜZEL SANATLAR ÜNİVERSİTESİ
MÜZİK VE GÜZEL SANATLAR ENSTİTÜSÜ

SONIC-PI YAZILIMI İLE SES EKLENTİLERİNİN ENTEGRASYONU

YÜKSEK LİSANS TEZİ

Doğuhan Eren KARACAN

Müzik Bilimleri Yüksek Lisans Programı

Tez Danışmanı: Doç. Dr. Abdurrahman TARİKCİ

Ağustos 2020

T.C.
ANKARA MÜZİK VE GÜZEL SANATLAR ÜNİVERSİTESİ
MÜZİK VE GÜZEL SANATLAR ENSTİTÜSÜ

SONIC-PI YAZILIMI İLE SES EKLENTİLERİNİN ENTEGRASYONU

YÜKSEK LİSANS TEZİ

Doğuhan Eren KARACAN
1802011

Müzik Bilimleri Yüksek Lisans Programı

Tez Danışmanı: Doç. Dr. Abdurrahman TARİKCİ

Ağustos 2020

ETİK BEYAN

Ankara Müzik ve Güzel Sanatlar Üniversitesi Müzik ve Güzel Sanatlar Enstitüsü Lisansüstü Tez Yazım Kılavuzunda belirtilen hususlar doğrultusunda hazırladığım bu tez çalışmasının akademik ve etik kurallar çerçevesinde gerçekleştirilmiş olduğunu; tezde yer verdiğim tüm bilgi, belge, bulgu, yorum ve sonuçları bilimsel etik ilkelere bağlı kalarak oluşturduğumu; yararlandığım eserlerin tümüne etik kurallar dâhilinde atıfta bulunup kaynakçada yer verdiğimi ve Enstitüye teslim ettiğim çalışmanın bütünüyle özgün nitelikte olduğunu bildirir; tez çalışmamla ilgili olarak burada dile getirdiğim kişisel bildirimin aksi yönünde ortaya çıkabilecek ve etik kuralların ihlal edilmiş olduğuna işaret eden bir tespit durumunda, akademik kariyer bağlamında aleyhime doğabilecek tüm hak kayıplarını peşinen kabullendiğimi beyan ederim.

31.08.2020 Doğuhan Eren KARACAN

ÖNSÖZ ve TEŞEKKÜR

Elektronik müzik, tasarım aşamasında oldukça geniş olanaklar sunsa da performans aşamasında birtakım sorunlar barındırmaktadır. Bu sorunlar, elektronik müzik performansının gerçek potansiyeline erişmesini engellemektedir. Bu çalışma, elektronik müzik performansında önemli bir araç olma ihtimalini taşıyan canlı kodlama pratiğinin geliştirilmesi ve yaygınlaştırılması amacıyla gerçekleştirilmiştir.

Bu çalışma konusunun seçiminde ve yazımında yapmış olduğu katkılardan dolay tez danışmanım sayın Doç. Dr. Abdurrahman TARİKCİ'ye, bu alana dair merakımı ve hevesimi güçlendiren sayın Doç. Dr. Mustafa Kemal KARAOSMANOĞLU'na, tezin yazım sürecinde desteklerini esirgemeyen Sina ŞEHLAVER'e, Serkan ÇOLAK'a ve aileme teşekkürlerimi sunarım.

Ağustos 2020 Doğuhan Eren KARACAN

İÇİNDEKİLER

Sayfa

KISALTMALAR

ENIAC **:** Electronic Numerical Integrator and Computer

CSIRAC **:** Commonwealth Scientific and Industrial Research Automatic Computer

TOPLAP **:** Temporary Organisation for the Promotion of Live Algorithm Programming

MIDI **:** Music Instrument Digital Interface

STEIM **:** Studio for Electro Instrumental Music

KIM **:** Keyboard Input Monitor

DSL **:** Domain Specific Language

IDE **:** Integrated Development Environment

FM **:** Frequency Modulation

PCM **:** Pulse Code Modulation

DAW **:** Digital Audio Workstation

API : Application Programming Interface

UDP : User Datagram Protocol

IPC : Interprocess Communication

OSC : Open Sound Control

GUI : Graphical User Interface

CLI : Command Line Interface

GPL : General Public License

ŞEKİL LİSTESİ

Sayfa

SONIC-PI YAZILIMI İLE SES EKLENTİLERİNİN ENTEGRASYONU

ÖZET

Bilgisayar müziği, yirminci yüzyılın ikinci yarısında ortaya çıkmış bir elektronik müzik türüdür. Bilgisayar müziğinde ses bilgisayar hesaplamalarıyla dijital olarak üretilmektedir. Bu sebeple bilgisayar müziğinin gelişimi bilgisayar teknolojilerinin gelişimi ile paralellik göstermiş ve dolayısıyla 1990'lara kadar girilen kodlar eşzamanlı olarak hesaplanıp seslendirilememiştir. 2000'lere gelindiğinde ise bilgisayar müziği çoğu kişisel bilgisayar tarafından gerçek zamanlı olarak yapılabilir hale gelmiştir ve bilgisayar müziği alanında canlı performansın yolu açılmıştır. Ancak bu canlı performans imkânı, elektronik müziğin canlı performansına ilişkin belli zorluklar ile birlikte kazanılmıştır.

2000'li yıllarda yaygınlaşan bir bilgisayar müziği dalı haline gelen ve ses sentezinde kullanılan algoritmaların performans esnasında yazılması ve değiştirilmesi prensibine dayanan canlı kodlamanın (*live coding*) bu sorunların biçoğunu çözüme kavuşturması mümkündür. Ancak bu çözümün elektronik müzik icracıları tarafından kullanılabilmesi için öncelikle Sonic-Pi gibi yazılımlar vasıtasıyla canlı kodlamanın sadece bilgisayar müzisyenleri değil herkes tarafından kolaylıkla öğrenilebilecek ve icra edilebilecek hale getirilmesi gerekmektedir.

Sonic-Pi, aslen ilkokul öğrencilerine kod öğretmek amacıyla geliştirilmiş bir canlı kodlama yazılımıdır. Programın özgün geliştirilme amacı ve bu amaca yönelik tasarlanmış içeriği, aranan kolay öğrenilebilme şartını yerine getirmesini sağlamaktadır. Ancak yalnızca Sonic-Pi ile elektronik müzikte çok sık rastlanan ses eklentilerini (*plug-in*) kullanmak mümkün değildir. Bu imkânsızlık, Sonic-Pi'nin bir elektronik müzik aracı olarak işlevini ve cazibesini sınırlamaktadır.

Çalışmanın ana fikri, canlı kodlamanın elektronik müzikteki canlı performans sorununa bir çözüm yolu teşkil edebileceği hipotezine dayanmaktadır. Çalışma ile canlı kodlamanın elektronik müzik performansında daha sık kullanılan bir araç haline gelmesi amaçlanmaktadır. Bu sebeple, canlı kodlamanın elektronik müzisyenlere daha cazip hale getirilmesi amacıyla öğrenilmesi kolay Sonic-Pi yazılımına elektronik müzikte çok sık kullanılan ses eklentilerini destekleme özelliği eklenmiş, Sonic-Pi'nin bir elektronik müzik aracı olarak işlevselliği artırılmışır.

INTEGRATION OF SONIC-PI SOFTWARE WITH AUDIO PLUG-INS

SUMMARY

Computer music is a genre of electronic music that emerged in the second half of the twentieth century. In this genre, sound is produced digitally by computer calculations. For this reason, the development of computer music has been in line with the development of computer technologies and the code that was entered could not be calculated and converted to sound simultaneously until the 1990's. By the 2000's, most personal computers could create real-time computer music and thus, live performances in the genre of computer music became a possibility. However, this possibility came with with certain difficulties related to live performance of electronic music.

It is possible for live coding, which has become a widespread branch of computer music in the 2000s and is based on the principle of writing and changing the algorithms used in sound synthesis mid-performance, to solve many of these problems. However, for such a solution to be implemented by electronic music performers, first of all, live coding should be made easy to learn and perform by not only computer musicians but everyone, via software such as Sonic-Pi.

Sonic-Pi is a live coding software originally developed to teach young students programming. The unique development purpose and the content designed for this end ensure that the program meets the requirement of high learnability. However, it is currently not possible to use plug-ins, which are very common in electronic music, just with Sonic-Pi. This impossibility limits Sonic-Pi's functionality and appeal as a tool for electronic music.

The subject matter of the study is based on the hypothesis that live coding can be a solution to the problem of live performance in electronic music. With this study, it is aimed to make live coding a more frequently used tool in electronic music performance. For this reason, in order to make live coding more attractive to electronic musicians, Sonic-Pi software has been expanded to support audio plug-ins widely used in electronic music, thus the functionality of Sonic-Pi as an electronic music tool is enhanced.

1 GİRİŞ

2000'lerin başında ortaya çıkan canlı kodlama, elektronik müzik performansının mevcut sorunlarına önemli bir çözüm yolu teşkil edebilecek bir performans pratiğidir. Ancak bunun için öncelikle canlı kodlamanın birçok elektronik müzisyen tarafından öğrenilmesi ve kullanılması gerekmektedir. Bu amaçla, öncelikle canlı kodlamanın halihazırdaki az bilinirliğinin sebeplerini araştırmak ve bu sebepler üzerine çalışmak faydalı olacaktır. Bu doğrultuda canlı kodlamanın elektronik müzik içerisindeki mevcut yerinin ve öneminin anlaşılması için tarihsel gelişimini incelemek oldukça önemlidir. Ayrıca bir performans pratiği olarak canlı kodlamanın getirilerinin tam olarak saptanabilmesi için elektronik müzik performansının mevcut durumuna değinilmesi gerekmektedir. Böylece canlı kodlamanın bir elektronik müzik aracı olarak yaygınlığının ve kullanılabilirliğinin artırılması için gerekenler daha açık olarak tespit edilebilecektir.

Bilgisayarlardan ses çıktısı alma işlemi 1947 yılında ilk elektronik bilgisayar olarak geliştirilen ENIAC (*Electronic Numerical Integrator and Computer*) ile başlamıştır . Ancak bu dönemde alınan ses çıktısı, sanatsal olmaktan çok işlevseldir ve sistemin bir döngüde takılıp takılmadığını anlamak amacıyla kullanılmıştır. Dijital bilgisayarlardan müzikal sesler üretilmesi amacıyla yazılan ilk programlardan biri ve en çok ses getireni, 1957 yılında Bell Laboratuvarları'nda çalışan Max Matthews'ın yazdığı MUSIC isimli yazılımdır. 1960'lı yıllarda beş farklı sürüm ile güncellenip geliştirilen bu yazılım, dönemin en güçlü bilgisayarları üzerinde çalıştırılmasına rağmen basit seslerden meydana gelen bir dakikalık bir eseri hesaplayabilmesi yaklaşık bir saat sürmektedir. Ancak buna rağmen, ortaya çıkan bu yeni müzik türünün potansiyelinden dolayı; James Tenny, Jean Claude Risset ve Charles Dodge gibi besteciler, laboratuvara sıklıkla uğrayarak yazılım üzerinde ciddi eserler ortaya koymuşlardır ("Short History of Computer Music", t.y.).

Bahsedilen bestecilerin çalşmalarının yanısıra Max Matthews'ın Science dergisinde yayınlanan "The Digital Computer as a Musical Instrument" isimli makalesi, MUSIC programının ve dolayısıyla bilgisayar müziği (*computer music*) olarak isimlendirilen

bu yeni müzik türünün popülerleşmesini ve yayılmasını sağlamıştır (1963). Laboratuvara sıklıkla uğrayıp MUSIC programı ile çalışan yeni mezunların büyük bir kısmının da farklı üniversitelerde akademik pozisyonlara gelmesi ile bilgisayar

müziği 1970 yılında birçok üniversitede araştırılan bir müzik türü ve müzik üretim yöntemi haline gelmiştir (Manning, 2004).

Bütün bu gelişmelere karşın, bilgisayarlar bu dönemde ticari müzik endüstrisi içerisinde varlık gösterememiş, dolayısıyla yaygınlaşamamıştır. Manning (2004), bu durumun iki önemli sebebi olarak bu programların çalıştırılabileceği bilgisayarların kişisel olarak satın alınamayacak derecede pahalı olması ve programların da karmaşıklaşmasıyla birkaç dakikalık bir müzik eserinin saatler süren hesaplama süreçleri sonucunda elde edilebilmesinden bahsetmektedir. Ayrıca bilgisayarlardan detaylı programlamalar sonucu elde edilen sesler bu dönemde hızlı bir gelişme yaşayan analog sentezleme teknolojileri ile nispeten kolayca elde edilen seslere göre ve bu seslerin elde edilme hızına göre daha az cazip bulunmaktadır. Dolayısıyla, alanda yapılan çalışmalar dijital olarak kontrol edilen analog sentezleyiciler üzerine yoğunlaşmıştır. Analog sentezleme üzerine yapılan çalışmalar alandaki ve müzik endüstrisindeki hızlı ve hatta gerçek zamanlı sonuç ihtiyacını karşılamıştır ancak elde edilebilecek ses çeşitliliği bakımından bilgisayar temelli ses sentezlemenin sunduğu olanakları sunamamıştır. Zira doğru algoritmalar ve yeterli hesaplama süresi ile çalışıldığında bilgisayar temelli sentezleme çok geniş bir tını skalası sunabilmekteyken analog sistemlerin sunduğu nispeten kısıtlı tını skalasının değişebilmesi için sistemdeki belirli parçaların fiziksel olarak değiştirilmesi gerekmektedir.

1990'lara gelindiğinde, mikroişlemci temelli kişisel bilgisayarlar artık genel kullanıma yönelik yazılımlar ve algoritmalar ile gerçek zamanlı bilgisayar müziği üretebilecek noktaya gelmiştir (Doornbusch, 2009; Manning, 2004). Bu durum, bilgisayar müziğinin canlı performanslara konu olmasına olanak sağlamıştır. Bilgisayar ile üretilen sesler bu tarihten önce de canlı performanslarda sergilenmiştir fakat bu performanslarda kullanılan sesler önceden programlanmış seslerdir ve performans esnasında sıfırdan sentezlenmemektedir. Ancak kişisel bilgisayarların gerçek zamanlı müzik üretebilmesi ile, sonradan canlı kodlama olarak isimlendirilecek yeni bir performans pratiği ortaya çıkmıştır. Bu pratik, ses sentezinde kullanılan algoritmaların performans esnasında yazılması ve değiştirilmesi yoluyla müziğin gerçek zamanlı olarak (*real-time/on-the-fly*) programlanmasına dayanmaktadır.

Ron Kuivila ve The Hub tarafından öncül performansları gözlemlenebilen (Ward ve diğerleri, 2004) bu performans pratiği, 2000'li yıllarda kişisel bilgisayar sistemlerinin daha da güçlenmesiyle hız kazanmıştır. Bu dönemde kaynak kodu yoluyla sahne üzerinde canlı müziksel üretim yapan sanatçıların artması, 2003 yılı civarında bu pratiğin "canlı kodlama" olarak isimlendirilmesi (McLean, 2011) ve 2004 yılında canlı

kodlamanın geliştirilip yaygınlaşması amacıyla “TOPLAP” isimli bir organizasyonun meydana getirilmesi ile sonuçlanmıştır. (Zmölnig ve Eckel, 2007)

TOPLAP organizasyonu, canlı kodlama sanatçılarının iletişime geçebileceği, canlı kodlama pratiğini teorik ve pratik bağlamda geliştirebileceği ve dolayısıyla bir canlı kodlama kültürü oluşturabileceği bir ortam teşkil etmiştir. Oluşan bu kültür, belirli performans alışkanlıklarını da beraberinde getirmiştir. Magnusson, bu alışkanlıklardan “performans gelenekleri” olarak bahsetmektedir (2011) ve canlı kodlama alanında çalışan pek çok araştırmacının da kullanımıyla “gelenek” kelimesi belirli canlı kodlama alışkanlıklarına işaret etmek için sıklıkla kullanılır hale gelmiştir. Bu geleneklere göre canlı kodlama bir izleyici kitlesi önünde gerçekleştirilir. İcracılar genellikle boş bir sayfa ile başlar ve bestelerini sıfırdan, performans anında tasarlar. Besteler yeni kodların yazılıp eklenmesi; mevcut kodun değiştirilmesi, silinmesi ya da kopyalanarak tamamen yeni ögeler oluşturulması yoluyla evrilir ve ilerler. TOPLAP manifestosunda da (“ManifestoDraft—Toplap”, t.y.) ifade edildiği gibi, yazılan kodun izleyicilere gösterilmesi performansın önemli bir parçasıdır. Böylece izleyiciler hem icracının kod ile etkileşimini takip edebilir hem de dinlenilen sesin ögelerini ayırt edip eserin formu hakkında fikir sahibi olabilir (Magnusson, 2011a).

Bu geleneklerden de anlaşılabileceği gibi, canlı kodlama büyük oranda doğaçlamaya dayalı bir performans pratiğidir. Bunlar bütün canlı kodlama performanslarını şekillendirecek kadar katı olmasa da, bilgisayar müziği doğaçlamasında önemli bir yer edinmesine önemli derecede katkı sağlamıştır.

Dijital işlemcilerin ve kişisel bilgisayarların bahsedilen gelişimi, yalnızca bilgisayar müziği akımını etkilememiştir. 1920’lerden beri elektronik cihazlardan yararlanan popüler müzik (Manning, 2004), günümüze kadar hızla gelişen müzik teknolojisi alanı ile yakın bir etkileşim halinde olagelmiştir. Yukarıda bahsedilen bilgisayar müziği teknolojilerinin çoğunluğu popüler müzik için fazla işlev sağlamamıştır. Ancak süreç boyunca geliştirilen analog/dijital sentezleyiciler ve dijital örnekleyiciler (*sampler*) popüler müzik sanatçıları tarafından sıklıkla kullanılır hale gelmiştir. Özellikle 1980’lerden sonra oldukça erişilebilir hale gelen bu cihazlar, pek çok popüler müzik türünde önemli ölçüde yer kazanmıştır. Bu durum *techno* gibi tamamen ses teknolojisi cihazları kullanılarak üretilen müzik türlerinin ortaya çıkmasını sağlamıştır. Dolayısıyla kişisel bilgisayarların ve ses teknolojisi yazılımlarının hızlı gelişimi, popüler müzik dünyasını da en az bilgisayar müziği dünyası kadar derinden etkilemiştir. 1990’larda yaygınlaşan dijital ses atölyesi (*Digital Audio Workstation - DAW*) yazılımlarının sunduğu hassas araçlar, analog sistemlerle mümkün olmayacak kesinlikte çalışmayı mümkün kılmıştır (Doornbusch,

2009). Aslen grafik yazılımlarınca geliştirilen ve ana uygulamaya küçük eklentilerle çeşitli yeni fonksiyon ve özellikler kazandıran eklenti (*plug-in*) sistemi, 1990'larda hızla yaygınlaşan DAW'lar ile de kullanılmaya başlanmıştır (Collins, 2012). Günümüzde DAW'lar ve ses eklentileri, hem müzik prodüksiyonu hem elektronik müzik tasarımında endüstri standardı haline gelmiştir. Dijital kayıt yöntemlerinin gelişmesi, DAW'ların sağladığı kullanım pratikliği ve ses eklentilerinin daha önceden münferit donanımlar ile sağlanan pek çok işlevi tek bir bilgisayara kazandırması ile popüler müzik önemli ölçüde dijitalleşmiştir. Bu dijitalleşme sürecine internet ortamının hızlı gelişiminin de eklenmesi ile, günümüzde neredeyse sadece bilgisayar üzerinden üretilen ve icra edilen pek çok popüler elektronik müzik türü ortaya çıkmıştır.

Bilgisayarlar oldukça büyük imkânlar sağlamaktadır fakat hâlâ bazı sorunlar mevcuttur. Bu sorunlara örnek olarak popüler elektronik müziğin canlı performansı gösterilebilir. Popüler elektronik müzik, pek çok farklı türü barındırsa da, genel yapısı itibariyle birçok elektronik ses ve enstrümandan bir araya gelmiş bir orkestranın müziğine benzetilebilir. Bilgisayarlar bu orkestra müziğinin bestelenme sürecinde oldukça geniş olanaklar sunmaktadır ancak orkestranın canlı performans içerisinde nasıl yönetileceğine dair oldukça az araç bulunmaktadır. Bir eserde aynı anda tetiklenen birçok farklı ses bulunabileceği için, icracının bu seslerin hepsi ile aynı anda ilgilenmesi oldukça zordur. Bu sebeple icra edilecek eserlerin belirli kısımlarının kayıttan, belirli kısımlarının ise *launchpad* ya da *MIDI klavye* gibi enstrümanlar ile icracı tarafından çalınması; sıklıkla kullanılan bir müzik pratiği haline gelmiştir. Ancak bu pratik belirli sorunları beraberinde getirmektedir. Bu sorunlardan en çok dikkat çekeni, kayıtlı materyalin kullanılması ile eserin yapısına dair neredeyse tüm tasarrufun müzisyenden kayıtlı materyale devredilmesidir (Dannenberg, Gold, Liang ve Xia, 2014). Bu nedenle Kayıtlı materyal kullanılan bir eserde icracıların hızlanma, yavaşlama ya da eserin belirli bir kısmına tekrar dönme gibi özgürlükleri kısıtlanmakta; öte yandan eser içerisinde doğaçlama imkanı oldukça azalmaktadır (Kjus ve Danielsen, 2016).

Popüler elektronik müzikteki canlı performans güçlüklerinin canlı kodlama ile çözümlenebileceği varsayılabilir. Bunun için canlı kodlamanın popüler elektronik müzik performansında kullanılabilecek bir araç haline getirilmesi gerekmektedir. Ayrıca bu aracın programlama deneyimine sahip olmayan kimseler – özellikle popüler elektronik müzisyenler – tarafından anlaşılabilir ve kullanılabilir olması da oldukça önemlidir. Bu şartlar sağlandığı takdirde canlı kodlamanın elektronik popüler

müzik içerisindeki yaygınlığı artacak ve bir performans aracı olarak kullanılabilirliği daha geniş kitlelerce denenebilecektir.

Mevcut canlı kodlama uygulamaları arasında amaca en uygun uygulamalardan birinin Sonic-Pi olacağı düşünülmüştür. Zira yukarıdaki tarihçede de bahsedildiği üzere bilgisayar müziği ve dolayısıyla onun bir alt dalı olan canlı kodlama zaman içerisinde çoğunlukla akademik ve deneysel bir alan haline gelmiştir. Bu durum, üretilen canlı kodlama programlarının da çoğunlukla bilgisayar müzisyenlerine hitap eden akademik uygulamalar olmasına sebep olmuştur. Dolayısıyla mevcut canlı kodlama programlarının çoğunluğu, anlaşılabilir olmalarından ziyade işlevsel olmaları hedeflenerek geliştirilmiştir.

Sonic-Pi, geliştirilme amacı bakımından diğer canlı kodlama uygulamalarından ayrılmaktadır. Çünkü Sonic-Pi, aslen ilkokul öğrencilerine kodlama öğretmek amacıyla geliştirilmiştir (Aaron ve Blackwell, 2013). Dolayısıyla aranan programlama deneyimine sahip olmayan kimseler tarafından anlaşılabilir ve kullanılabilir olma gerekliliğini büyük oranda sağlamaktadır. Öte yandan beraberinde sunulan eğitici materyalin çeşitliliği, Sonic-Pi'yi öğrenilmesi en kolay dillerden biri haline getirmektedir. Sonic-Pi'nin ana akımda diğer canlı kodlama dillerine oranla daha popüler olmasının da bu öğrenme ve anaşılma kolaylığıyla ilgili olduğu düşünülebilir.

Amaca yönelik ikinci gereklilik, bahsedildiği üzere, belirlenen canlı kodlama dilinin popüler elektronik müzik performansına uygun hale getirilmesidir. Popüler elektronik müzikte sıkça kullanılan araçların canlı kodlama programınca desteklenmesi, bu amaca yönelik önemli bir aşamadır. 1990'lardan günümüze müzik prodüksiyonu ve elektronik müzik tasarımında önemli yer kazanmış ses eklentileri, elektronik müzikte sıkça kullanılan araç tanımına uymaktadır. Ancak anlaşılırlığı dolayısıyla seçilen Sonic-Pi – ve diğer canlı kodlama uygulamalarından hiçbiri – ses eklentilerinin kullanımını desteklememektedir. Ses eklentilerinin desteklenmemesi, Sonic-Pi'nin elektronik müzisyenler için cazibesini ve kullanışlılığını sınırlamaktadır.

Bu çalışma ile ses eklentilerinin Sonic-Pi yazılımınca desteklenmesi sağlanacaktır. Yazılacak olan yardımcı bir yazılım ile ses eklentilerinin yüklenmesi ve Sonic-Pi'den gelecek komutlara göre çalıştırılması sağlanacaktır. Sonic-Pi'nin kaynak koduna yapılan eklemelerle de yardımcı yazılım ile Sonic-Pi arası iletişim sağlanacaktır. Bu imkanlar ile Sonic-Pi ile gerçekleştirilen bir canlı kodlama performansı esnasında ses eklentilerinin yüklenmesi mümkün olacaktır. Ayrıca eklentilerin ses çeşitliliğini sağlayan parametreler de performans esnasında değiştirilebilecek, kullanılan sesler

istenilen şekle sokulabilecektir. Bu gelişmeler ile Sonic-Pi'nin ses yelpazesinin oldukça büyük bir çeşitliliğe sahip olması amaçlanmaktadır. Böylece Sonic-Pi'nin elektronik müzisyenler tarafından kullanılma potansiyelinin ve tasarım olanaklarının artırılacağı düşünülmektedir. Sonic-Pi'ye yapılan geliştirmenin canlı kodlamanın popülerleşmesine ve bir popüler elektronik müzik enstrümanı olarak tanınmasına hizmet etmesi amaçlanmaktadır. Böylelikle elektronik müziğin canlı performans sorununun giderilmesine katkı sağlanacağı öngörülmektedir.

2 KAVRAMSAL ÇERÇEVE VE LİTERATÜR TARAMASI

2.1 Canlı Kodlama Kavramı

2.1.1 Canlı Kodlamanın Tarihçesi

Canlı kodlama akımı, kökleri 1950'lerde atılan bilgisayar müziği akımının gelişiminin bir basamağı olarak varlık kazanmıştır. Bu sebeple canlı kodlamanın işlevini, önemini ve amaçlarını tam olarak kavrayabilmek için bilgisayar müziğinin tarihçesinin de kısaca incelenmesi gerekmektedir.

Bilgisayarlardan ses elde edilmesi, ilk bilgisayar olan ENIAC sisteminin 1947 yılında çalışır hale gelmesi ile gerçekleşmiştir. Ancak bu dönemde elde edilen ses çıktısı, genellikle bilgisayarın üzerinde çalıştığı programın durumuna dair bilgi almak amacıyla kullanılmıştır. Bilgisayarın işlemcisine bağlanan bir hoparlör ile takip edilen sesin rastgele aralıklarla perde değiştirmesi programın farklı veri işleme süreçlerinde olduğunu, sabit bir perdede takılı kalması ise programın sonsuz bir döngüye girdiğini ve kapatılması gerektiğini göstermiştir ("Short History of Computer Music", t.y.). Bu farklı veri işleme süreçlerinin farklı perdelerde ses üretmesi prensibi kullanılarak ortaya koyulan ilk müzikal çıktı, 1950'lerin başında Geoff Hill tarafından programlanmıştır (Doornbusch, 2004).

Daha sonra bilgisayar müziği alanındaki en büyük yenilik sayılabilecek dijital ses sentezi, New Jersey'deki Bell Laboratuvarları'nda geliştirilmiştir. Telekomünikasyon teknolojileri alanında çalışan Bell Laboratuvarları'nın akustik araştırma grubu, telefon konuşmalarının dijital formda iletilmesi üzerine çalışmalar yapmıştır (Manning, 2004). Laboratuvardaki araştırmacılardan biri olan Max Mathews, alandaki çalışmaların bilgisayar müziği alanındaki kullanılabilirliğini fark etmiş ve dijital ses araştırmalarına dair bulgulardan faydalanarak dijital ses sentezi üzerine çalışmıştır (Mathews, 1963). Bu çalışmaların sonucu olarak ortaya çıkan ve bilgisayar yoluyla dijital ses sentezi amacıyla yazılan ilk program, 1957 yılında ilk sürümü geliştirilen MUSIC programıdır (Keislar, 2009).

MUSIC programı, dönemi içerisinde gördüğü geniş ilginin yanısıra günümüzde kullanılan dijital ses sentezleme teknolojilerinin temelini oluşturması açısından oldukça önemlidir. Program, ilk sürümünün geliştirildiği 1957 yılından 1980'lere kadar farklı araştırmacılar tarafından MUSIC II, MUSIC III gibi onlarca çeşitli sürümle geliştirilmiş ve farklı sistemlere adapte edilmiştir. 1986 yılında MIT üyesi Barry Vercoe tarafından MUSIC XI'in C dilinde tekrar yazılması ile ortaya çıkan CSOUND programı ise, günümüzde hâlâ kullanılmaktadır (Manning, 2004).

MUSIC programı ve türevlerinin artan işlevselliği ve popülerliği, dönemin birçok bestecisinin dikkatini çekmiştir. Özellikle 1960 yılında MUSIC III ile kullanıma sunulan ve sonraki sürümlerle gittikçe geliştirilen "unit generator" özelliği ile programın kullanılabilmesi için besteciden istenen komutların sadeleştirilmesiyle besteleme süreci basitleşmiş ve programın besteciler açısından kullanımı kolaylaşmıştır. Programın geliştirilme süreci boyunca James Tenny, F. B. Moore, Jean Claude Risset gibi birçok besteci laboratuvarlara sıkça uğrayarak program ile çalışmaya başlamış, önemli eserler bestelemiştir ("Short History of Computer Music", t.y.). Bu eserlerin bestelenmesi, Max Mathews'ın "The Digital Computer as a Musical Instrument" makalesinin 1963 yılında Science dergisinde yayınlanması ve konu ile ilgilenen araştırmacıların Avustralya, Kanada, Fransa, Birleşik Krallık ve Amerika'daki çeşitli araştırma merkezlerinde MUSIC programlarını kullanması ile bilgisayar müziği alanı daha fazla yaygınlık ve görünürlük kazanmıştır. Daha sonra 1970'lerde bilgisayar müziğinin gelişimi hızla artmış, 1974'te *International Computer Music Association* kurulmuş*, International Computer Music Conference* ilk kez düzenlenmiş ve 1977'de elektronik müziğin günümüzde de en önemli dergilerinden olan *Computer Music Journal* yayın hayatına başlamıştır.

1970'li yılların ortalarından başlayan süreçteki anlamlı ilerlemelere rağmen, bilgisayar müziği alanının daha fazla besteciye ulaşmasının önündeki bazı teknik engeller varlığını sürdürmektedir. Daha açık bir ifadeyle, ses sentezinin yapılacağı bilgisayarların çok büyük ve pahalı makineler olması ve birkaç dakikalık bir bilgisayar müziği eserinin bestelenmesinden analog kayıt olarak elde edilmesine kadar geçen sürenin uzunluğu bestecilerin çalışmalarını zorlaştırmaktadır. Paul Doornbusch, tipik bir bilgisayar müziği eserinin meydana gelme aşamalarını şu şekilde anlatmıştır:

> MUSIC V; popülerliği ve ulaşılabilirliği bakımından bilgisayar müziği için büyük bir ilerleme teşkil etse de, kullanımı hala oldukça zordu. Bilgisayarlar genellikle erişilebilir olmazdı. Bestecinin, nota ve orkestra dosyalarını oluşturan delikli kartlarını bir kutu içerisinde sentezleme işleminin yapılacağı hesaplama tesisine götürmesi gerekirdi. Bu tesisteki hesaplama genellikle bütün bir gece sürerdi çünkü bir dakikalık sesin hesaplanması birçok saat sürebilirdi. Eğer bir hata meydana gelirse, bu hatanın düzeltilmesi ve düzeltilmiş halinin tekrar tesise sunulması gerekirdi. Program düzgün çalıştığında ve hesaplama tamamlandığında, besteciye çıktıları içeren bir manyetik bant verilirdi, ancak bu bant içerisinde örnek (*sample*) bilgisi barındıran bir dijital bant olurdu. Bu aşamadan sonra bestecinin – belki de oldukça uzakta olan – başka bir tesise gidip dijital bandı bir dijital-analog dönüştürücüden geçirmesi ve analog bir banda kaydettirmesi gerekirdi. Ancak bu işlemin ardından – programın yazılmasından günler ya da haftalar sonra – hesaplamanın sonucu duyulabilir hale getirilebilirdi.(Doornbusch, 2009,s.48)

Paul Doornbusch'ın da örneklerle anlattığı güçlükler gibi nedenlerle, bilgisayar müziği alanında denemeler yapmak isteyen bestecilerin önemli bir kısmı alandan uzaklaşmaştır. Ayrıca 1970'lerde analog sentez teknolojisi hızlı bir ilerleme kaydetmiş ve bestecilere anlık sonuçlar sunabilecek noktaya gelmiştir. Anlog sentezin anlık sonuçlarının dijital sentezin zorluğuyla kuvvetli bir zıtlık göstermesi de, elektronik düzlemde çalışmak isteyen bestecilerin büyük bir kısmının analog stüdyolara yönelmesi sonucunu doğurmuştur (Manning, 2004).

1970'lerin sonunda araştırma merkezlerinin çoğu, bilgisayarla müzik üretimini kolaylaştırmak için, gerçek zamanlı sentezlemeyi mümkün kılacak yazılım ve donanımların geliştirilmesi üstüne çalışmıştır. Ancak dönemin işlemci teknolojisi henüz gerçek zamanlı algoritmik sentezlemeyi destekleyecek kadar ilerlemiş değildir ve analog sentezleme teknolojisi hâlâ hem kullanım hem çıktı açısından daha caziptir. Bu sebeple, analog sentezleme teknolojilerinin mikroişlemciler yoluyla dijital olarak kontrolü (hibrid sistemler) üzerine pek çok çalışma yapılmıştır. ("Short History of Computer Music", t.y.). Sonuçta birçok yeni enstrüman tasarımı ortaya çıkmış ve bazıları ticari dolaşım içerisinde yer kazanmıştır. Dönemin en çok ses getiren buluşlarından biri olan MIDI protokolünün, hibrid sistemler üzerine yapılan çalışmaların da etkisiyle ortaya çıktığı savunulabilir (Pejrolo ve Metcalfe, 2017). Ancak gerçek zamanlı dijital sentezleme sistemlerine ilişkin çalışmalar, hâlâ yeni gelişmelere açıktır.

Dijital sesin bilgisayarlar tarafından gerçek zamanlı sentezlenmesini amaçlayan çalışmalar hızlanmış olsa da bu çalışmalar nihayetinde dönemin işlemleme teknolojisine bağımlıdır ve gerçek zamanlı sentezleme ye yönelik gelişme, üretilen bilgisayarların işlemleme hızı ile paralel olarak ilerlemektedir. Minibilgisayarlar 1970'lerde boyutu ve ucuzluğuyla geniş çevrelerce erişilebilir hale gelmiştir (Buradaki mini sözcüğü iki buzdolabı büyüklüğüne karşılık gelmektedir). Fakat en büyük dönüm noktası, aynı dönemde silisyumun elektriksel özelliklerinin keşfi ile hızla gelişmeye başlayacak olan mikroişlemci teknolojisi olmuştur (Manning, 2009). Ancak bu işlemciler hâlâ gerçek zamanlı sentezleme için oldukça yavaştır. Diğer taraftan, mikroişlemcilerin gördüğü ilgi sonucunda tüketim taleplerinde artış olmuştur. Bu artış, mikroişlemci pazarının gittikçe genişlemesi ve bazı imalatçıların mikroişlemci, ufak hafıza bankası ve basit bir arayüzden oluşan kendin-yap mikroişlemci setlerini kullanıma sunmasıyla sonuçlanmıştır. Zamanla bu setler gittikçe yaygınlaşmış ve "bilgisayar hobisi" gibi yeni bir hobi alanı ortaya çıkmıştır. Böylelikle de bilgisayar teknolojilerinin evde kullanılma potansiyeline dair bir

farkındalık oluşturmuştur (Manning, 2004) . Ancak bu donanımların yavaşlığı dolayısıyla ciddi bir ürün ortaya çıkarılamamıştır.

Bilgisayar üreticilerinin mikroişlemci setlerine gösterilen ilgiyi gözlemlemesi, mikroişlemci bazlı mikrobilgisayarların geliştirilme sürecini hızlandırmıştır. Bu gelişmelere paralel olarak Synclavier ve Fairlight CMI gibi belirli sentezleme, modelleme veya örnekleme (*sampling*) işlemlerini yapabilen mikroişlemci bazlı enstrümanlar geliştirilmiştir. Fakat MUSIC tabanlı programların sunduğu teorik düzlemde sınırsız sentezleme imkanına ulaşılamamıştır. Daha sonra pek çok şirketin alandaki potansiyeli görmesi ile "mikrobilgisayar" terimi "kişisel bilgisayar" olarak değiştirilmiş ve kişisel bilgisayar teknolojileri büyük bir hızla ilerlemeye başlamıştır. 1990'ların ortalarına gelindiğinde ise mikroişlemci tabanlı kişisel bilgisayarlar, dönemin en çok kullanılan gerçek zamanlı olmayan sentez programı CSOUND'u artık gerçek zamanlı çalıştırabilecek kadar hızlanmıştır (Doornbusch, 2009)

Bilgisayarların gerçek zamanlı sentezlemeyi destekleyebilir hale gelmesi, bilgisayar müziği açısından oldukça büyük bir gelişme olmuştur. MUSIC programının geliştirildiği 1957 yılından bu yana, bilgisayar müziğinin önündeki engellerden biri ve belki de en önemlisi olan besteleme ile üretilen seslerin duyulması arasındaki uzun süre, teorik olarak sıfırlanmıştır. Bir başka büyük sorun olan sentezlemenin yapılacağı bilgisayarların ulaşılma güçlüğü ise; gittikçe gelişen, ucuzlayan ve yaygınlaşan kişisel bilgisayar teknolojileri sayesinde çok büyük oranda aşılmıştır. 1990'ların sonundan itibaren artık bilgisayar müziği yapmak için bir laboratuvara değil, sadece en yeni bilgisayar oyunlarını kaldırabilecek güçte bir bilgisayara ihtiyaç duyulmaktadır ("Short History of Computer Music", t.y.).

1990'ların sonundan itibaren artık temel problem cümlesi "Sentezleme işlemi nasıl gerçek zamanlı yapılabilir?" değil, "Gerçek zamanlı sentezleme nasıl en etkin ya da kullanışlı şekilde yapılabilir?" şeklinde anlamlı sonuçlar doğurabilecek bir hale dönüşmüştür.

Gerçek zamanlı sentezlemenin olanaklarıyla ortaya çıkan ilk önemli sonuç, dijital olarak sentezlenmiş sesler kullanmak isteyen bestecilerin artık tasarılarını neredeyse hiçbir bekleme süresi olmadan dinleyebilmeleridir. Böylelikle besteciler artık birçok farklı fikri seri olarak deneyip çeşitli farklı yöntem ve algoritmaları daha büyük bir titizlikle inceleyebilmektedir. Diğer bir önemli sonuç ise, gerçek zamanlı sentezleme ile bilgisayarın bir enstrüman olarak potansiyelinin ve işlevinin de çok büyük ölçüde artmasıdır. Bilgisayar ile müzik üretiminin bu açısındaki gelişmeleri de

tam olarak kavrayabilmek için, bilgisayarın bir performans enstrümanı olarak tarihçesinin de kısaca incelenmesi önemlidir.

Aslında bilgisayarlar gerçek zamanlı sentezleme mümkün olmadan çok daha öncelerden beri, müzik enstrümanı olarak kullanılmaktadır. Teyit edilmiş ilk müzikal performans, 1951'de Australian Computer Conference'ın açılış seromonisinde CSIRAC (*Commonwealth Scientific and Industrial Research Automatic Computer*) bilgisayarı vasıtasıyla basit melodilerin seslendirilmesi ile gerçekleştirilmiştir (Doornbusch, 2004). Ancak bu performans tamamen dijital ses sentezleme teknolojisine değil; işlemciye bağlanan hoparlöre gönderilen sinyalin çeşitli komutlarla manipüle edilmesi ile gerçekleştirilen bir tür analog sentez yöntemine dayanmaktadır. Diğer bir ifadeyle bu performans türünde bilgisayar sesin kendisini üretmemektedir ve yalnızca hoparlöre gönderilecek elektrik sinyalini kontrol etmektedir. Dolayısıyla ortaya çıkan ses oldukça basittir ve sesin zarfı veya tınısı üzerinde ayarlama yapmak mümkün olmamaktadır. Bilgisayarlardan analog sentez yöntemiyle üretilen ses, Doornbusch'un tarafından "bir bilgisayarla elde edilmesi beklenebilecek en ham ses" olarak tanımlanmıştır ve Bell Laboratuvarları İletişim Bilimleri grubunun yönetici müdürü John Pierce'ın ifadesine göre bilgisayarların "cızırtılar ile müzik yaptığı" döneme aittir (Doornbusch, 2004). Dijital ses sentezinin henüz ortaya çıkmadığı bu dönemde bilgisayar teknik olarak bir enstrüman olarak kullanılmıştır ama bahsedilen kısıtlılıklar sebebiyle bestecilerin ilgisini çekmemiş, dolayısıyla ciddi eserlerde kullanılmamıştır.

Dijital ses sentezi, erken dönem bilgisayar müziğinin "elde edilebilecek en ham ses" sorununu büyük ölçüde çözmüştür. Çünkü teorik olarak dijital sentez yöntemiyle elde edilen seslerde tını kısıtlaması yoktur, doğru algoritmalar bulunduğu sürece var olan her ses elde edilebilir (Mathews, 1963). Ancak bilgisayarlar ile geleneksel enstrümanlar – ve hatta analog sentezleyici sistemleri – arasında hâlâ iki büyük fark mevcuttur: bilgisayarlar sesi gerçek zamanlı olarak yaratamamaktadır, ve sahneye kolaylıkla yerleştirilemeyecek kadar büyüktür. Bu yüzden, bilgisayarlar mikroişlemciler yeterince güçlenene kadar bir performans aracından ziyade bir besteleme aracı olarak kullanılmıştır.

Mikroişlemcilerin kuvvetlenmesi ile bilgisayarın bir enstrüman olarak kimliği yeniden tanımlanmıştır. Artık bilgisayarlar hem sahnede kullanılacak kadar küçük ve hafiftir, hem de gerçek zamanlı ses üretebilmelerinin bir sonucu olarak canlı performanslar esnasında sıfırdan ses üretebilecek seviyeye gelmiştir. Böylece bilgisayarların performans enstrümanı olarak kullanılmasının ve hatta bilgisayarlar ile doğaçlama

müzik icra edilmesinin önü açılmıştır. Canlı kodlama, bu iki imkan üzerine çalışan bir performans pratiğidir.

Canlı kodlama kavramı; Ward, Rohrhuber, Olofsson, McLean, Griffiths, Collins ve Alexander (2004) tarafından "bir programın [belirli kısımlarının] program çalışırken yazılması" olarak tanımlanmıştır ve şu şekilde açıklanmıştır:

> Canlı bilgisayar müziği ve görsel performans, artık algoritmik süreçlerin interaktif kontrolünü de kapsayabiliyor. Normal uygulamada, böyle bir faaliyetin arayüzü konserden önce belirlenir. Yeni canlı kodlama disiplinine göre algoritmaların kontrol yapıları programın çalışma-zamanı (run-time) esnasında değiştirilebilir. Bu tür algoritmik ince işçilik en doğal haliyle yazı tabanlı ve yorumlanan (interpreted) bir programlama dili ile tetkik edilebilir. Birçok uygulamacı, birçok farklı platform ve dilin yardımıyla hem halka açık hem özel bağlamlarda bu imkanların keşfi üzerine halihazırda çalışmaktadır. (s.243)

2.1.2 Canlı Kodlamanın Tanımı

"Canlı kodlama" isimlendirmesi, 2003 yılı civarında ortaya atılmıştır (McLean, 2011) ancak daha eski tarihlerde verilen tanıma büyük ölçüde uyan öncül canlı kodlama performanslarına da rastlanmıştır. İlk öncül canlı kodlama performanslarından biri, Ron Kuivila'nın 1985 yılında Amsterdam'daki yeni müzikal enstrümanlar için araştırma ve geliştirme merkezi olan STEIM'da (*Studio for Electro Instrumental Music*) gerçekleştirdiği performanstır. Kuivila'nın kullandığı yazılım 1980'lerin teknik yetersizlikleri sebebiyle konser sırasında hata vermiş ve çökmüştür ancak sanatçı, program çökmeden önce ilgi çekici bir müzikal performans ortaya koymayı başarmıştır (Roads, 1986). Kuivila performansında FORTH dili üzerine çalışmış ve bir Apple II bilgisayarı kullanmıştır. Bu performans öncül bir canlı kodlama performansı olarak değerlendirilmektedir ancak ilerleyen bölümlerde bahsedilecek canlı kodlama pratiklerinin tüm özelliklerini taşımamaktadır. Sonradan oluşacak canlı kodlama geleneği ile en büyük farkı, kullandığı kodun seyirciye sunulmuş olmamasıdır (Ward ve diğerleri, 2004).

Kuivila ile yakın tarihlerde karşılaşılan diğer öncül canlı kodlama performansları, 1980'ler boyunca İngiliz deneysel ağ bilgisayarı grubu "*The Hub*" tarafından gerçekleştirilmiş performanslardır (Zmölnig ve Eckel, 2007). The Hub grubu, San Fransico körfez bölgesinde kendi yazılım ve donanımlarını geliştiren bilgisayar müzisyenlerinden meydana gelmiştir (C. Brown ve Bischoff, 2002)). Grup, The League of Automatic Music Composers (kısaca The League) grubu ile birlikte bilgisayar ağlarının müzikal kompozisyon ve performans alanlarındaki potansiyelini

araştırmak için yola çıkan ilk iki gruptan biridir ve The League üyesi Tim Perkis'in bir program notunda da bahsedildiği üzere aslen The League grubunun yeni çalışmaları ile ortaya çıkan dağınıklığı gidermek için kurulmuştur (C. Brown ve Bischoff, 2002). The League grubunun ilk çalışmaları, grup üyelerinin basit bir klavye ve tek kartlı bir bilgisayardan oluşan KIM-1 (*Keyboard Input Monitor*) sistemlerini biribirine ağ bağlantısı yoluyla bağlaması ve programlanmış bestelerin tüm bilgisayarlarca birlikte çalınması şeklinde meydana gelmiştir. İlk çalışmalarda müzisyenler bilgisayar sistemlerini kurduktan sonra bilgisayarları önceden programlanmış besteleri çalmaya bırakmış, konseri seyirciler ile birlikte takip etmişlerdir. Ancak grup bir süre sonra çalışan bilgisayarlara çalışma esnasında müdahale etmeye başlamış, ortaya çıkan müziği şekillendirmek amacıyla parametrelerle oynamışlardır. Grubun performansları oldukça ilgi çekmiştir ve bu ilgi zamanla daha fazla bilgisayar müzisyeninin grup ile müzik yapmak istemesine sebep olmuştur. Ancak mevcut sistem yeni müzisyenlerle çalışmak için pek uygun değildir. Her performans için birçok bilgisayar arasında karmaşık bağlantıların kurulması gerekmektedir. Fakat çok sayıda karmaşık bağlantı ile hem henüz yeterince güçlü olmayan bilgisayarların hata verme ihtimali artmakta, hem de yeni müzisyenlerin performans grubuna eklenebilmesi oldukça zorlaşmaktadır. Karşılaşılan sorunların kolay kontrol edilebilen ve neredeyse her tür bilgisayar ile bağlanılabilecek kadar standart bir arayüze sahip yeni bir sistemle çözülmesi öngörülmüştür. Bu sistem; kullanıcılar arasında mesaj transferini sağlayan ve her kullanıcının faaliyetinin diğer kullanıcılara sunulabilecek bir veri bankasında toplandığı ufak bir bilgisayar olarak vücut bulmuştur. Kurulan sistem bilgisayarına "aktarma merkezi" anlamına gelen "The Hub" ismi verilmiştir ve müzisyen grubunun performansları da bu isim altında yapılmıştır (C. Brown ve Bischoff, 2002). Ron Kuivila ve The Hub gibi münferit örnekler mevcutsa da, canlı kodlamanın yaygınlaşması yirmibirinci yüzyılın başında gerçekleşmiştir.

2000'lerden itibaren canlı kodlama performanslarının ve müzisyenlerinin sayısının giderek artması, bir iletişim ve teşkilatlanma ihtiyacını beraberinde getirmiştir ve bir grup canlı kodlama müzisyeninin bir araya gelip canlı kodlamanın geliştirilmesi ve yaygınlaştırılmasını amaçlayan bir organizasyon kurması ile sonuçlanmıştır. TOPLAP olarak isimlendirilen kuruluş, "15 Şubat 2004 yılında saat 01:00'da Hamburg'da bir barda kurulmuştur" (Ward ve diğerleri, 2004). TOPLAP, hem bir mail grubu yoluyla hem de bir internet sitesi aracılığıyla canlı kodlama müzisyenleri arasında irtibat sağlamıştır. Ayrıca internet sitesinde çeşitli canlı kodlama konser ve atölye duyurularına, canlı kodlama ile ilgili akademik yazılara, canlı kodlamanın

tarihine dair çeşitli kayıt ve belgesellere, canlı kodlama konusundaki son gelişmelerin ele alındığı blog ve köşe yazılarına ve mevcut canlı kodlama programlarının, dillerinin ve kütüphanelerinin derlendiği geniş bir listeye yer vererek canlı kodlama kültürüne önemli katkılarda bulunmuştur. Bunların yanısıra yerel canlı kodlama topluluklarına kendilerini TOPLAP "node"ları olarak kaydetme imkanı verilmiştir. "Ağ noktası" olarak çevrilebilecek node kelimesi, her yerel topluluğun genel TOPLAP ağının bir bağlantı noktası olduğunu sembolize etmektedir. Temmuz 2020 itibariyle TOPLAP'e kayıtlı 21 farklı ülkeden 30 adet node bulunmaktadır ("TOPLAP nodes", 2016).

TOPLAP'in canlı kodlama alanına yaptığı önemli katlkılardan bir başkası da, TOPLAP resmi sitesinde yayımlann canlı kodlama manifestosudur ("ManifestoDraft—Toplap", t.y.). Ward vd.'nin TOPLAP'in kuruluşuna dair bilgiler verdiği makalede TOPLAP'in canlı kodlama pratiğine hakemlik etmek gibi bir amacı olmadığından ancak bu yeni alanın keşfedilme sürecini beslemek amacıyla kalite ve açıklığa dayalı bazı faydalı prensiplerin teşvik edilebileceğinden bahsedilmiştir (Ward ve diğerleri, 2004). Dolayısıyla bahsedilen manifestonun bağlayıcı değil yol gösterici olması amacıyla hazırlandığı düşünülebilir. Ancak yazım amaçları ne olursa olsun TOPLAP'in canlı kodlama üzerine çalışan en büyük organizasyonlardan biri olması, yazılan manifestonun çoğu canlı kodlama performansı üzerinde önemli bir etki yaratmasına sebep olmuştur. Yayımlanan manifesto şu şekildedir:

Şunları talep ediyoruz:

- Bize icracının zihnine, yani tüm insan enstrümanına erişme imkanı sağlayın.
- Bilmesinlercilik tehlikelidir. Bize ekranlarınızı gösterin.
- Programlar kendilerini değiştirebilen enstrümanlardır.
- Programların ötesine geçmek gereklidir – bunun yolu yapay dildir.
- Kod duyulduğu kadar görülmelidir de, ayrıca altta yatan algoritmalar görsel sonuçlarına şahit olunduğu gibi aynı zamanda incelenebilmelidir.
- Canlı kodlama araçlardan ibaret değildir. Algoritmalar düşüncedir. Elektrikli testereler araçtır. Algoritmaların bazen elektrikli testerelerden daha zor fark edilebilir olması bu yüzdendir.

Etkileşim ve derinlik sürekliliğini kabul ediyoruz, ancak şunları tercih ediyoruz:

- Algoritmalara dair kavrayış
- Zihinsel maharetliliğin izlenimci/dışavurumcu bir göstrerimi olarak hünerli algoritmik doğaçlama
- Yedeksizlik (minidisk, dvd, yedek bilgisayar)

Şunları teslim ediyoruz:

- Canlı kodlamaya yabancı bir izleyici kitlesinin kodu takdir beğenmesi için anlaması gerekmemektedir, tıpkı bir gitar performansını takdir beğenmesi için gitar çalmayı bilmesinin gerekmediği gibi.
- Canlı kodlamaya el hünerinin etkileyici bir gösterimi ve yazım sisteminin sergilenmesi eşlik edebilir.
- Performans etkileşimsel süreklilikten oluşur. Bu etkileşim zaman zaman sanat eserinin parametre düzlemlerinden oluşan kontrollerden, zaman zaman jestlerden, özellikle dışavurumsal detayların doğrudanlığından meydana gelir. Enstrümantal müzikte dışavurumun bir aracı olarak gelenekselleşmiş temasa dayalı zamanlama sapmaları kod ile yaratılamamaktadır ancak geçmişi tekrar etmeye de gerek yoktur. Şüphesiz kod yazımı ve fikirsel dışavurum pratikleri kendi nüanslarını ve geleneklerini yaratacaktır. ("ManifestoDraft—Toplap", t.y.)

TOPLAP gibi girişimlerin de katkısıyla 2000'ler boyunca yaygınlaşmaya devam eden canlı kodlama, kendine yeni performans alanları bulmuştur. Bu performans alanlarına verilebilecek belki de en önemli örnek, *algorave*lerdir. "Algorave" isimlendirmesi 2011 yılında Alex McLean tarafından "algorithm" ve "rave" kelimelerinin birleştirilmesiyle aslen bir şaka olarak meydana getirilmiştir. "Rave" kelimesi "çıldırmak" anlamına gelmektedir ve ilk olarak 1950'lerin sonlarında İngiltere'deki bohem partilerini isimlendirmek için kullanılmıştır. Kelime 1980'lerin başında Londra'da yapılan yasadışı ambar partileri ile tekrar kullanıma girmiş ve günümüze kadar gelmiştir. Günümüzde artık bir grup insanın elektronik müzik eşliğinde dans etmek için bir araya geldiği hemen hemen her senaryoda kullanılır durumdadır (Evans, 1992). Algorave kavramı ise, www.algorave.com sitesinde şu şekilde tanımlanmıştır:

> Algorave, büyük oranda birbirini takip eden mükerrer koşullu ifadelerin iletimi ile nitelenen veya tamamen bu şekilde oluşturulan seslerden oluşur. Bugünlerde neredeyse her tür elektronik müzik yazılımlar vasıtasıyla meydana getirilmektedir, ancak bu yazılım algoritmalarını geliştiren insanlarla müziği yapan insanlar arasında yapay bariyerler mevcuttur. IXI Lang, puredata, Max/MSP, SuperCollider, Extempore, Fluxus, TidalCycles, Gibber, Sonic Pi, FoxDot ve Cyril gibi algoritmik müzik ve görsellerin oluşturulması için yaratılmış sistemler kullanılarak bu bariyerler yıkılmakta; müzisyenler çalışmalarını algoritmik müzik formunda canlı olarak besteleyip icra edebilmektedir. Bunun çeşitli iyi ve kötü yönleri mevcuttur, ancak farklı yaklaşımlar ilgi çekici yollar açmaktadır.
>
> Bu fikir yeni değildir, ancak Algoraveler insanların müzik yapması ve bu müzik ile dans etmesi üzerine yoğunlaşmaktadır. Algorave müzisyenleri yaratıcılık vasfını yazılımlarına yüklemez, yaptıkları müziğin sorumluluğunu üstlenir ve müziği ellerindeki vasıtalarla şekillendirirler. Daha önemlisi, odak müzisyenin ne yaptığı üzerinde değil, ortaya çıkan müzikte ve bu müzikle dans eden insanlardadır. Algoraveler geçmiş ravelerdeki olağandışı sesleri kucaklar ve garip, algoritma-

> yardımlı süreçlerle yeni olağandışı ve fütüristik ritimler ve altyapılar sunar. ("Algorave", t.y.)

Algorave ismi altında sunulan ilk performans 17 Mart 2012'de Londra'da gerçekleştirilse de, algorave tanımına uyabilecek öncül performanslar 1992 yılına kadar takip edilebilmektedir (N. Collins ve McLean, 2014). Algoraveler canlı kodlama müzisyenleri tarafından ortaya çıkarılmalarının yanı sıra, itibariyle muhtevasında canlı kodlamadan başka performans pratikleri de bulundurabilmektedir. Ancak canlı kodlamanın artan popülerliği ve algorave içeriğine uygunluğu sebebiyle şimdiye kadarki her algorave kuvvetli bir canlı kodlama mevcudiyeti göstermiştir (N. Collins ve McLean, 2014).

2.1.3 Canlı Kodlama Dilleri

Canlı kodlama dilleri üzerine yapılan çalışmalar, gerçek zamanlı sentezleme olanağı ile birlikte hızlanmıştır. Kişisel bilgisayar teknolojisinin gerçek zamanlı sentezi destekleyecek seviyeye gelmesi ile bilgisayar müziği için yalnızca yeni performans alanları değil, yeni araştırma ve deney alanları da meydana gelmiştir. Gerçek zamanlı sentezin kolaylıkla ulaşılabilir hale gelmesi ile canlı kodlama çevresinde ana hedef, dijital ses sinyali işlemleme sistemlerinin kontrolünü mümkün olduğunca müzikal dışavuruma uygun hale getirmek olmuştur (A. C. Sorensen, 2018). Dolayısıyla pek çok farklı canlı kodlama dili üzerinde çalışılmıştır.

Canlı kodlamanın ilk örneklerinde Lisp (The Hub) ve FORTH (The Hub ve Ron Kuivila) gibi genel amaçlı diller (*general purpose language*) kullanılmıştır (Zmölnig ve Eckel, 2007). Lisp ve FORTH dillerinin ikisinin de yorumlanmış (*interpreted*) olması, kodun çalışırken değiştirilebilmesi konusunda kolaylık sağlamıştır (A. C. Sorensen, 2018). Ancak yine de bunlar genel amaçlı dillerdir ve canlı kodlama alanında kullanımları mümkün olsa da basit ya da verimli değildir.

Canlı kodlama, oldukça özgün bir programlama pratiğidir ve dolayısıyla kullanılacak dilin bazı özel işlevlere sahip olması gerekmektedir. Bir canlı kodlama dilinin sahip olması gereken en önemli özel işlevin zaman kontrolü olduğu söylenebilir. Bilgisayar biliminde programın çalışma zamanı genellikle işlevsel bir özellik değil, minimize edilmesi gereken bir değer olarak görülmektedir (McLean, 2011). Ancak müzikte ve dolayısıyla canlı kodlamada zamansallık, ritim gibi çok temel ve önemli bir kavramı doğurmaktadır. Dolayısıyla genel amaçlı dillerin canlı kodlamada kullanılabilmesi, ancak bu dillere bazı işlevlerin kazandırılması ile mümkün olabilmektedir.

Genel amaçlı dillere canlı kodlamada ihtiyaç duyulan işlevlerin kazandırılması ancak kullanıcının dil içerisinde belirli kompleks sistemler kurması ile mümkün

olabilmektedir. Örneğin 2000'lerde aktif olarak sahne alan Slub grubu, Perl ve REALbasic dillerini kullanarak sahnede kullandıkları ses programlarını kendileri yazmıştır. Dolayısıyla genel amaçlı dillerin canlı kodlamada kullanımının mümkün olduğu söylenebilir, ancak bunun için oldukça meşakkatli bir hazırlık ve tasarım süreci gerekmektedir.

Genel amaçlı dillerin elverişsizliği, araştırmacıların canlı kodlama için alana özgü diller (*domain specific language - DSL*) üzerine çalışmalarına sebep olmuştur. Alana özgü diller, genel amaçlı dillerden farklı olarak belirli bir uygulamaya hizmet etmesi amacıyla geliştirilen dillerdir. Canlı kodlama uygulamalarına yönelik geliştirilen DSL'ler ise, günümüzde canlı kodlama dilleri olarak sınıflandırılan dillerdir. 2003 yılında Ge Wang tarafından geliştirilen ChucK dilinin canlı kodlama için özel olarak geliştirilen ilk DSL olduğu düşünülmektedir (Zmölnig ve Eckel, 2007). 2005 yılında ise, Andrew Sorensen tarafından Impromptu dili geliştirilmiştir (Sorensen, 2005). ChucK, Impromptu ve ilerleyen tarihlerde ortaya çıkacak diğer canlı kodlama dilleri; kullanım bakımından birbirlerinden oldukça farklı olsa da genel tasarım bakımından benzerlik göstermektedir.

2.1.4 Canlı Kodlama Dillerinin Yapısı

Canlı kodlama dillerinin geneli, iki parçalı bir yapı halinde tasarlanmaktadır (A. C. Sorensen, 2018). Yapının üst katmanı kullanıcının doğrudan etkileşime geçtiği ve dilin görünürdeki özelliklerini oluşturan arayüz kısmıdır. Alt katmanda ise kullanıcıdan gelen komutları dinleyen ve bu komutlara göre sentezleme işlemlerini gerçekleştiren bir "ses motoru" mevcuttur. Ses motorlarının neredeyse tamamı düşük dereceli (*low-level*) ve yüksek performanslı C veya C++ dilleri ile yazılmaktadır (Sorensen, 2018). Ancak düşük dereceli diller – yüksek dereceli (high-level) dillerin aksine – "live editing" olarak ifade edilen kodun çalışırken değiştirilmesi pratiği için oldukça kısıtlı bir destek sunmaktadır. Dolayısıyla canlı kodlama dillerinin genel tasarımı, yüksek dereceli bir dil ile tasarlanmış kullanıcı arayüzü ve düşük dereceli bir dil ile tasarlanmış bir ses motoru şeklinde olagelmiştir. Ses motorlarının en yaygın kullanılanlarından biri ise, SuperCollider yazılımıdır.

James McCartney tarafından ilk sürümünün geliştirildiği 1996 yılından itibaren (McCartney, 1996) önemli ses sentezleme yazılımlarından biri haline gelen SuperCollider, ikinci sürümü (SC2) ile canlı kodlama için oldukça önemli bir gelişmeye imza atmıştır. SC2 itibariyle girilen kodları düzenleyen yorumlayıcı (*interpreter*), artık ses sentezi esnasında da çalışmaktadır. Böylece SuperCollider vasıtasıyla sentezlenen sesler, sentezlenme esnasında değiştirilebilir hale gelmiştir

(McCartney, 1998) ve SuperCollider canlı kodlama alanında da oldukça önemli bir program haline gelmiştir (Collins, McLean, Rohrhuber, & Ward, 2003).

McCartney, 2002 yılında geliştirdiği üçüncü sürümde SuperCollider sistemini canlı kodlama dillerinin ikili yapısına uygun olacak şekilde bölmüş, üst katmanı "SuperCollider Language", alt katmanı ise "SuperCollider Server" olarak isimlendirmiştir (McCartney, 2002). İki katman arasındaki iletişim OSC (*Open Sound Control*) protokolü baz alınarak tasarlanmıştır. Program 2002 yılında "GNU General Public License" ile lisanslanarak açık erişimli ücretsiz bir program haline getirilmiştir (Doornbusch, 2009; "SuperCollider", t.y.) . Böylece OSC protokolü yoluyla iletişim kurabilen her program SuperCollider'ın güçlü ses motorundan faydalanabilir hale gelmiştir. Bir canlı kodlama dilinin tasarlanmasındaki en zorlu süreçlerden birinin ses motorunun yazımı olduğu göz önünde bulundurulacak olursa, bu gelişimin oldukça önemli olduğu söylenebilir. SuperCollider Server'ın ücretsiz olarak erişime açılması, canlı kodlama dili tasarımı üzerinde çalışan pek çok araştırmacıya büyük bir kolaylık sağlamıştır (Sorensen, 2005). Bu sayede araştırmacılar, ses motoru yazmanın zorluklarına katlanmaksızın çalışmalarını gelişmiş kullanıcı arayüzlerinin tasarımı üzerine odaklanma fırsatı kazanmışlardır. Günümüzde kullanılan TidalCycles, ixiLang, Overtone, Sonic-Pi gibi pek çok önemli canlı kodlama dili ses motoru olarak SuperCollider Server'ı kullanmaktadır (Aaron ve Blackwell, 2013; Magnusson, 2011b; *Musikinformatik/SuperDirt*, 2020). Dolayısıyla SuperCollider'ın üçüncü versiyonunun canlı kodlama dillerine çeşitlilik kazandırdığı söylenebilir.

Meydana gelen canlı kodlama dili çeşitliliğinin de katkısıyla, canlı kodlamanın yaygınlığı ve görünürlüğü önemli ölçüde artırmıştır. *International Computer Music Conference*, 2011 yılından itibaren sunumlar ve gerçekleşecek performanslar için ayrı bir canlı kodlama kategorisi açmış, dolayısıyla canlı kodlamayı bilgisayar müziğinin canlı performansı sorununa bir çözüm yöntemi olarak tanımıştır (Magnusson, 2011a). Öte yandan canlı kodlamayı disiplinlerarası bir alan olarak tanıtmayı amaçlayan *International Conference on Live Coding*, 2015 yılında ilk oturumunu gerçekleştirmiştir ve 2020 yılı itibariyle 5 farklı ülkede düzenlenmiştir ("International Conference on Live Coding—Home", t.y.).

2.1.5 Canlı Kodlama Pratikleri

Canlı kodlama, pek çok araştırmacı tarafından farklı şekillerde tanımlanan bir performans pratiğidir. Bu farklı tanımların ortak noktalarından yola çıkarak şu şekilde bir genel tanım yapmak mümkündür: Canlı kodlama, çeşitli algoritmaların kullanılması ve bu algoritmaların performans esnasında düzenlenmesi ile meydana gelen bir performans türüdür (Magnusson, 2014). Tanımın genişliği, canlı kodlama adı altında birkaçı aşağıda örneklendirilecek pek çok farklı performans türünün doğmasına zemin hazırlamıştır.

Canlı kodlama performans türlerine ilk örnek olarak *kod DJ'liği* gösterilebilir. Kod DJ'liği performansında hazır kod parçaları ve algoritmalar kullanılır. Sanatçı sahneye çıkmadan önce belirli algoritmalardan bir seçki hazırlar ve performans esnasında bu algoritmalar farklı kombinasyon ve bağlantılarla sunulur (N. Collins ve McLean, 2014). Kod DJ'liği sanatçılarından Timeblind, performansını şu şekilde anlatmıştır:

> Görsel olarak bakıldığında benim kurulumum aslında sadece seçkilerden oluşuyor. Seçkideki nesnelerin davranışları "sabit" ve "kesinlikle tahmin edilemez" uçları arasında çeşitlilik gösteriyor. Bu davranışlara işaret edecek herhangi bir görsel işaret yok. Süreç tamamen benim bu ufak varlıklara [nesnelere] ve davranışlarına aşinalığım üzerinden ilerliyor. Tüm değişiklikler fareden çok daha hızlı olan klavye aracılığıyla yapılıyor. Nesneler bir veri bankası ortamında türetiliyor. Benim işim türetmenin yeni yollarını bulmak ve çirkin olanları ortadan kaldırmak. Birçok farklı ortamda çalıyorum ve çaldığım ortama göre canlı performans esnasında alacağım risk miktarı değişiyor. Her zaman kendimi şaşırtacak şeyler duymam lazım, ancak şapkamda birçok tavşan bulundurmam da gerekiyor. (Collins, 2003, s. 73)

Bir başka canlı kodlama performans türü, Josh Dubrau ve Mark Havryliv'in, farklı bir düşünce üzerinde çalışarak geliştirdiği P[a]ra[pra]xis yazılımı ile örneklenebilir. Yazılım, düz yazı veya şiirin gerçek zamanlı olarak yaratılması ve eşzamanlı olarak sese dönüştürülmesi amacıyla tasarlanmıştır. Kullanıcı, yazılımın ilk parçası olan P[a]ra[pra]xis Collection Editor'ü kullanarak belirli kelimelerden ve bu kelimelerin çeşitli gramatik özelliklerinden oluşan bir sözlük oluşturur. Kelimelerin muhtemel ikameleri de sözlüğe kullanıcı tarafından eklenir. Yazılımın ikinci kısımı olan Realtime P[a]ra[pra]xis ise kullanıcının yazdığı düz yazı veya şiirdeki kelimelerden bazılarını tanımlanmış ikame kurallarına göre eş zamanlı olarak değiştirir ve yazının asıl ve değiştirilmiş hallerine dair bilgiyi ses motoruna gönderir. Ayrıca yazının kendisi dışında ortalama yazı girişi hızı, cümle uzunluğu, mesaj uzunluğu gibi bazı faktörler de girilen yazının sese dönüştürülmesinde parametre olarak kullanılabilmektedir.

Dizüstü bilgisayar müzik toplulukları (*laptop ensemble*), en yaygın karşılaşılan canlı kodlama performans türlerinden biridir. Bu topluluklar birden fazla dizüstü bilgisayar müzisyeninin farklı bilgisayarlar kullanarak oluşturduğu canlı müzik gruplarıdır. Birden fazla bilgisayarın aynı ritmik temsile göre işletilebilmesi, ortak bir sistem saatine senkronize edilmeleri ile mümkün olmaktadır. Bu yüzden genellikle bilgisayarlardan biri asıl sistem saati kaynağı olarak atanır ve diğer bilgisayarlar tempo değerlerinde bu kaynağa bağlı kalacak şekilde ayarlanır. Müzisyenlerin arasındaki iletişim için ise, bir sohbet uygulaması tasarlanır (N. Collins, 2003). aa-cell, Slub, Benoit and the Mandelbrots, Wrongheaded gibi pek çok önemli canlı kodlama grubu dizüstü bilgisayar müzik topluluğu kategorisine dahil olsa da müzik ve performans yöntemleri büyük değişiklikler gösterebilmektedir. Örneğin Powerbooks Unplugged grubu sahnenin, sesin çeşitli sistemlerle güçlendirilmesinin ve sanatsal bireyciliğin birer iktidar teşhiri olduğunu savunmaktadır ve bu düşünceden hareketle birkaç performans kuralı geliştirmiştir. Bu kurallara göre konserler sırasında grup üyeleri sahnede değil seyircilerin arasında oturur ve bilgisayar ekranlarını seyircilere açar. Herhangi bir seslendirme sistemi kullanılmaz ve ses, müzisyenlerin dizüstü bilgisayarlarının hoparlörlerinden sunulur. Grup üyelerinin yazdığı kod parçaları kullanılan sohbet uygulamasına aktarılır ve isteyen her müzisyen uygulamadaki kod parçasını aynen kopyalayabilir veya değiştirerek kullanabilir. Kullanıcıların sisteme sunduğu her ses komutu rastgele veya algoritmik olarak üretilmiş bir sayı ile etiketlenir. Bu sayı sesin hangi bilgisayardan çalınacağını gösterir. Dolayısıyla hangi kullanıcının kodunun hangi bilgisayardan seslendirileceği belirsizleştirilmiştir (Rohrhuber ve diğerleri, 2007).

Canlı kodlama alanında bahsedilmesi gereken başka bir performans türü ise, canlı kodlamanın tanımlayıcı ve en çok bilinen performans pratiği olduğu söylenebilecek boş levha (*blank slate*) yöntemidir (N. Collins ve McLean, 2014; McLean, 2011). Boş sayfa yönteminde müzisyen, performansına tamamen boş bir sayfa ile, hiçbir hazır materyal kullanmaksızın başlar ve performans yeni algoritmaların yazılması; değiştirilmesi ve silinmesi ile şekillenir (Magnusson, 2011a). Bu performans yöntemi içerisinde barındırdığı risk faktörü bakımından oldukça özeldir, zira programa sunulan yeni bir algoritmanın bir teste tabi tutulması ya da hata ayıklama işlemine sokulması mümkün değildir (N. Collins ve diğerleri, 2003). Dolayısıyla icracı her hamlesinde bilgisayara hatalı bir kod tanıtıp bütün sistemin çökmesine sebep olma tehlikesi ile karşı karşıyadır, ve bu riske karşı güvenebileceği tek şey kendi tecrübesi ile yeteneğidir (Artut, 2016). Dolayısıyla canlı kodlama performanslarının

çoğunluğunun önemli miktarda doğaçlama ve risk içerdiğini söylemek mümkündür. Cocker, canlı kodlamanın doğaçlama ve risk ile ilişkisini şu şekilde açıklamıştır:

> Doğaçlama – bir performansa nereye varacağını bilmeksizin hazır bir plan olmaksızın başlama – düsturu bir risk ve belirsizlik hissini, bilinenden bilinmeyene yönelimi kucaklamaktadır. (...) Canlı kodlama performansları ise bu bilmeme anlarını ve bütün deneme yanılma girişimlerini (…) aktif bir biçimde sergilemektedir. Dahası, bir bilmeme vaadinin (ve içerdiği risk ve belirsizliklerin) canlı kodlamanın doğaçlama icrasının bir parçası olduğunu söylemek mümkündür. (2016, s. 108-109)

Canlı kodlamanın doğaçlamaya müsait ve yüksek risk içeren kendine has yapısı, icracının performans içerisindeki konumunu da belirlemektedir. Canlı kodlamada icracı, her şeyi kontrol edebildiği bir merkez noktada durmaktadır ve performansı dinamik tutacak koşulları sağlamakla yükümlüdür. Bunu yaparken ne zaman kontrolü mevcut algoritmalara devredip yeni algoritmalar tasarlayacağını ve ne zaman kontrolü ele alıp mevcut algoritmalara müdahale edeceğini sürekli olarak tartmak durumundadır. Canlı kodlamada kodun çalışma esnasında değiştirilebilmesi icracıya yeni imkanlar vermekle birlikte yeni sorumluluklar da yüklemektedir. Yazılan algoritmalar yürürlük kazandıktan sonra performansın geneli içindeki rolleri sürekli olarak tartılmalı, işlevsel olmayı kestikleri anda müdahalede bulunulmalıdır (Cocker, 2016). Canlı kodlamanın büyük oranda doğaçlamaya dayalı olması ise bahsedilen görevleri daha da karmaşıklaştırmaktadır. Zira doğaçlama esnasında bir hata ile karşılaşıldığında, tekrar deneme imkanı bulunmamaktadır. Dolayısıyla doğaçlama performansta hatalar da performansın bir parçası haline gelmektedir. Canlı kodlama performansında bu durum, Turkle ve Papert'in (1990) *bricolage* programlama yaklaşımı ile benzerlik göstermektedir (McLean, 2011). Turkle'a göre *bricoleur*, her fırça darbesinden sonra tuvalden bir adım uzaklaşarak sıradaki hamlesini düşünen bir ressama benzemektedir. Bricoleur'ler ilişkiler ve etkileşimler konusundaki uzmanlıkları ile plancılardan ayrılırlar. Hatalar plancılar için yanlış adımlardır, ancak bricoleur'ler için süreç içi düzeltmelerdir. Bir programı çalıştırmak bir plancı tarafından "söyleyeceğini söylemek" gibi algılansa da bir bricoleur için bir monologdan çok söyleşidir (Turkle ve Papert, 1990). Bir canlı kodlama müzisyeninin de performans esnasında bilgisayar ile ilişkisi bu şekilde açıklanabilir.

Bahsedilen tüm özellikleriyle bu özgün müzisyenlik türü doğaçlama, süreç hakimiyeti ve müzikal işlemlerin kod ile temsili konularında yüksek beceri gerektirmektedir (Brown, 2016). İcracıların sahip olması gereken yüksek beceri, canlı kodlama alanında virtüoziteden bahsedilmesini mümkün kılmaktadır. Hatta canlı kodlamanın elektronik müziğin birçok dalındaki virtüozite sorununa bir çözüm teşkil edebileceği bile öne sürülmüştür (Flašar, 2016; Nilson, 2007; A. Sorensen, 2005; Wang ve

Cook, 2004). Canlı kodlama virtüözitesi iki ayrı başlık altında incelenebilir: kodun bilgisayara girilmesi noktasındaki (fiziksel) beceri ve müzikal fikirlerin oluşturulup koda dökülmesi noktasındaki (bilişsel) beceri.

Kodun bilgisayara girilmesi noktasındaki beceri, tekrar ve çalışmayla geliştirilebilen bir motor yetisi olması bakımından geleneksel enstrümantal virtüöziteyle benzerlik gösterse de, canlı kodlama virtüözitesinin küçük bir kısmını oluşturur. İcracının kod yazma hızı, performansın akıcılığını artırmak ve belirli işlemleri müziğin belirli noktalarında devreye girecek şekilde yetiştirmek gibi konularda önemli faydalar sağlamaktadır. Ancak kodun içeriği, duyulacak materyali belirlemesi itibariyle kodun yazım hızından çok daha önemlidir. Dolayısıyla canlı kodlama virtüözitesine değinilen çalışmalarda genellikle bilişsel sürece yönelik beceriden bahsedilmiştir (A. R. Brown, 2016; Knotts, 2016; Wang ve Cook, 2004; Ward ve diğerleri, 2004; Zmölnig ve Eckel, 2007).

(Rohrhuber ve de Campo, 2009), canlı kodlamanın bilişsel sürecini estetik düşüncenin kod ile ifade edilmesi, sonucun algılanması ve tasarlanan ses ile duyulan ses arasındaki farkın ayırt edilmesi noktalarından müteşekkil bir döngü olarak tanımlamıştır. Böyle bir tanımdan hareketle bir canlı kodlama performansının neredeyse tamamen bu bilişsel süreçten oluştuğunu söylemek mümkündür. Sayer canlı kodlama ile geleneksel enstrümanistlik arasındaki virtüözite farkını açıklarken canlı kodlama sanatçısının enstrümanistin aksine bilişsel süreçlere daha çok, fiziksel süreçlere daha az yaslandığını belirtmiştir ve aradaki farkı açıklarken nesne hafızası (*object memory*) ve işlem hafızası (*process memory*) kavramlarını kullanmıştır (2015). Sayer'ın çalışmasına göre bir müzisyen, enstrümanı üzerinde çalışırken sonradan kullanmak üzere belirli "eylem birimlerini" hafızasına kodlamaktadır. Kodlanan eylem birimleri performans esnasında çeşitli farklı parametrelerle çağırılıp işletilebilmektedir. Bir gam üzerinde yeterince çalışmış bir piyanistin bu gamı konser esnasında farklı hızlarda çalabilmesi, eylem birimlerinin farklı parametrelerle çağrılmasına örnek olarak gösterilebilir. Eylem birimleri, performans esnasındaki bilişsel yükü azaltmak amacıyla nesne hafızasına depolanır. Eylem birimlerinin çeşitli dizilişler ve parametrelerle kullanılmasına yönelik bilgi ise, işlem hafızasında bulunur. Canlı kodlama ve enstrümanistlik, bu hafıza türlerinin önemi bakımından büyük farklılıklar göstermektedir. Nesne hafızası, bir enstrümanist için oldukça önemli olan "teknik" kavramına işaret ederken, bir canlı kodlama müzisyeni için o kadar da hayati olmayan klavye kullanımına işaret etmektedir. Canlı kodlamanın neredeyse tamamen işlem hafızasına dayanması, canlı kodlama ve enstrümanistliğin farklı bilişsel yüklere sahip olmasına sebep olmaktadır. Nesne

hafızasının enstrümanistlikte önemli bir yere sahip olması, enstrümanistlerin zaman zaman kendilerini "otomatik olarak, düşünmeksizin" çalarken bulmaları sonucunu doğurabilmektedir. Bunun sebebi zaman içerisinde nesne hafızasının belirli bağlantıları kurmaya alışması ve bu bağlantıları otomatik olarak kurabilmesidir. Canlı kodlamada böyle bir duruma rastlanmamaktadır. Zira canlı kodlamanın nesne hafızası dağarcığı işlem hafızasından bir komut gelmeksizin sonuç üretecek kadar etkin değildir. Bu nedenle bir canlı kodlama performansında üretilen hemen her ses bilinçli bir tercihin sonucu olmak zorundadır. Müzisyenin tekrara düşmesi ya da belirli alışkanlıklara dönmesi mümkün olsa bile bu durum bilinç dışında gerçekleşemez. Dolayısıyla canlı kodlamanın geleneksel enstrümantal performanslardan daha fazla bilişsel yüke sebep olduğunu, ve bu nedenle daha fazla bilişsel kaynak gerektirdiğini söylemek mümkündür (Sayer, 2015, 2016).

Canlı kodlama performans türlerinden bahsedilirken görsel unsurlara da değinilmesi oldukça önemlidir. Neredeyse tüm canlı kodlama performans türlerinde karşılaşılan görsel unsurlar, pek çok araştırmacı ve canlı kodlama müzisyeni tarafından tartışılan bir konu olagelmiştir. Bu tartışmalar iki ana konuda toplanabilir.

Tartışma konularından ilki seyircinin müzik ile müzisyen arasında kurduğu ilişkidir. Bu konudaki tartışmalarla ilgili olarak Cascone, akusmatik müziğin – sunum yöntemi açısından – dizüstü bilgisayar müziğinin bir öncülü olduğunu öne sürmüş ve dizüstü bilgisayar müzisyenlerinin bu sunum yöntemi ile birlikte akusmatik müziğe içkin bazı sorunları da iktisap ettiğini iddia etmiştir. Bu sorunları ise elektronik müziğin alışılmadık performans yöntemleri sebebiyle seyircinin icracıya bir "mevcudiyet" ve "sahicilik" atfetmemesi, icracının bir müzisyenden çok teknisyen olarak addedilmesi gibi örneklerle açıklamıştır. Cascone'a göre dizüstü bilgisayar müziğinin yegâne görsel elementi olan icracının elleri ve bileklerinde oluşan mikro-hareketler, alışılageldik görsel kodlardan oldukça uzaktır ve dolayısıyla ana akım seyircinin beklentilerini karşılamamakta; akusmatik müzikle benzer tepkilere yol açmaktadır (Cascone, 2003). Flasar ise akusmatik müziğin görselliği kasten devre dışı bırakarak müzikal performansların görsel veya teatral bileşenlerinin önemini kanıtladığını ifade etmiş, bu görsel materyal ihtiyacının en muhtemel açıklamasının ise insanların algılanan sesin kaynağına odaklanma eğilimi olduğunu öne sürmüştür (Flašar, 2016). Ancak canlı kodlamadaki görsel unsurların işlevine ilişkin tartışmalar, yalnızca müzisyen ve müzik ilişkisine yönelik değildir.

İkinci tartışma konusu ise seyircinin müziğin görsel ve duysal ögeleri arasında kurduğu ilişkiye yöneliktir. Geleneksel müzik performansı yöntemlerinde icranın görsel boyutu ile duysal boyutu arasında doğrudan bir ilişki mevcuttur. Görülenler,

duyulanların nasıl meydana geldiğini destekler ve açıklar niteliktedir. Ancak alışılageldik bir dizüstü bilgisayar müziği performansı bu ilişkiyi sağlamamakta, duyulanın nasıl meydana geldiğine dair görsel bir bilgi sunmamaktadır. Bu bağlamda Aaron ve Judge, bilgisayarı bir kara kutuya benzetmiştir (Aaron ve Judge, 2012). McLean, sahnedeki dizüstü bilgisayar müzisyeninin bilgisayarında çok fazla hareketlilik bulunmasına rağmen bu hareketliliğin görülememesinin performansı eksik kıldığını, icra ve icracı hakkında bir fikre varılmasını zorlaştırdığını ifade etmiştir (McLean, 2003). Magnusson ise kodun duysal ögelerin temsili olduğunu; kod bilinmediği zaman birçok müzikal katmanın yanlış anlaşılabileceğini yazmıştır. Ayrıca kodun eserdeki enstrümanlar, sesler ve eserin formu hakkında detaylı bilgi sunabileceğini öne sürmüştür (Magnusson, 2011a). Puckette bu sorunlardan başka biri olan tekrar edilebilirlik ile ilgili, şu açıklamalarda bulunmuştur:

> Varsayalım ki bir icracı başka bir icracının bilgisayar müziği repertuvarından bir eser çalmak istiyor; ya da bir başka icracının icadı olan bir müzikal fikri ödünç almak istiyor, tıpkı bir caz müzisyeninin başka bir müzisyenden bir "lick" ödünç alması gibi. (…) Bunu mümkün kılmak için kullandığımız kontroller ve duyduğumuz sesler arasında doğrudan bir ilişki bulunmalıdır (Bu durum ayrıca seyirciler açısından da fena olmazdı). Bir tuşa basıp bir sekansı başlatan bir müzisyen bize müziğin nasıl oluşturulduğu hakkında hiçbir bilgi vermemektedir. Bu durumda müzik hakkında öğrenebildiklerimiz kulaklarımızın duyduğuyla sınırlı kalmaktadır. Ancak eğer icracının eylemleri ile duyulan sesler daha yakın bir ilişki içerisinde bulunursa, müziğin fikirsel içeriği hakkında daha fazla bilgi edinebilir ve kendi müziğimizi de bu doğrultuda geliştirebiliriz. Böylece bilgisayar müziği icrasında daha güçlü ve anlamlı bir gelenek oturtulabilir. (1991, s. 6-7)

Canlı kodlama icracılarının görsellik konusundaki tartışılan sorunlara yönelik geliştirdiği çözüm yöntemlerinden biri, ekran yansıtma uygulamasıdır. Müzisyenlerinin ekranlarının – ve dolayısıyla yazdıkları kodun – projektör vasıtasıyla bir duvara herkesin görebileceği şekilde yansıtılmasından meydana gelen uygulama; Alex McLean gibi aktif araştırmacıların yazıları, çeşitli canlı kodlama konserleri ve TOPLAP manifestosu vasıtalarıyla canlı kodlama içerisinde bir gelenek haline gelmiştir (Lawson ve Smith, 2019; McLean, Griffiths, Collins ve Wiggins, 2010; Zmölnig ve Eckel, 2007). Bu gelenek icracının virtüözitesinin anlaşılması (A. R. Brown, 2016) ve müzikal ögelerin daha anlaşılabilir olması (Magnusson, 2011a) gibi konularda takdir kazanmıştır fakat çeşitli eleştirilere de konu olmuştur.

Canlı kodlamada üzerine en çok konuşulmuş ve çalışılmış eleştiri noktalarından biri, programlama bilmeyen dinleyici ile yansıtılan kod arasındaki ilişkidir. İcracının ekranındaki kodun büyükçe yansıtılıp yegâne görsel öge olarak sunulmasının,

programlama ile ilgisi olmayan dinleyicileri ötekileştirebileceği öne sürülmüştür (McLean, 2011; Rohrhuber ve diğerleri, 2007). Ayrıca yansıtılmış kodun seyirciyi müziği dinlemekten çok kodu okumaya teşvik edebileceğinden ve dolayısıyla dikkat dağıtıcı olabileceğinden bahsedilmiştir (N. Collins ve McLean, 2014; McLean, 2008). Slub grubu üyesi Adrian Ward, projeksiyon uygulamasının pasif ve sıkıcı olduğunu; aynı zamanda dinleyici/icracı hiyerarşisini desteklediğini ifade etmiştir (McLean, 2011). Başka bir eleştiri noktası ise bazı canlı kodlama programlarının beyaz arkaplanlı tasarımlarının yansıtıldığı zaman rahatsız edici bir parlaklığa sebep olmasıdır (N. Collins ve McLean, 2014).

Diğer taraftan Ekran yansıtma uygulamasına yapılan eleştirilere karşılık pek çok çalışma ve yayın yapılmıştır. Örneğin TOPLAP manifestosunda "*Canlı kodlamaya yabancı bir izleyici kitlesinin kodu beğenmesi için anlaması gerekmemektedir, tıpkı bir gitar performansını beğenmesi için gitar çalmayı bilmesinin gerekmediği gibi."* maddesine yer verilmiş ("ManifestoDraft—Toplap", t.y.), birçok canlı kodlama yazılımına "karanlık mod" seçeneği eklenmiş, Slub ve Powerbooks Unplugged gibi gruplar tarafından alternatif kod sunum yöntemleri denenmiş, (McLean, 2008; Rohrhuber ve diğerleri, 2007) kodlama bilmeyen seyircilere yönelik alternatif kod tasnif yöntemleri üzerinde çalışılmıştır (Herrera Machuca, Lobato Cardoso, Torres Cerro ve Lomelí Bravo, 2016).

2.1.6 Canlı Kodlamanın Elektronik Müzik Performansı Bağlamındaki Önemi

20. yüzyılda analog sentezleyici teknolojisinden dijital ses kaydına kadar birçok ses ve müzik teknolojisi alanı oldukça hızlı bir gelişim kaydetmiştir. Kaydedilen gelişim, ses ve müzikle doğrudan etkili pek çok alanı da şekillendirmiştir.

Ses ve müzik teknolojisindeki bu gelişmeler elektronik müzik araçlarının birçok müzik türüne dahil olması ile sonuçlanmıştır. Uzun bir süre boyunca yalnızca elektronik müzik içerisinde kullanılagelen araçlar artık farklı popüler müzik eserlerinde de kullanılır hale gelmiştir. Örneğin teyp kaydı ve analog ses sentezi teknolojileri *Musique Concréte* ve *Elektronische Musik* ekollerinde 1940'lardan beri kullanılsa da ilgili ekipmanlar oldukça pahalıdır ve erişimleri zordur. Dolayısıyla bu dönemde eletronik müzik sayılı kişi tarafından icra edilebilen bir müzik türü konumundadır. Ancak 1960'larda hem tape kaydına hem analog ses sentezine yönelik araçların daha kullanışlı ve ucuz hale gelmesiyle elektronik müzik geleneği biri deneysel/akademik besteciliğe diğeri popüler müziğe yönelen iki kola ayrılmıştır (Pejrolo ve Metcalfe, 2017).

1960'lardan itibaren elektronik enstrümanların popüler müzikte kullanımı birçok müzisyenin ve müzik grubunun etkisiyle yaygınlaşmıştır. Örneğin Beatles 1967 yılında çıkardığı ünlü "Strawberry Fields Forever" gibi birçok eserinde *mellotron* ve kaydedilmiş ses gibi elektronik müzik tekniklerinden ve enstrümanlarından yararlanmış, Pink Floyd ve Genesis gibi gruplar eserlerinde sentezleyicilere yer vermiştir. Ayrıca bu dönemde Silver Apples grubu sadece elektronik bazlı sesler kullanarak dans müziği üretmeye başlamış, Gershon Kingsley'nin yazdığı "Popcorn", birçok ülkenin popüler müzik listelerinde yüksek sıralarda yer alarak en ünlü elektronik dans müziği eserlerinden biri haline gelmiştir.

Çeşitli müzisyenlerin çalışmaları ve hızla gelişen teknoloji ile elektronik müzik 1970'ler boyunca gittikçe erişilebilir hale gelmiş ve yaygınlaşmıştır. Bu durum birçok popüler müzik türünün elektronik müzikle daha yakın bir ilişki içine girmesi sonucunu doğurmuştur.

1980'lere girildiğinde dijital teknolojinin gelişmesi ile mevcut elektronik müzik ekipmanları da çeşitlenmiştir. Analog sentezleyiciler polifonik hale gelmiş, FM (*frequency modulation*) sentezi yöntemi kullanan dijital sentezleyiciler geliştirilmiş ve PCM (*pulse code modulation*) tabanlı örnekleyiciler piyasada yerini almıştır. 1980'lerdeki gelişmelerle elektronik enstrümanların popüler müzikteki yeri sağlamlaşmış, üstelik tamamen elektronik müzik teknolojisi etrafında kurulmuş pek çok yeni popüler müzik türü ortaya çıkmıştır.

1990'larda hızla gelişen kişisel bilgisayar teknolojileri pek çok alanda olduğu gibi müzik sektöründe de büyük değişimlere sebep olmuştur. DAW'lar ve eklentiler, bu değişimlerin en önemlilerindendir (Bölüm 2.3'te daha detaylı açıklanacaktır). 21. yüzyıla girildiğinde bir müzik eserinin baştan sona bir kişisel bilgisayarda üretilmesi, normal bir durum haline gelmiştir. Bu durum birçok popüler elektronik müzik türünün sadece bilgisayar yazılımları vasıtasıyla üretilir hale gelmesine sebep olmuştur.

Bir eserin sadece bilgisayar ile var edilebilmesi üretim alanında büyük bir kolaylık sağlasa da, eserin performansının nasıl olacağı noktasında belirsizlik yaratmaktadır. Elektronik müzik eserleri genellikle baştan sonra belirlenmiş, formalize edilmiş yapılardır ve bir popüler elektronik müzik eseri türüne bağlı olarak onlarca veya yüzü aşkın farklı ses kanalından meydana gelebilmektedir. Dolayısıyla bir eser içerisinde pek çok ses kaynağının aynı anda tetiklenmesi gerekebilmektedir.

Elektronik müziğin canlı performansının birçok farklı yöntemi ve türü bulunsa da, bu sorun genellikle iki farklı şekilde çözülebilmektedir. Birinci yöntem seslerin birçok icracı arasında paylaştırılıp geleneksel enstrümanlar ya da *launchpad*, MIDI klavye

gibi kontrol aygıtları vasıtasıyla canlı olarak çalınmasıdır. Bu yöntem hem icracıların organize edilmesi hem de eserin birçok icracıya verimli ve anlamlı şekilde dağıtılması gibi zorluklar barındırmaktadır. İkinci yöntem ise belirlenen bölümlerin icracı veya icracılar tarafından canlı olarak çalınması veya manipüle edilmesi, belirli bölümlerin ise bilgisayardaki hazır kayıt üzerinden oynatılmasıdır. Bir elektronik müzisyen olan Hanne Hukkelberg, bir eserdeki bütün sesleri çalacak sayıda müzisyen bulmanın zorluğu dolayısıyla genellikle bazı kısımları bilgisayardan çalmak zorunda kaldığını belirmiş ve bu durum ile ilgili şu değerlendirmeleri yapmıştır:

> Grubumla bilgisayarın sahnede gösterilebilirliğine ve konserdeki materyalin en fazla ne kadarının bilgisayardan çalınabileceğine dair sonsuz tartışmalar yaparız. Genellikle, arka plandaki ufak vurguların bilgisayardan çalınması kabul edilebilir, ancak eğer ses çok özelse bilgisayardan çalınmamalı. Çünkü o zaman eserin asıl hareketi kaçırılıyor; ve bu bize hilekarlık gibi geliyor. (Kjus ve Danielsen, 2016)

Bilgisayardaki hazır kayıtların performans esnasında oynatılması, elektronik müzisyen ve müziği arasındaki zaten alışılmıştan zayıf olan bağı zedeleyip performansın canlılığı konusunda soru işaretleri yaratabilmektedir. Buna ek olarak, hazır kayıtların yol açtığı başka sorunlardan da bahsedilebilir.

Konser esnasında kaydedilmiş bir sesin çalınıyor olması, en azından sesin duyulduğu süre boyunca müzisyenleri büyük ölçüde kısıtlamaktadır. Zira kaydedilmiş sesin temposu, uzunluğu ve ritmik yapısı bellidir, performans esnasında değiştirilemez. Dolayısıyla hızlanma, yavaşlama ve hatta eserin belirli bir kısmına geri dönme gibi müzisyenlerin sıklıkla başvurduğu performans araçları kullanılamaz hale gelmektedir. Bunlara ek olarak eser içerisinde doğaçlamaya başvurmak da oldukça zorlaşmaktadır (Kjus ve Danielsen, 2016). Kaydedilmiş ses kullanımının bahsedilen kısıtlılıkları, müzisyenleri canlı performans esnasında *sentezleyici* ve tetiklenmiş sesler kullanarak tüm kontrolü elde tutmak ile kaydedilmiş sesler kullanarak tüm kontrolü bilgisayara devretmek seçenekleri arasında bir tercih yapmaya zorlamaktadır (Dannenberg ve diğerleri, 2014).

Elektronik müziğin canlı performansı ile ilgili bir başka konu ise, doğaçlamadır. Elektronik müzik ilk bakışta oldukça soyut görünse de genellikle müzisyenlerin tuşlara basmak, anahtarları çevirmek ve belirli kabloları bağlamak gibi müdahaleleri sonucu devrelerde hareket eden elektronlardan meydana gelmektedir. Sesin nedenselliğinin performansın bu denli içerisinde olması, gerçek zamanlı doğaçlamaya zemin hazırlamaktadır (Julian ve de Campo, 2009b). Ancak elektronik müzik doğaçlaması konusunda da bazı zorluklar mevcuttur. Bu zorluklardan bir

tanesi, elektronik müzik enstrümanlarının tasarımı ve kontrolü için gereken parametrelerin fazlalığı olarak teşhis edilebilir.

Elektronik müzik enstrümanları alışılageldik enstrümanlardan oldukça farklıdır zira teorik olarak elektronik müzik enstrümanları ile elde edilebilecek seslerin sınırı yoktur. Bu sınırsızlık elektronik müzisyenlerin doğru sesleri ve enstrümanları bulmak için uzun süreler boyunca aktif olarak çalışmasını gerektirmektedir (Mazierska, 2018). Elektronik müziğin kendine has yapısının bir analoji ile açıklanması gerekirse, elektronik müzisyenler orkestrasındaki bütün enstrümanları baştan tasarlayan bestecilere benzetilebilir. Bu tür olanaklar elektronik müziğin tasarımında fayda sağlasa da doğaçlamayı zorlaştırmaktadır çünkü yeni bir enstrümanın gerçek zamanlı olarak tanımlanması oldukça zordur. Bir müzisyenin aklında istediği sese dair tüm parametrelerin olduğu varsayılsa bile, bu parametrelerin ilgili yazılımlara teker teker girilmesi ve sonradan bu enstrümanların kullanılması genellikle canlı performans sırasında sahip olunamayacak kadar fazla zaman gerektirebilmektedir. Ancak elektronik müzik ve doğaçlama arasındaki mesafenin sebebi sadece enstrümanların detaylılığı değildir.

Elektronik müzikte doğaçlama ile ilgili bir diğer güçlük ise, müziğe dahil olan ögelerden ileri gelmektedir. Geleneksel anlamda bakıldığında müzik melodik hat, ritm, form, armoni ve tını gibi birçok ögeden oluşabilir. Çeşitli müzik türlerinde bu ögelerden bazıları oldukça önemsenirken bazıları ikinci plana atılabilir, ya da müziğe hiç dahil edilmeyebilr. Elektronik müzisyenlerin seslerin birbirleriyle bağlantılarından çok tekil tınıları ile ilgilendiği, dolayısıyla elektronik müzikte tınının oldukça ön planda olduğu ifade edilmiştir (Kaiser, 2013). Bu tespit genel anlamda doğru olsa da elektronik müziğin akademik/deneysel ve popüler olarak sınıflandırılmış iki dalında farklı şekillerde ortaya çıkmaktadır.

Akademik/deneysel elektronik müzik olarak ifade edilebilecek dalda genellikle en ön plandaki ögenin tını olduğunu söylemek mümkündür (Kaiser, 2013). Bu durumun sonucu olarak akademik/deneysel elektronik müzik tasarımcıları; melodi, ritm, armoni gibi özelliklerini görmezden gelerek – ve hatta zaman zaman reddederek – canlı performanslarını sadece tını üzerine kurabilmekte, doğaçlamalarını yalnızca tını manipülasyonları üzerinden şekillendirebilmektedir. Ancak popüler elektronik müzikte durum daha farklıdır. Popüler elektronik müzik başlığı altında birbirine benzemeyen birçok müzik türünden bahsedilebilmektedir ancak bu türlerin büyük çoğunluğunun ortak özellliği; ritm, melodi ve armoni gibi popüler müzikte sıkça rastlanılan ögelere kuvvetle yer vermeleridir. Tını, popüler elektronik müzikte de oldukça önemli bir yerde durmaktadır ancak müziğin yegane tanımlayıcı unsuru

değildir. Bu nedenle birçok unsurun aynı anda gözetilmesi gereksinimi, popüler elektronik müziğin doğaçlamasını zorlaştırmaktadır. Zira müzisyenin – yukarıda bahsedilen zorlukların yanısıra – doğaçlama esnasında müziğin ritim ve melodi gibi birçok ögesine ilişkin anlamlı değişiklikler yapması, ya da en azından birçok ögenin aksamadan ilerleyebilmesini sağlaması gerekmektedir. Canlı kodlama, bu noktada devreye girmektedir.

Canlı kodlama aynı anda birçok sesin tetiklenmesi, kaydedilmiş materyal kullanımı, ve birçok unsurun aynı anda gözetilmesi gibi zorlukları tamamen ortadan kaldırmasa da önemli bir çözüm yaklaşımı teşkil etmektedir. Öncelikle elektronik müziğin formalize yapısı canlı kodlama ile oldukça uyumludur çünkü kodlama, belirli komutların uygulanmak üzere bilgisayara girilmesinden ibarettir. Bilgisayar işlemcisinin yüksek hızından ve aynı anda birden fazla işlem gerçekleştirebilen iş parçacıklarından (*thread*) faydalanan canlı kodlama ile aynı anda birçok sesin tetiklenmesi gereksinimi büyük oranda cevaplanmaktadır. Canlı kodlamada sesleri tetiklemesi istenen müzisyenlerin rolünü işlemci iş parçacıkları alabilmektedir. Dolayısıyla canlı kodlama vasıtasıyla müzisyen ile birlikte çalışan yüzlerce başka icracının varlığı simüle edilebilir. Ayrıca bu "simülasyon icracılar" performans esnasında farklı şekillerde görevlendirilebilmekte ve tek bir elden yönetilebilmektedir. Öte yandan canlı kodlama yönteminde oynatılan kayıtların manipüle edilmesi ve performansa uygun olarak değiştirilmesiyle hazır kayıtların oynatılmasının getirdiği sorunlar da bir miktar çözülebilmektedir. Hatta kayıtların uygun olarak organize edilmesiyle istenilen anda eserin herhangi bir noktasına dönülebilir, istenilen kısım tekrarlanabilir, eser hızlandırılıp yavaşlatılabilir.

Elektronik müzikte doğaçlama ile ilgili bahsedilen ilk sorun olan kullanılan seslerin tasarım ve kontrol detayından ileri gelen performans güçlüğü, canlı kodlama ile tam bir çözüme kavuşturulabilmiş değildir. Bu sorunun üstesinden gelinmesi müzisyenin bilişsel becerilerine bırakılmıştır. Çünkü canlı kodlama ile kullanılan sesler çeşitli yöntemlerle performans esnasında manipüle edilebilmekte, şekillendirilebilmektedir. Öte yandan canlı kodlamanın algoritmaları ve yazılı arayüzü, mevcut standart olan görsel arayüzler ve fare imleci kontrollerine önemli bir alternatif teşkil etmektedir.

Görsel arayüzler ve fare imleci elektronik müziğin stüdyo içerisindeki basit ve sade kullanımı için tasarlanmıştır ve herhangi bir beceriye dayanmamaktadır; dolayısıyla canlı performansa ve özellikle doğaçlamaya uygun değildir. Fakat canlı kodlama ile aynı kontroller çok daha hızlı sayılabilecek yazılı arayüz (Nash ve Blackwell, 2011) ve oldukça derinleştirilebilecek algoritmik tasarım ile sağlanmıştır. Bu kontrol şeması elektronik müziğin içkin "çok fazla parametre" sorununa bir çözüm değildir. Ancak

canlı kodlama ile müzisyene bu sorunun üstesinden gelebilmek için geliştirebileceği özel bir tür virtüözite sunulmaktadır. Dolayısıyla istediği sesin bütün parametrelerini öngörebilen bir müzisyen yeterince hızlı klavye kullanabildiği ve yeterince işlevsel bir algoritmik tasarım kurabildiği sürece sesi performans esnasında yaratabilecek ve manipüle edebilecektir. Bu çeşit bir virtüözite, doğaçlama esnasında aynı anda birden fazla ögenin kontrol edilmesi güçlüğünün de üstesinden gelmekte kullanılabilecektir.

Bilgisayarın aynı anda birçok işi kolaylıkla yapabilmesi, müzisyenin de aynı anda birçok öge ile ilgilenebilmesini mümkün kılmaktadır. Dolayısıyla bir müzisyenin doğru algoritmaları tanımlaması ile eserin ritmik yapısını baştan sona kurabilmesi, veya performansın ilerleyen bir kısmındaki önemli bir müzikal hareketi eserin herhangi bir noktasında belirlemesi mümkün olmaktadır. Melodi ve armoni ögeleri için de aynı imkân mevcuttur. Böylece icracının kurduğu algoritmalarla birden fazla müzikal ögeyi aynı anda idare etmesi kolaylaşmaktadır.

Canlı kodlamanın sunduğu bu avantajlar, elektronik müzik performansı için oldukça işlevsel bir araç olmasını sağlamaktadır. Ne var ki canlı kodlamanın bir araç olarak yaygınlaşabilmesi için, işlevsel olmasının yanında kolaylıkla kullanılabilir olması da gerekmektedir. Bu kapsamda kolaylıkla erişilebilen, öğrenilebilen ve çok gelişmiş sistem olanakları gerektirmeyen diller büyük önem taşımaktadır. Bunlardan biri de Sonic-Pi’dir.

2.2 Sonic-Pi Yazılımı

2.2.1 Sonic-Pi'nin Geliştirilme Süreci ve Teknik Özellikleri

Sonic-Pi, basitliği ve eğitici materyal miktarı bakımından en öğrenilebilir canlı kodlama yazılımlarındandır. Ruby dilinde geliştirilen yazılım ve dil, aslen Raspberry Pi bilgisayarının programlama eğitimindeki işlevselliğini geliştirmek amacıyla geliştirilen bir projedir.

Raspberry Pi, ilk sürümü 2012 yılında İngiliz Raspberry Pi Vakfı tarafından geliştirilmiş bir tek kartlı bilgisayardır. Kar amacı gütmeyen bir kuruluş olan Raspberry Pi Vakfı bu proje ile çocukları ve gençleri bilgisayar teknolojilerini deneyimleyerek ve kurcalayarak öğrenmeye teşvik etmeyi amaçlamıştır ("Raspberry Pi Foundation—About Us", t.y.). Raspberry Pi, kredi kartı ebatlarında çıplak bir devre kartı olarak satışa sunulmaktadır. Farklı sürümleri farklı fiyatlar üzerinden satılsa da fiyat aralığı 35 – 75 dolardır (Upton, 2020).

Raspberry Pi bilgisayarının düşük fiyatı, ufak boyutu ve üretim amacı; pek çok eğitim kurumunun ilgisini çekmiştir ve Raspberry Pi'ın teknoloji eğitiminde yer kazanması için pek çok çalışmaya imza atılmıştır. 2013 yılında ortaya çıkan Sonic-Pi projesi de bu çalışmalardan biridir. Proje, Raspberry Pi Vakfı'nın bilgisayar biliminin temel prensiplerinin öğretiminde okullarda kullanılabilecek yeni bir programlama dilinin geliştirilmesi daveti üzerine başlamıştır. Proje müziğin halihazırda tempo, melodi ve armoni gibi soyut matematiksel fonksiyonlarla ifade edilebilen ögelere sahip olduğunu ve dolayısıyla basit dijital müzik eserlerinin temel bir programlama becerisi ile kolaylıkla üretilebileceğini öne sürmüş; bu tür deneyim odaklı bir yöntemin erken programlama eğitiminde başarılı olabileceğini iddia etmiştir (Aaron ve Blackwell, 2013).

Sonic-Pi projesinin kuruluş amacı hem öğrencilerin ilgisini cezbedecek müzikal bir deneyim sunacak, hem de çok sayıda temel hesaplama kavramını tanıtacak minimal bir DSL ve tümleşik geliştirme ortamı (*integrated development environment - IDE*) yaratmaktır. Sonic-Pi programı ile beş saatlik ders sürecinin sonunda hiçbir programlama geçmişi olmayan öğrencilerin tatmin edici müzikal bir eser üretebilen bir programı tamamlayacak seviyeye gelmeleri amaçlanmıştır. Dilin sentaktik ve semantik unsurları da, bu amaçlar göz önünde bulundurularak, hiçbir programlama geçmişi olmayan 12 yaşında öğrenciler hedeflenerek tasarlanmıştır (Aaron ve Blackwell, 2013).

Sonic-Pi'nin Ruby dili çevresinde oluşturulmasının iki sebebi vardır. Birinci sebep, projenin kabulü ile yazılımın ilk sürümünün sunumu arasındaki sürenin yalnızca üç

hafta olması dolayısıyla programın yaratıcısı Sam Aaron'ın uygun bir dil arama adımını atlayarak bildiği bir dille çalışmaya başlamak zorunda kalmasıdır. İkinci sebep ise programın daha çok insana ulaşmasına yöneliktir. Sonic-Pi projesinin kabul edildiği 2013 yılında İngiltere'de programlama eğitimi alanında Python kullanımı oldukça yaygındır. Ruby de semantik ve sentaktik açılardan Python ile benzerlik gösteren bir programlama dilidir. Dolayısıyla eğitim amaçlı bir kullanımda Ruby'nin geniş çevrelerce kabul edilme olasılığının yüksek olacağı düşünülmüştür (Aaron ve Blackwell, 2013).

Sonic-Pi dili basit ve gömülü bir Ruby DSL'i olarak tasarlanmıştır. Dilin tasarımının temel taşı belirli yardımcı fonksiyonlar vasıtasıyla Ruby'nin fonksiyonalitesini ihtiyaç doğrultusunda genişleten SpiderLang nesnesidir. SpiderLang nesnesi, bulundurduğu spider_eval isimli fonksiyonun aldığı metin (*string*) argümanı vasıtasıyla programa girilen kodun gelişmiş SpiderLang fonksiyonalitesi ile işlenmesini sağlar. Dolayısıyla Sonic-Pi dilinin asıl giriş noktası (*entry point*), şu Ruby ifadesidir:

```
SpiderLang.new.spider_eval(sonic_pi_kodu)
```

Sonic-Pi, SuperCollider'ın güçlü ses motoru SuperCollider Server üzerine kurulan yüksek dereceli arayüzlerle meydana getirilmiş dillerden bir tanesidir. SpiderLang nesnesi, ilişkilendirildiği SuperCollider Server'ına ilkleyici (*initializer*) fonksiyonu ile bağlanır ve Server'ı temizler. Böylece sistem kullanıcıdan ses komutları almaya hazırlanmış olur (Aaron ve Blackwell, 2013).

Sonic Pi'nin temel "çalma" ve "susma" komutları, "*play*" ve "*sleep*" fonksiyonlarıdır. Play fonksiyonu parametre olarak çalınması istenen notanın midi değerini alırken, sleep fonksiyonu sistemin kaç vuruş boyunca susması gerektiğini belirten sayısal bir değer alır. Dolayısıyla birer vuruşluk suslarla bir do majör arpeji yapılması istendiğinde, aşağıdaki kod yazılmalıdır:

```
play 60
sleep 1
play 64
sleep 1
play 67
```

Görüldüğü üzere girilen kod, midi nota numaraları dışında (sonraki geliştirmelerle notaların midi numaralarıyla girilme zorunluluğu ortadan kalkmıştır) insan diline makine dilinden daha fazla benzemektedir. Bunun sebebi, Ruby'nin ayrıştırıcısının

(*parser*) esnekliğidir. Böylelikle parametreleri çevreleyen parantezler ve komutların sonuna eklenen noktalı virgüller olmaksızın kod yazmak mümkün hale gelmektedir. Bu durumun programın çocuklar – ve programlamadan uzak kimseler – üzerindeki cazibesini önemli ölçüde etkilediği düşünülebilir. Çünkü aşağıdaki kod parçası sentaktik olarak daha eksiksiz olsa da hem görünüş olarak daha az ilgi çekicidir; hem de bahsedilen noktalama işaretlerinin kullanılması bu işaretlerin kullanım sebebine dair soruları ve cevaben sıkıcı teorik açıklamaları beraberinde getirecektir.

```
play (60);
sleep (1);
play (64);
sleep (1);
play (67);
```

Sonic-Pi IDE (*Integrated Development Environment*), kullanıcının yazdığı kodu sisteme girmesini sağlayan araçtır. IDE de, Sonic-Pi'nin diğer özellikleri gibi, mümkün olduğunca sade bir kullanıma uygun olacak şekilde tasarlanmıştır. Dolayısıyla çoğu programlama editöründe ve IDE'sinde bulunan karmaşık kontrollere ve araçlara yer verilmemiştir. IDE ilk sürümünde editör paneli, bilgi paneli, hata paneli, kontrol tuşları ve sekmeler olmak üzere yalnızca beş ögeden oluşan oldukça sade bir yapıya sahiptir.

Sonraki sürümlerde eklenen kılavuz (*tutorial)* paneli, programın kullanılabilirliği ve öğrenme kolaylığı açısından oldukça önemlidir. Bu paneldeki çeşitli sekmeler vasıtasıyla kullanıcı program ile ilgili genel bilgiye, elli dokuz başlıktan oluşan bir eğitim metnine, zorluk seviyelerine göre sıralanmış kod örneklerine ve programın içerisindeki tüm komutların açıklamalı bir listesine erişebilmektedir. Üçüncü sürüm itibariyle bahsedilen sade tasarım büyük ölçüde korunsa da programın bir enstrüman olarak popülerliğinin artması çeşitli ses görselleştirme seçeneklerinin (lissajous eğrileri, spektroskop, dalga formu görüntüsü) ve 2.1.2'de söz edilen "karanlık mod"un eklenmesini gerektirmiştir.

2.2.2 Sonic-Pi'nin Gelişimi ve Önemi

Sonic-Pi, ilk sürümünün tamamlandığı 2013 yılından bu yana istikrarlı bir gelişim göstermiştir. Sonic-Pi'nin yaratıcısı Sam Aaron, 2016 yılında müzik ve teknoloji festivali Moogfest'te sahne almıştır. Popüler müzik dergisi Rolling Stone'un bu festivali inceleyen yazısında Aaron'ın performansından ve Sonic-Pi'den özellikle bahsedilmiştir. Yazıda performansın "günümüzün ötesine geçtiği" ve "bir

performanstan çok herkesi kendi macerasını kodlamaya çağıran bir davet gibi göründüğü" ifade edilmiştir (Weingarten, 2016). Aynı yıl içerisinde Sonic-Pi'nin kılavuz panelindeki eğitici metne destek olması ve programın öğrenilmesini kolaylaştırması amacıyla bir kitap yazılmıştır (Aaron, 2016). 2017 yılında Sonic-Pi kullanıcılarının iletişime geçmesi ve yardımlaşması amacıyla in_thread forumu kurulmuştur ("In_thread", t.y.). Aynı yılda internet üzerinden ücretsiz dersler sunan Mehackit platformunun Sonic-Pi dersi, Finlandiya eğitim standardı Kokoa ile tescillenmiştir ("Mehackit Creative programming workshop with Sonic Pi", t.y.). Yine 2017 yılında Google, açık kaynaklı projeleri ödüllendirdiği *Open Source Peer Bonus Program*'in kazananları arasında Sonic-Pi'ye de yer vermiştir ("The latest round of Google Open Source Peer Bonus winners", t.y.). Bu gelişmelerin Sonic,Pi'nin popülerliğine büyük katkı sağladığı söylenebilir.

Şu ana kadar bahsedilmiş canlı kodlama araçları arasında ana akımda en popüler olanın Sonic-Pi olduğu savunulabilir. Bu iddianın bir ölçümle kanıtlanması – özellikle canlı kodlamanın pek yüksek sayılamayacak görünürlüğü de hesaba katılırsa – oldukça zordur ve bu alanda herhangi bir çalışmaya rastlanmamıştır. Ancak ilişkili sosyal medya hesaplarının takipçi sayısı, tartışma forumlarının aktifliği, internet haberlerinde bahsedilme sıklığı gibi – kanıt niteliği taşımasa da bir fikir verebilen – parametreler dikkate alındığında Sonic-Pi'nin diğer canlı kodlama araçlarına oranla daha görünür olduğu söylenebilir. Bu durum Sam Aaron'ın Sonic-Pi'nin tanıtımı için aktif bir biçimde çalışmasının yanısıra (diğer canlı kodlama dillerinde buna benzer çalışmalara nadiren rastlanmaktadır) Sonic-Pi'nin özel yaratılış amacı ile de açıklanabilir.

Sonic-Pi'nin 12 yaşındaki öğrencilere programlama öğretmek amacıyla tasarlanmış olması, programlama bilmeyen seyirci kitlesi için nispeten daha cazip olmasını beraberinde getirmektedir. Kullanılan dilin sadeliği ve insan diline yakınlığı, programlama bilmeyen insanların bile koda bakarak isabetli çıkarımlar yapabilmesine olanak sağlamaktadır. "Play" (çal), "sleep" (uyu), "10 times do" (10 defa yap), "loop" (döngü) gibi fonksiyonlar kullanım amaçlarını betimleyecek şekilde isimlendirilmiştir. Böylece programlama bilmeyen ancak canlı kodlama ile ilgilenen kişilerin koddaki dilsel ipuçları ile müzikteki duyulan değişiklikleri eşleştirebilmeleri kolaylaştırılmıştır. Bu durumun iki sonucu öngörülebilir. Öncelikle daha fazla sayıda seyirci – yansıtılan ekranda gördükleriyle ilgilenmeleri durumunda – performansın içeriğine dair fikir sahibi olabilmektedir. Böylece, Bölüm 2.1.2'de de tartışıldığı üzere, seyircinin performans ile daha yüksek seviyede etkileşim kurması sağlanmaktadır. Öte yandan, bir canlı kodlama performansı ile bu şekilde etkileşime geçebilen bir

seyircinin canlı kodlamaya daha az mesafeli duracağı savunulabilir. Seyirci için tamamen anlaşılmaz olan ve makine diline daha yakın sentaksa sahip programlama dilleri, seyircinin programlama ile arasındaki mesafenin pekiştirildiğini hissetmesine sebep olabilir. Bu mesafe, seyirci/programcı arasındaki ayrımı derinleştirerek canlı kodlamanın yaygınlaşmasına engel olmaktadır. Ancak daha basit (insan diline daha yakın) sentaksa sahip diller programlamanın o kadar da zor ve yabancı bir eylem olmadığı, dahası, bir miktar çalışma ile herkes tarafından yapılabileceği düşüncesinin yayılmasına sebep olabilir.

Sözü edilmiş diğer canlı kodlama dillerinin büyük çoğunluğu programcılar tarafından, programcılar için geliştirilmiştir. Dillerin tanıtıldığı yayınlarda bahsedilen amaçlara ve özelliklere bakıldığında bu çıkarımı yapmak mümkün olmaktadır (Aaron ve Blackwell, 2013; Magnusson, 2011b; McCartney, 1996; McLean, 2014; A. Sorensen, 2005; A. C. Sorensen, 2018; Wang ve Cook, 2003). Programcılara yönelik geliştirilen karmaşık dillerin çoğunluğu belirli bir teknik yahut tasarımsal sıkıntıyı çözmek, ya da yeni bir tasarım fikrini orya koymak amacıyla meydana getirilmekte; genellikle işlevsellik ve derinlik ön planda tutularak geliştirilmektedir. Bu tanıma uyan Overtone dili, Sonic-Pi ile aynı yayında yer almasına ve aynı araştırmacı tarafından geliştirilmesine rağmen, Sonic-Pi kadar popüler hale gelmemiştir.

Karmaşık dillerin programcılar için geliştirildiğine dair bir başka gösterge de, sunulan öğretici kaynaklardır. Canlı kodlama konusunda kurumsal bir eğitimin mevcut olmaması, canlı kodlamaya yeni başlayacak kişilerin sunulan öğretici materyale bağımlı olması sonucunu doğurmaktadır. Bahsedilen karmaşık dillerin çoğunluğu öğretici materyal değil, sadece dil dokümantasyonları sunmaktadır. Dil dokümantasyonları ise genellikle okuyucunun bir programlama altyapısına sahip olduğu varsayımıyla yazılmaktadır. Programlama bilmeyen insanlara yönelik öğretici yazılar sunan komplike diller de vardır, ancak yine de bu dillerin gelişmiş yapıları sıfırdan öğrenmeyi zorlaştırmaktadır. Sonic-Pi bu noktada da diğer dillerden ayrılmaktadır.

Sonic-Pi, sentaksının basitliğinin yanısıra öğretici materyal miktarı ile de diğer dillerden daha kolay öğrenilebilir durumdadır. Programın içerisinde her an erişilebilen kılavuz ve açıklamalı dokümantasyonun yanı sıra bu dokümantasyonu tamamlaması amacıyla yazılmış ücretsiz olarak sunulan bir kitap ve sertifikalı bir ücretsiz internet dersi mevcuttur. Ayrıca kullanıcılar bu kaynaklarda çözüm bulamadıkları sorularını in_thread forumunda paylaşıp yardım alabilmektedirler.

Bahsedilen erişilebilirlik ve kolaylıklar, Sonic-Pi'yi canlı kodlamanın yaygınlaşmasında oldukça önemli bir konuma getirmektedir. Program ve dil, basitliği sebebiyle teknik ve pratik bazı dezavantajlar barındırsa da belirli kullanım senaryolarında oldukça işlevseldir. Öte yandan canlı kodlamanın programcı olmayan insanlara açılabilmesi için Sonic-Pi gibi belirli başlangıç dillerinin mevcudiyeti de oldukça önemlidir.

Sonic-Pi'nin, tüm bu avantajların yanı sıra, elektronik müzisyenlere hitap edebilmesi için elektronik müzik araçlarına da erişim sağlayabilmesi gerekmektedir. Ses eklentileri bu araçların en önemlilerindendir. Ancak Sonic-Pi, mevcut versiyonunda ses eklentilerini çalıştırma ve yönetme özelliklerine sahip değildir. Bu sebeple Sonic-Pi'nin elektronik müzisyenler arasında bir alternatif olarak kabul edilebilmesi için, ses eklentilerini destekleyebiliyor durumda olması oldukça önemlidir.

2.3 Ses Eklentileri

Dijital ses kaydı teknolojisinin ticarileşmesi, 1970'lerin başlarında Denon ve Decca gibi şirketlerin çalışmalarıyla gerçekleşmiştir. 1970'lerde dijital sesin erişilebilirliği oldukça düşüktür, ancak bilgisayar teknolojisinin hızlı gelişimi ve sabit disk fiyatlarının düşmesi ile 1980'lere girildiğinde dijital kayıt teknolojileri pek çok kayıt stüdyosunda rastlanır hale gelmiştir. 1978 yılında Soundstream Digital Editing System gibi dijital ses editörleri geliştirilmiştir ancak bu editörler yalnızca dijital sesin düzenlenmesi alanında kullanılabilmektedir (Burgess, 2014). Sektördeki müzisyenler ise çoğunlukla MIDI ve *sequencer* sistemleri kullanmaktadır. Dolayısıyla bu sistemlerin dijital kayıt teknolojisiyle birleşmesi ile kişisel bir müzik prodüksiyon yazılımı mümkün olacaktır. Üreticilerin de bu durumu fark etmesi ile birlikte sequnecer'ların yeni sürümleri dijital ses özellikleri ile, dijital ses düzenleme yazılımları ise MIDI sequencing imkanları ile piyasaya çıkmaya başlamıştır. (Doornbusch, 2009)

Sequencer'lar ve dijital ses düzenleme yazılımları arasındaki karşılıklı dönüşüm 1990'larda hem çok kanallı dijital ses kaydı yapabilen hem de MIDI sequencing imkanı sunan programları çalıştıracak kadar güçlü bilgisayarların üretilmesi ile tamamlanmıştır. Bu programlara örnek olarak MOTU şirketinin geliştirdiği Digital Performer, Cakewalk şirketinin geliştirdiği Pro Audio ve Emagic şirketinin geliştirdiği Logic Audio gösterilebilir (Manning, 2004). Bu programların sunduğu hassas editleme ve kayıpsız ses kaydı imkanı, analog kayıt sistemleri ve usturalı editleme (*razor-blade editing*) ile asla mümkün olmayacak kadar detaylı çalışmaya olanak sağlamıştır. (Doornbusch, 2009)

1990'larda büyük bir hızla gelişen ses yazılımları, bu yazılımlar için oluşturulmuş eklentileri (*plug-in*) de beraberinde getirmiştir. Günümüzde birçok farklı türdeki yazılımda sıklıkla kullanılan eklenti sistemi, grafik yazılımcılığı tarafından ortaya atılan bir fikirdir. Adobe gibi grafik yazılım üzerine çalışan şirketler, geliştirdikleri büyük yazılımlara kolaylıkla eklenebilecek "eklenti yazılım" mimarisi geliştirmişlerdir. Geliştirilen eklentiler belirli bir klasöre dahil edilerek kurulabilmekte ve asıl programa dahil olmaktadır. Ancak en önemlisi, bu eklentilerin üçüncü şahıslar tarafından da geliştirilebilir olmasıdır. Grafik tasarım yazılımı Adobe Photoshop, bu eklenti sistemini kullanan ilk yazılımlardandır ve çok geçmeden aynı şirketin video düzenleme yazılımı Premiere için de bir eklenti formatı geliştirilmiştir. Bilindiği üzere videolar görüntünün yanında ses de bulundurabilmektedir ve bu sebeple video düzenleme yazılımları belirli ses işleme fonksiyonları da sunmaktadır. Cybersound şirketi de Premiere'in ses düzenleme fonksiyonalitesini geliştirmek amacıyla sadece ses işlemleyen bir dizi eklenti geliştirmiştir. Cybersound şirketinin eklentilerin piyasaya çıkmasının hemen ardından ise bazı ses yazılımcıları Adobe Premiere'in eklenti mimarisini kendi ses yazılımlarına eklemlemişlerdir (M. Collins, 2012). Öte yandan bazı ses yazılımcıları da kendi eklenti mimarileri üzerine çalışmaya başlamıştır.

Digidesign, kendi eklenti mimarisini tasarlayan ilk üreticilerdendir ve İsrail merkezli Waves şirketi, Digidesign sistemleri için ses işleme eklentileri geliştiren ilk büyük şirkettir. 1996 yılında iki önemli gelişmeden bahsedilebilir: Digidesign mevcut Sound Designer yazılımı üzerindeki çalışmalarını sonlandırmış ve kaynaklarını sonradan en popüler DAW'lardan biri haline gelecek Pro Tools üzerine yoğunlaştırmıştır; ve Steinberg Cubase DAW'ı ile birlikte sonradan en popüler ses eklentisi mimarilerinden biri haline gelecek VST'yi piyasaya sürmüştür (Manning, 2009). Bu minvalde ses yazılımlarının gelişmesi ile ses eklentileri de önem kazanmaktadır.

Eklenti üretiminin kâr potansiyelini öngören bazı yatırımcılar hem profesyonel kullanıcılara yönelik gelişmiş ve karmaşık eklentilerin, hem de düşük bütçeli kullanıcılara yönelik daha basit eklentilerin üretimini desteklemiştir. Böylece ortaya çıkan eklenti türlerinde büyük bir çeşitlilik meydana gelmiş, sonuç olarak eklentilerin kullanım alanları oldukça genişlemiştir (Doornbusch, 2009). Meydana gelen çeşitlilik içerisinde en çok rağbet gören eklenti türü, ilk örneği 1997 yılında Propellerhead Software'in geliştirdiği ReBirth ile görülen sentezleyici simülasyonu eklentileridir. Setezleyici simülasyonu eklentileri, eski sentezleyicilerin seslerinin ve arayüzlerinin yazılım vasıtasıyla simüle edilmesi ile meydana getirilmektedir (M. Collins, 2012). Eski donanımların yenilenerek üretilmesi genellikle riskli bir yatırım olarak görüldüğü

için eklenti teknolojisi bu noktada cazip bir alternatif olarak kendini göstermiş, yaygınlığını artırmıştır (Manning, 2004).

Dijital ses teknolojileri 21. yüzyılın başından itibaren büyük ilerleme kat etmiş, büyük bir hızla gelişmiş ve yaygınlaşmıştır. Yazılımların ve donanımların ucuzlaması ve internet kaynaklarının büyük bir hızla artması, günümüzde bir müzik prodüksiyon stüdyosunun kurulmasını yirmi yıl öncesine kıyasla çok kolaylaştırmıştır. Ortaya çıkan kolaylık, bilgisayar müzisyenliğini de benzer şekilde etkilemiştir. Günümüzde bir müzik eserinin prodüksiyon sürecinin kayıt aşamasından sonraki kısmı tamamen tek bir DAW ve bu DAW'a eklenmiş eklentiler üzerinden yürütülebilmektedir. Bir bilgisayar müziği eseri ise – fiziksel enstrümanlar kullanılmadığı sürece – tamamen müzisyenin bilgisayarının içerisinde oluşturulup tamamlanıp yayımlanabilmektedir. DAW'ların çoğunun sunduğu ses işlemcilerinin ve sanal enstrümanların kısıtlı olduğu göz önünde bulundurulacak olursa, bu süreçlerin büyük oranda ses eklentilerinin desteğiyle gerçekleştiği iddia edilebilir.

Günümüzde ses eklentilerinin kullanımı o kadar yaygındır ki, ses mühendislerinin kullandığı birçok eklentiden oluşan işlemleme zincirlerinin analizine dayalı (Stasis, Jillings, Enderby ve Stables, 2017), reverb eklentilerine girilecek parametrelere yönelik (Peters, Choi ve Lei, 2012) ve ses eklentisi tercihine yönelik (Moura da Silva, Cavalcante Mattos ve de Souza Junior, 2019) öneri sistemlerine dair araştırmalar yapılmıştır.

Eklentilerin teknik yönünün incelenmesi gerekirse, 1990'lardan günümüze kadar birçok ticari ve akademik kuruluş tarafından çok sayıda farklı eklenti formatının geliştirildiği görülmektedir. 1990'lardan bu yana geliştirilen formatların büyük bir kısmı artık desteklenmese de VAMP gibi bazı formatlar spesifik kullanım alanlarında hizmet vermektedir. Ancak üç eklenti formatı uzun yıllardır ses eklentisi yazılımcılığında endüstri standardı olarak kullanılmaktadır: bunlar VST, AU ve AAX'tir.

İlk sürümü 1996 yılında Steinberg tarafından geliştirilmiş olan VST (*Virtual Studio Technology*) formatı, günümüzde üçüncü sürümündedir. 1999 yılında geliştirilen ve tüm ses eklentsi formatları arasında en çok sayıda yazılımın geliştirilmesinde kullanıldığı düşünülen VST2, 2018 yılının ekim ayı itibariyle desteklenmemektedir. Ayrıca resmi VST SDK'sından (*Software Development Kit*) çıkarıldığı için günümüzde artık yeni yazılımcıların yasal yollarla VST2 geliştirmesi mümkün değildir (SDK'lar yazılımcıların belirli bir platformda uygulama geliştirebilmesini sağlayan araçlardır). Ancak Steinberg'le halıhazırda lisans anlaşması bulunan

geliştiriciler bu durumdan muaftır ve VST2 yazılımları geliştirebilmektedirler. VST3 SDK, Steinberg'in resmi internet sitesinden ücretsiz olarak indirilebilmektedir ("3rd Party Developer", t.y.; "Technologies", t.y.; Pirkle, 2019).

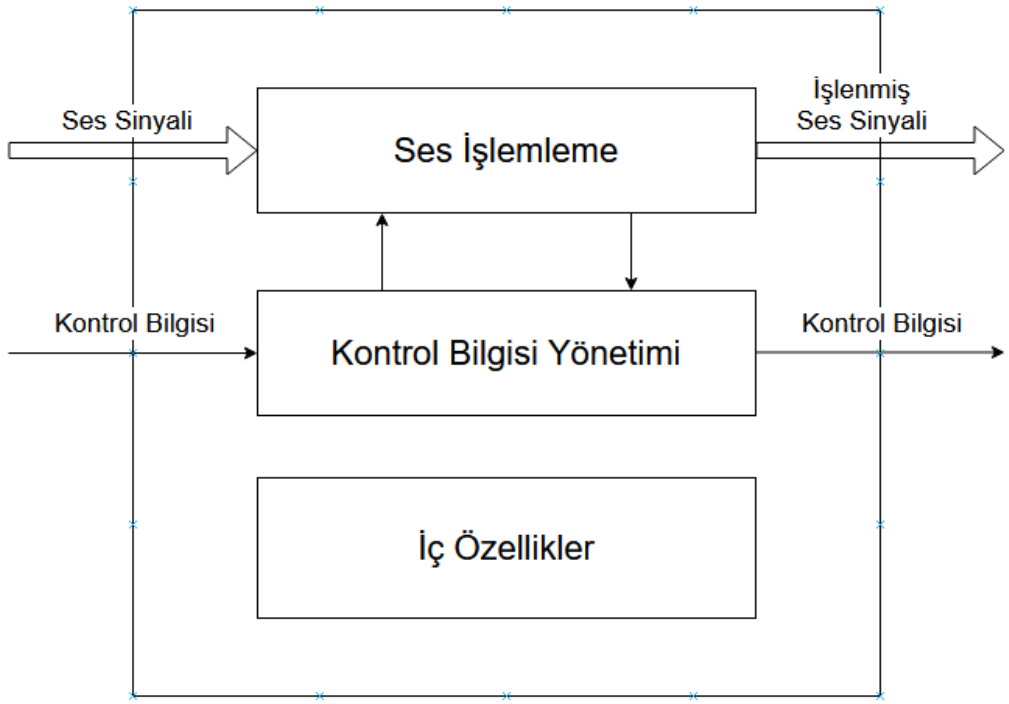
Audio Unit mimarisi, VST ve AAX'in aksine, işletim sisteminin ses API'ı (*Application Programming Interface*) ile doğrudan entegre edilmiştir. Apple'ın macOS ve iOS işletim sistemlerindeki Core Audio tarafından sunulan bu format, içerik olarak VST ile benzerlik göstermektedir. Günümüzde AUv2 ve AUv3 olmak üzere iki sürümü bulunmaktadır. AUv3 genellikle iOS uygulamaları tarafından, AUv2 ise genellikle iOS dışı uygulamalar tarafından kullanılmaktadır. AU SDK, Apple'ın resmi internet sitesinden ücretsiz olarak indirilebilmektedir ("AudioUnit | Apple Developer Documentation", t.y.; Pirkle, 2019).

AAX (*Avid Audio eXtension*), günümüzde Avid olarak bilinen Digidesign şirketi tarafından geliştirilmiştir. İki başarılı eklenti mimarisi olan RTAS (*Real Time Audio Suite*) ve ProTools TDM (*Time Division Multiplexing*) üzerine kurulmuş üçüncü nesil bir eklenti formatı ve API'ıdır. AAX'ler DSP ve Native olmak üzere iki türe sahiptir. AAX DSP'ler harici DSP devreleri ve DSP kodu kullanırken AAX Native eklentiler kullanıcının işlemcisi üzerinde çalıştırılır ve C++ ile yazılır. AAX'in en önemli farkı, özellikle ProTools yazılımı, diğer AAX Host'ları ve bazı Avid donanımlarınca çalıştırılmak üzere tasarlanmış olmasıdır. Bu sebeple AAX eklentileri ProTools dışındaki hiçbir DAW'da çalıştırılamamaktadır. AAX SDK, Avid'in resmi internet sitesinden ücretsiz olarak indirilebilmektedir. Ancak kullanıcının geliştirdiği AAX eklentisini denemesi ve hatalarını ayıklaması için Pro Tools yazılımının geliştiricilere yönelik sürümü Pro Tools Dev'e sahip olması gerekmektedir. Çünkü eklentinin başka bir yazılımla test edilmesi mümkün değildir ("Alliance Partner Program | AAX Connectivity Toolkit | Avid", t.y.; Pirkle, 2019).

Bahsedildiği gibi birçok farklı eklenti mimarisi bulunsa da, neredeyse tüm mimarilerin üzerine kurulmuş olduğu bazı temeller vardır. Tarihçede de değinildiği üzere, eklentiler belirli bir ana yazılıma çeşitli farklı fonksiyonlar eklemek amacıyla yazılmış nispeten ufak yazılımlardır. Bu bağlamda eklentileri eski Atari oyun kartuşlarına benzetmek mümkündür. Kartuşlar tek başlarına bir işleve sahip değillerdir. Ancak içlerinde bulunan oyun yazılımını doğru bir şekilde çalıştırabilen sistemlere entegre edilerek çalıştırılabilirler. Günümüzde bazı eklentiler müstakil olarak da çalıştırılabilecek şekilde geliştirilse de – bunlara "*standalone*" denmektedir – eklentilerin genel çalışma prensibi Atari kartuşlarıyla aynıdır. Bir eklentinin çalışması için entegre edileceği bir sistem gerekmektedir. Bu sistemlere "eklenti konakçısı" anlamına gelen *plugin host* ismi verilmektedir.

Ses sinyali genellikle belirli bilgi blokları halinde işlem görür. Bu bloklara arabellek (*buffer*) ismi verilmektedir. Arabelleklerin içerisinde ses sample'ları bulunur ve arabellekler sample rate ile senkronize olarak düzenlenir. Kontrol bilgileri genellikle daha düşük hızlı çevrimlerle kontrol edilir ve eklenti parametreleri gibi işlemlemeye yönelik bilgilerden oluşur. Kontrol bilgileri çoğunlukla iki ses arabelleği arasında işleme alınır, ancak çeşitli zaman damgaları ile işlem sırasına alınmaları da mümkündür. İç özellikler genellikle statik durum işaretlerinden oluşur. Bu durum işaretleri çoğunlukla eklentinin giriş/çıkış kanal sayısı, gecikme süresi, MIDI işlemleme imkanı, host'un mevcut temposu gibi eklenti-host etkileşimini sağlayan belirli bilgilerden oluşur(Goudard ve Muller, 2003).

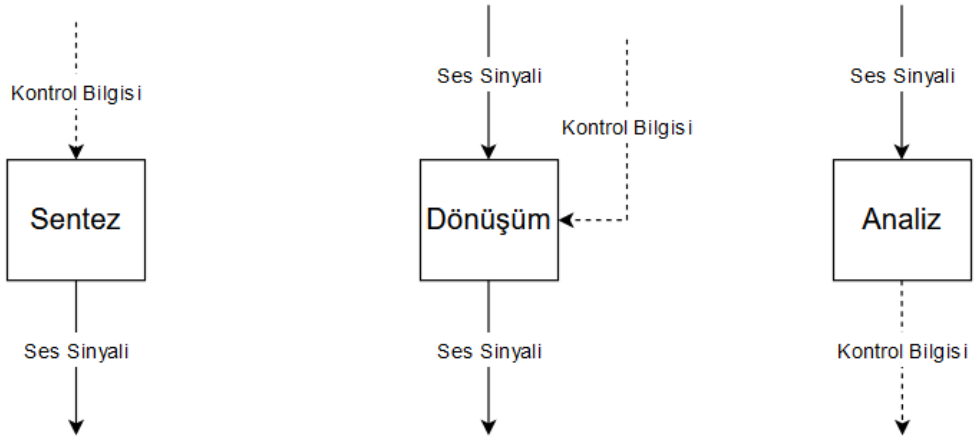
Eklentiler ses sinyalleri üzerinde çalışır ve çalışmalarını belirli kontrol bilgilerine göre yönlendirir. Bu ses sinyalleri de kontrol bilgileri de host ve eklenti arasındaki veri alışverişi ile aktarılır. Bu sebeple host ve eklentinin birbirlerinin düzenini ve özelliklerini bilmesi oldukça önemlidir. Bir eklentinin iç yapısına dair genel şema, Şekil 2.1'de sunulmuştur.



Şekil 2.1 Eklentilerin iç yapısına dair genel şema

Şekil 2.1'de sol taraftaki sinyal ve kontrol bilgisi okları host'tan eklentiye gönderilen giriş akışını, sağ taraftaki oklar ise eklentiden host'a gönderilen çıkış akışını sembolize etmektedir. Giriş ve çıkış bilgileri, eklenti algoritmalarının işlevine bağlı olarak önem kazanmaktadır. Eklentiler tarafından yapılan işlemlemeler kabaca üç

kategoriye ayrılabilir: sentez, dönüşüm ve analiz. Bu kategorilere dair giriş ve çıkış akışları Şekil 2.2'de gösterilmiştir.



Şekil 2.2 Temel eklenti türlerinin sinyal akışı

Şekilde gösterilen giriş/çıkış akış şemaları temel türleri kabaca kategorize etmek amacıyla çizilmiştir. Uygulamada çok farklı durumlarla karşılaşılabilmektedir. Örneğin günümüzde kullanılan pek çok sentezleme eklentisi otomasyon ya da döngüde çalınabilme gibi farklı nedenlerle giriş olarak gelen kontrol bilgisini ayrıca bir çıkış olarak sunmaktadır. Bunun dışında bahsedilen kategorizasyona girmeyen eklenti türleri de vardır. Yalnızca kontrol bilgisi alıp kontrol bilgisi veren MIDI efektleri, bunlara bir örnek olarak gösterilebilir. Ayrıca faz tersleyici gibi hiçbir kontrol bilgisi almaksızın çalışan eklentilere de rastlanabilmektedir. (Goudard ve Muller, 2003)

3 YÖNTEM

3.1 İhtiyaçların Belirlenmesi

Sonic-Pi'nin mevcut versiyonunda, ses eklentilerini çalıştırabilme ve yönetebilme ihtiyacı mevcuttur. Bu ihtiyacın giderilebilmesi için öncelikle ses eklentilerini çalıştıracak bir yazılımın geliştirilmesi gerekmektedir. Ardından geliştirilen yazılım ile Sonic-Pi arasındaki iletişmi sağlayacak araçların Sonic-Pi kaynak koduna eklenmesi gerekmektedir. Dolayısıyla bu çalışmada Sonic-Pi uygulamasının ses eklentileriyle kullanılmasını sağlayan yazılım (*SonicPluginHost*) geliştirilmiştir. Ayrıca Sonic-Pi ile geliştirilen yazılım arasındaki iletişimi sağlamak için Sonic-Pi'nin kaynak koduna bazı eklemeler yapılmıştır.

SonicPluginHost yazılımının geliştirilme amacını gerçekleştirebilmesi için sahip olması gereken başlıca özellikler şunlardır:

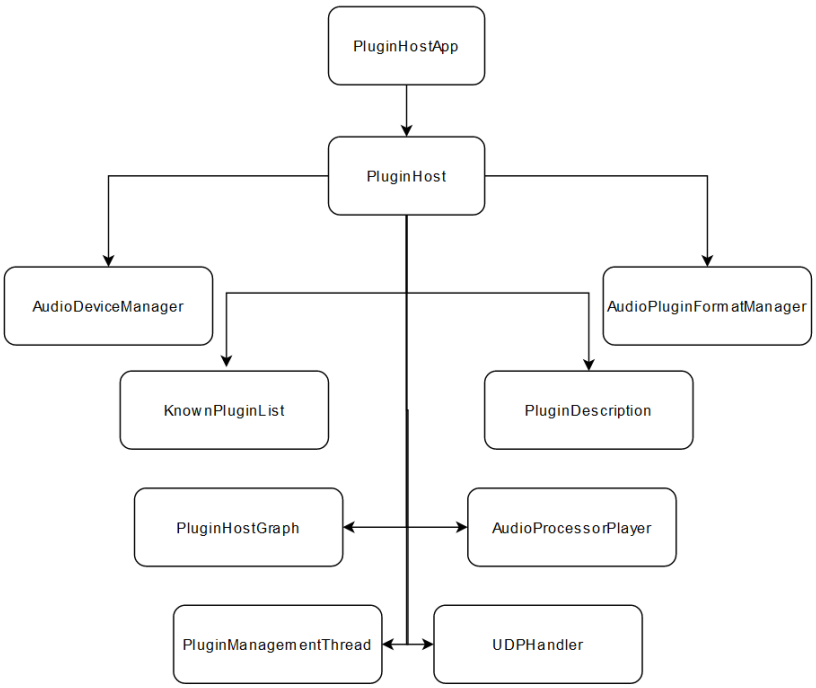
- Belirli bir klasördeki eklentileri tarayabilmelidir.
- Taranan eklentileri çalışmaya hazır halde tutabilmelidir.
- Sonic-Pi'den gelecek komutları her an dinleyebilmelidir.
- Sonic-Pi'den gelecek komutlara bağlı olarak istenen eklentiyi çalıştırabilmelidir.
- Çalıştırılan eklentinin ses-MIDI bağlantılarını yapabilmelidir.
- Sonic-Pi'den gelen MIDI mesajlarını eklentiye aktarabilmelidir.
- Sonic-Pi'den gelen parametre mesajlarını eklentiye aktarabilmelidir.
- Eklenti(ler)den alınan sesi çalınmak üzere bilgisayarın ses cihazlarına gönderebilmelidir.

Sonic-Pi yazılımı da kullanıcı tarafından girilen eklenti yükleme ve belirli eklentilere MIDI/parametre mesajı gönderme komutlarını SonicPluginHost'a doğru bir şekilde iletilmesini sağlamalıdır.

Bu çalışmada; bilgisayara yüklü ses cihazlarının saptanmasında, birçok eklenti formatının birlikte kullanılabilmesinde ve programın platformdan bağımsız çalışabilmesinde sağladığı verimden dolayı SonicPluginHost'un geliştirilmesinde JUCE Framework kullanılmıştır.

3.2 SonicPluginHost Yazılımı

3.2.1 SonicPluginHost'un mimarisi



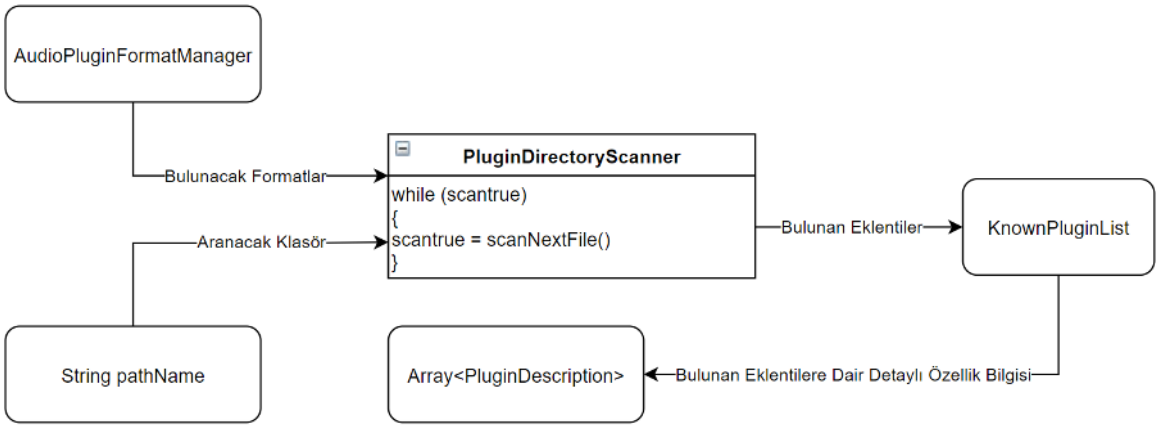
Şekil 3.1 SonicPluginHost genel mimarisi

SonicPluginHost C++ dilinde nesne yönelimli programlama (*object oriented programming*) stiliyle tasarlanmıştır ve genel yapısı, Şekil 3.1'de sunulmuştur. Bu şemaya göre en üstteki PluginHostApp sınıfı (*class*), JUCEApplicationBase sınıfından türetilmiştir ve JUCE'un özel giriş noktası (*entry point*) olan START_JUCE_APPLICATION() fonksiyonuna parametre olarak girilmektedir. Yazılımın en geniş tek örnekli (*single instance*) sınıfı PluginHost sınıfıdır. PluginHost, farklı işlevlere sahip birçok sınıfı üye değişkenler (*member variable*) olarak bünyesinde barındırmaktadır. Bu sınıflardan:

- AudioDeviceManager bilgisayardaki ses cihazları ile,
- AudioPluginFormatManager çeşitli farklı eklenti formatları ile,
- KnownPluginList tespit edilen eklentiler ile,
- PluginDescription tespit edilen eklentilerin detaylı özellik bilgileri ile,
- PluginHostGraph eklentilerin çalıştırılabilmesi ile,
- AudioProcessorPlayer eklentilerin içerisinden geçirilecek veri akışı ile,
- PluginManagementThread Sonic-Pi'den gelebilecek mesajlar ile,
- UDPHandler UDP veri alışverişi ile,

ilgili fonksiyonlar içermektedir.

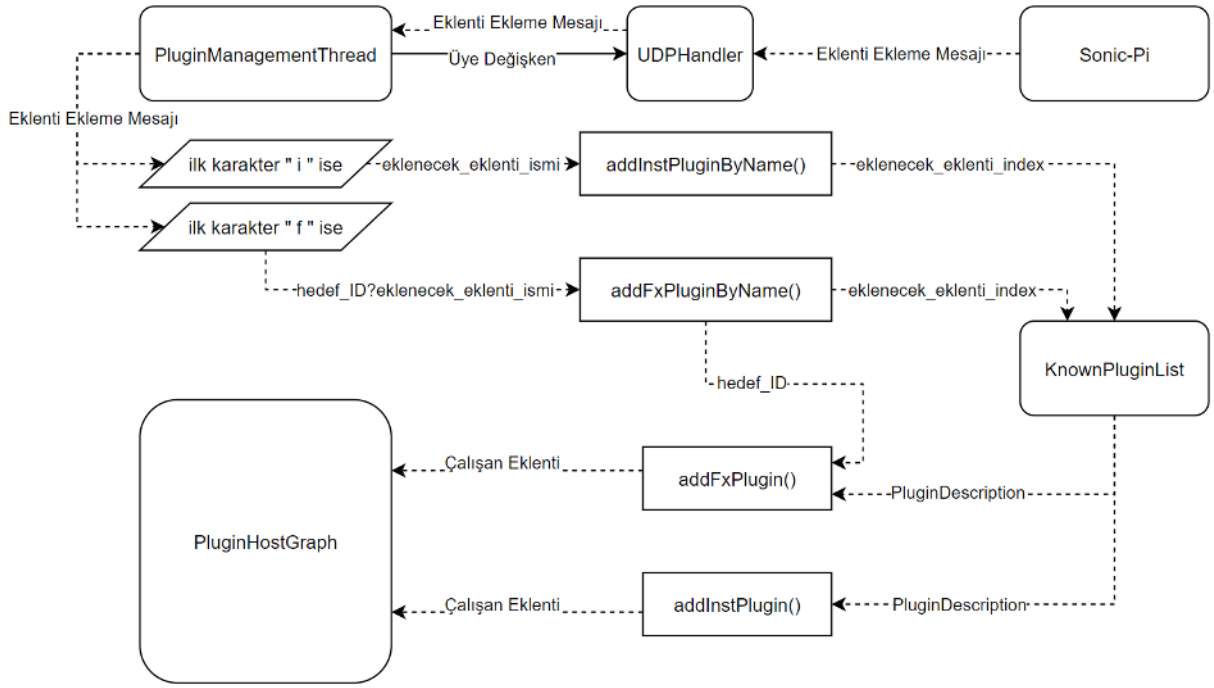
3.2.2 Belirtilen dizindeki eklentilerin tespit edilmesi



Şekil 3.2 Eklenti tespit süreci

SonicPluginHost'un kullanılabilecek eklentileri tespit etme süreci, JUCE'a ait PluginDirectoryScanner sınıfından bir nesnenin yaratılması ile başlamaktadır. Sürecin iş akışı Şekil 3.2'de sunulmuştur. Buna göre, bu PluginDirectoryScanner nesnesi yaratılırken AudioPluginFormatManager nesnesinden temin edilen bulunacak format tipi, aranacak klasöre dair dosya yolu (*file path*) bilgisini içeren bir string ile elde edilmiş bir FileSearchPath nesnesi, ve bulunan eklentilerin ekleneceği bir KnownPluginList nesnesi referansı parametreleri kullanılır. Bu parametrelerle yaratılmış nesnenin scanNextFile() fonksiyonu ise "scantrue" boolean değerine bağlı bir while döngüsü ile çağırılır. Bu fonksiyon yüklenmesi mümkün olan eklentileri bulur ve eklentiler KnownPluginList'e eklenir. Ardından, döngü tamamlandıktan sonra, KnownPluginList'teki tüm eklenti özellik bilgileri PluginDescription dizinine (*array*) aktarılır.

3.2.3 Eklentilerin Sonic-Pi'den gelecek mesajla çizelgeye eklenmesi



Şekil 3.3 Eklenti ekleme süreci

Eklentilerin Sonic-Pi'den gelecek mesajlarla çizelgeye (*graph*) eklenme süreci, ardışık birtakım işlemlerin başarılı şekilde sonuçlanmasıyla ortaya çıkmaktadır. Bu ekleme süreci ve ilgili işlemler Şekil 3.3'te sunulmuştur. Öncelikle Sonic-Pi'den gelecek mesajların sürekli olarak dinlenmesi gerekmektedir. Bu iş için bir PluginManagementThread nesnesi yaratılmıştır. Nesne, kendine ait bir iş parçacığı (*thread*) üzerinde sürekli olarak Sonic-Pi'den gelen mesajları kontrol etmektedir. Sonic-Pi'den gelen mesajların elde edildikleri anda işleme sokulmaları gerektiği için, bu işlemde ayrı bir iş parçacığının kullanılmıştır. Böylece SonicpluginHost, programın hangi aşamasında olursa olsun gelenmesajları dinleyebilmekte ve iletebilmektedir.

Gelecek mesajları dinleyecek araç geliştirildikten sonra, mesajların nereden dinleneceği tanımlanmalıdır. Bu iş için PluginManagementThread sınıfına bir UDPHandler nesnesi eklenmiştir. UDPHandler sınıfı, Sonic-Pi ile SonicPluginHost arasındaki Kullanıcı Veribloğu İletişim Kuralları (*User Datagram Protocol - UDP*) yoluyla gerçekleşecek veri akışını sağlamak amacıyla oluşturulmuştur. UDP, bağlantı kurmayan bir İşlemler Arası İletişim (*Interprocess Communication - IPC*) protokolüdür. Bağlantı kurmaması dolayısıyla güvensiz ancak hızlıdır. Bu sebeple ses ve görüntü işlemleme gibi hız gerektiren kullanımlarda sıklıkla tercih edilmektedir.

PluginManagementThread 54020 numaralı bağlantı noktasını (*port*) üyesi durumundaki UDPHandler nesnesi vasıtasıyla sürekli olarak dinlemektedir. Bu bağlantı noktası Sonic-Pi ve SonicPluginHost arasındaki ana iletişim kanalıdır. Bu kanaldan elde edilen mesajlar PluginManagementThread'e tanıtılmış anahtarlarla sürekli olarak karşılaştırılır ve doğru anahtarı taşıyan mesajlar işleme sokulur.

Eklenti ekleme mesajı mimarisi iki aşamalı olarak tasarlanmıştır. Bunun sebebi Bölüm 3.2.4'te daha detaylı açıklanacak olan enstrüman ve efekt eklentilerinin farklı veri akış şemalarıdır. Sonic-Pi'den SonicPluginHost'a gönderilen her mesaj işlevine göre bir anahtar taşımaktadır. Bu anahtarlar mesajların önüne eklenen tekil karakterlerden ibarettir.

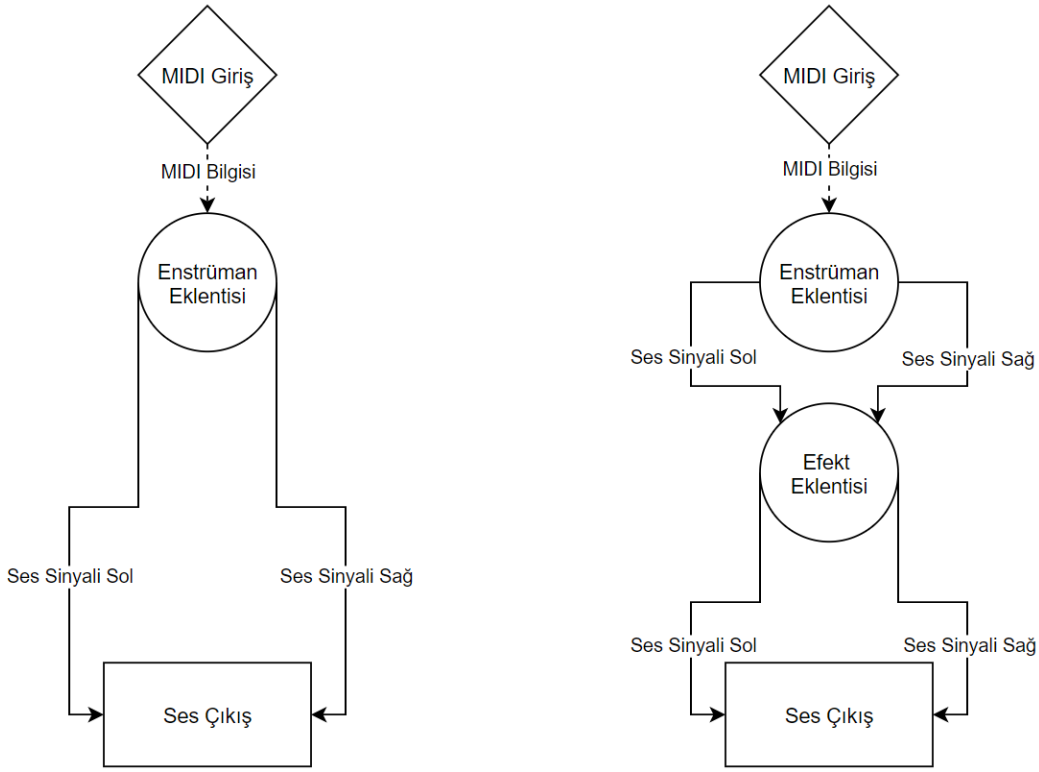
Sonic-Pi'den SonicPluginHost'a gönderilen mesajlar, maesajın işlevine göre belirli anahtarlarla ayrıştırılır. Enstrüman eklentisi ekleme mesajları "i" anahtarı ile, efekt eklentisi ekleme mesajları "f" anahtarı ile işaretlendirilir. SonicPluginHost, elde ettiği mesajları öncelikle başlarındaki anahtara göre ayrıştırır, sonra parçalar ve işleme koyar.

Enstrüman mesajlarının gövdesi tek parçalıdır. Bu tek parça eklenecek eklentinin isminden meydana gelir. addInstPluginByName() fonksiyonu ile mesajdaki eklenecek eklentinin ismi, KnownPluginList nesnesindeki eklentiler ile karşılaştırılır ve istenen eklentiye ait detaylı bilgileri içeren PluginDescription nesnesi elde edilir. Daha sonra, elde edilen nesne iki temel eklenti ekleme fonksiyonundan biri olan addInstPlugin()'den parametre olarak geçirilir. addInstPlugin() fonksiyonu bilgileri verilen eklentiyi çalıştırır ve PluginHostGraph'a, yani ses işlemcileri çizelgesine ekler. Bu çizelge, yüklenmiş ve çalıştırılmış eklentilerin eklendiği ve birbirlerine bağlanabildiği bir çizelgedir. Çizelgeye eklenen eklentiler artık *node* (bağlantı noktası) olarak isimlendirilir ve çizelgeye eklenen node'lara birer kimlik numarası (*ID*) atanır. Çalışmakta olan eklentiler ile ilgili yapılacak işlemler bu ID'ler üzerinden yapılır. SonicPluginHost'ta eklentiler çizelgeye eklenme sıralarına göre 101'den başlayacak şekilde ID'lendirilirler.

Efekt mesajlarının gövdesi ise üç parçadan meydana gelir. Efekt eklentileri ses üretmeyen, kendilerine sunulan ses üzerinde değişiklik yapan uygulamalardır. Bu sebeple bir efekt eklentisi eklendiğinde bu eklentinin ses kaynağı olarak atanacak ve mevcut olarak çalışmakta olan bir eklenti belirtilmelidir. Efekt mesajlarının gövdesi; ses kaynağı teşkil edecek çalışmakta olan eklentinin ID'si, "?" karakteri ve eklenecek efekt eklentisinin isminden oluşur. addFxPluginByName() fonksiyonu ile bu üç bölümlü mesaj, "?" karakteri orta nokta alınarak ikiye bölünür. Elde edilen eklenecek

eklenti ismi, enstrüman eklentileri ile aynı şekilde işlemlenir ve ilgili PluginDescription nesnesi elde edilir. Efektlere yönelik temel eklenti ekleme fonksiyonu olan addFxPlugin(), eklenecek eklentiye ilişkin PluginDescription ve hedef (ses kaynağı olarak atanacak) eklenti ID'si olmak üzere iki parametre ile çalışmaktadır.

3.2.4 Yüklenen eklentilerin bağlantıları



Şekil 3.4 Enstrüman ve efekt eklentilerin bağlantı şemaları

Enstrüman eklentileri ve efekt eklentileri, belirli karakteristik bağlantı şemalarına sahiptir. Bu şemalara göre enstrüman eklentileri bir MIDI kaynağından bilgi alacak şekilde bağlanır, ve işlemlediği ses bilgisini bağlandığı bir ses çıkışına gönderir. Efekt eklentileri ise çalışmak için ses bilgisi gerektirir. Dolayısıyla işlemlenecek bilgiyi almak için bir ses kaynağına, işlemledikleri bilgiyi iletmek için ise bir ses çıkışına bağlanırlar. Bu bağlantı şemasını görselleştirmek adına Şekil 3.4'te iki adet örnek eklenti çizelgesi içeriği sunulmuştur. SonicPluginHost'un eklenti ekleme mimarisi, bu özgün bağlantı şemalarına uygun olacak şekilde tasarlanmıştır.

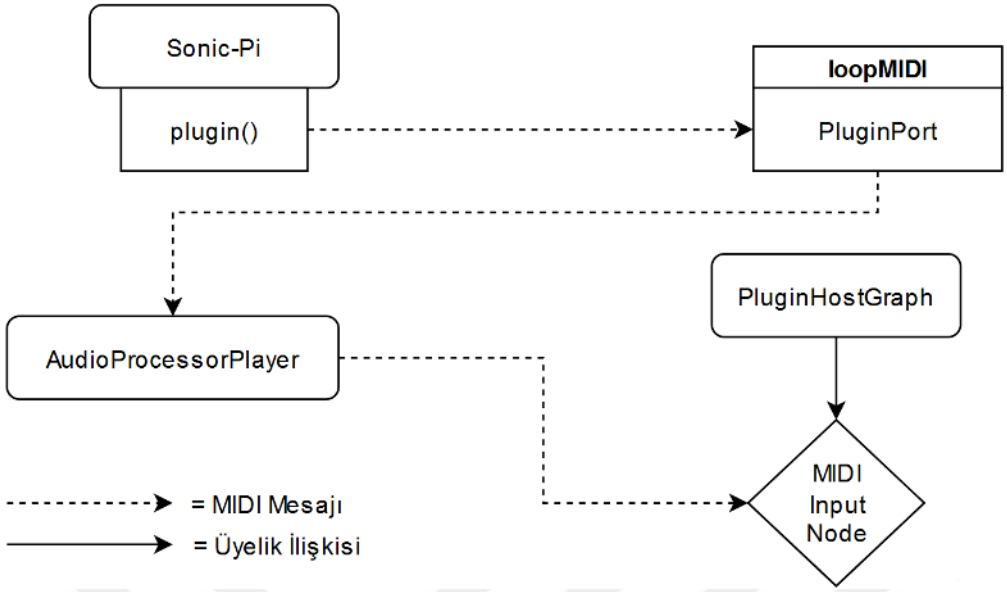
SonicPluginHost çizelgesi (PluginHostGraph) ses ve midi giriş/çıkışını sağlayan iki adet özel node ile birlikte oluşturulur. Bunlar MIDI girişini sağlayan "MidiInNode" ve ses çıkışını sağlayan "AudioOutNode" node'larıdır. Bu node'lar AudioProcessorPlayer nesnesi ile bağlantı kurarak gelen MIDI mesajlarının

çizelgeye sokulmasını ve çizelgede oluşturulan ses sinyalinin bilgisayara bağlı ses çıkış cihazlarına gönderilmesini sağlar. Özel node'lara birden fazla eklenti bağlanabilir ancak her çizelgede bu özel node'lardan yalnızca birer adet bulunabilir. MidiInNode'a birden fazla eklentinin bağlanması halinde aynı midi mesajı farklı eklentilerden çalınabilir. AudioOutNode'a birden fazla eklenti bağlanması durumunda ise bu eklentilerden alınan ses karıştırılarak oynatılacaktır.

Enstrüman ve efekt eklentilerinin özgün bağlantı şemalarından dolayı Sonic-Pi tarafından SonicPluginHost'a gönderilen eklenti ekleme komutları ikiye ayrılmıştır. Enstrüman ekleme komutu alındığında ismi verilen enstrümanın MIDI girişi MidiInNode'a, ses çıkışları ise AudioOutNode'a bağlanır.

Efekt eklentilerinin doğru bir şekilde çalışabilmeleri için işlenmesi istenen sesin eklentinin ses girişlerine gönderilmesi gereklidir. Bu sebeple çizelgeye efekt eklentisi ekleme komutu, hem eklenmesi istenen eklentinin ismini hem ses kaynağı olarak alınacak node'un ID'sini barındırmalıdır. Hedef eklentinin isminin değil ID'sinin girilmesiyle hem aktif node'lar arasından istenilenin bulunması aşaması hızlandırılmıştır, hem de canlı kodlama esnasında kullanıcının girmesi gereken karakter miktarı azaltılmıştır. Efekt eklentisi ekleme komutu alındığında öncelikle ID'si verilen eklentinin AudioOutNode ile bağlantısının olup olmadığı araştırılır, mevcut bağlantılar sökülür. Bu işlemden sonra ismi verilen efekt eklentisi çizelgeye eklenir ve ses girişleri hedef eklenti ile, ses çıkışları AudioOutNode ile bağlanır.

3.2.5 SonicPluginHost'un Sonic-Pi'den gelecek MIDI mesajlarını işlemlemesi



Şekil 3.5 Gelen midi mesajlarının işlemlenme şeması

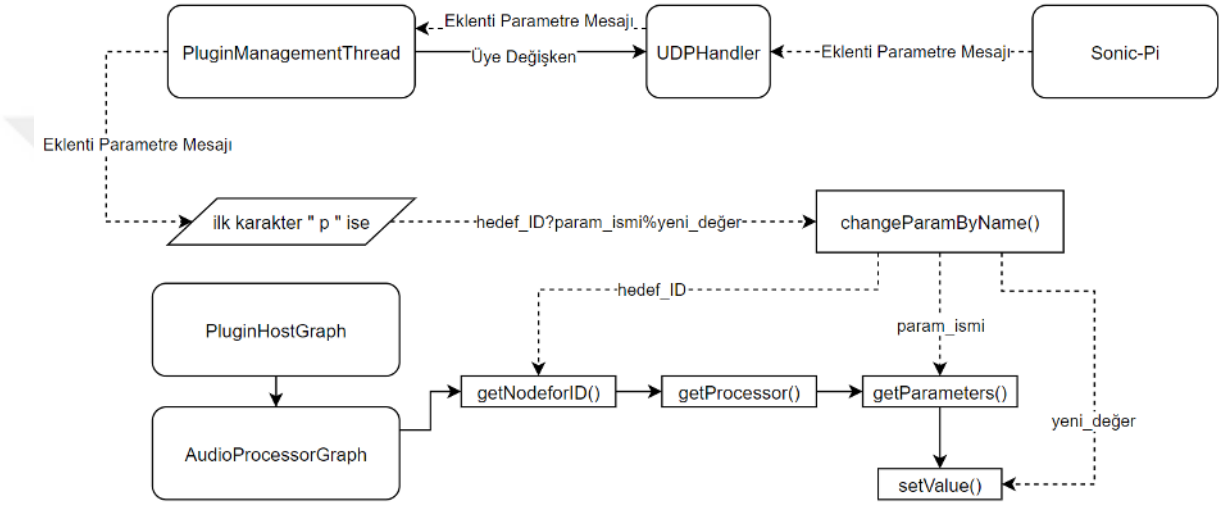
Eklenen enstrüman pluginlerinin kullanılabilmesi için MIDI mesajlarının doğru bir şekilde gönderilmesi ve elde edilmesi oldukça önemlidir. SonicPluginHost'un MIDI alışveriş şeması Şekil 3.5'te sunulmuştur. MIDI alışverişinin yürütülebilmesi için bilgisayarın işletim sistemi ile iletişime geçmek gerekebilmektedir. Windows yüklü cihazlarda MIDI cihazları ve aralarındaki MIDI veri akışları işletim sistemi yardımıyla yürütülmektedir. Bu durumun avantajları ve dezavantajları mevcuttur. Bu dezavantajlardan birisi veri alışverişine konu olabilecek bağlantı noktalarının tanımlanmasıdır. Uygulamaların Windows MIDI altyapısı üzerinden veri alışverişinde bulunabilmesi için Windows tarafından "MIDI Cihazı" olarak tanınmaları gerekmektedir. Dolayısıyla Windows tarafından cihaz olarak tanınmamış iki uygulama MIDI işlemleme özelliklerine sahip olsalar dahi birbirini görememekte, iletişim kuramamaktadır.

Bu sorunu aşmak için sanal MIDI bağlantı noktaları (*Virtual MIDI Port*) kullanılabilir. Sanal bağlantı noktaları, ürettikleri bağlantı noktalarını Windows'a MIDI cihazları olarak tanıtan yazılımlardır. Bu yazılımlar ile sanal bir MIDI bağlantı noktası oluşturulur, MIDI mesajını gönderecek uygulama gönderimi bu noktaya yapar ve MIDI mesajlarını dinleyecek uygulama aynı bağlantı noktasını bir MIDI cihazı olarak dinler. SonicPluginHost'un geliştirilmesinde MIDI iletişimini sağlayacak sanal bağlantı noktası olarak ücretsiz "loopMIDI" yazılımı kullanılmıştır. Ancak oluşturulan sanal MIDI cihazları isimlendirilebildiği sürece her sanal bağlantı noktası yazılımı kullanılabilmektedir. SonicPluginHost ve beraberinde yapılan değişikliklerle Sonic-Pi

uygulamaları, “PluginPort” isminde bir sanal bağlantı noktası aracılığıyla iletişime geçmektedirler.

PluginPort cihazı AudioDeviceManager vasıtasıyla AudioProcessorPlayer nesnesine tanıtılır. Böylece PluginPort bağlantı noktasına gönderilen her MIDI mesajı AudioProcessorPlayer tarafından dinlenecek ve PluginHostGraph içerisindeki MidiInNode vasıtasıyla bağlı eklentilere sunulacaktır.

3.2.6 SonicPluginHost’un Sonic-Pi'den gelecek parametre mesajlarını işlemlemesi



Şekil 3.6 Gelen parametre mesajlarının işlemlenme şeması

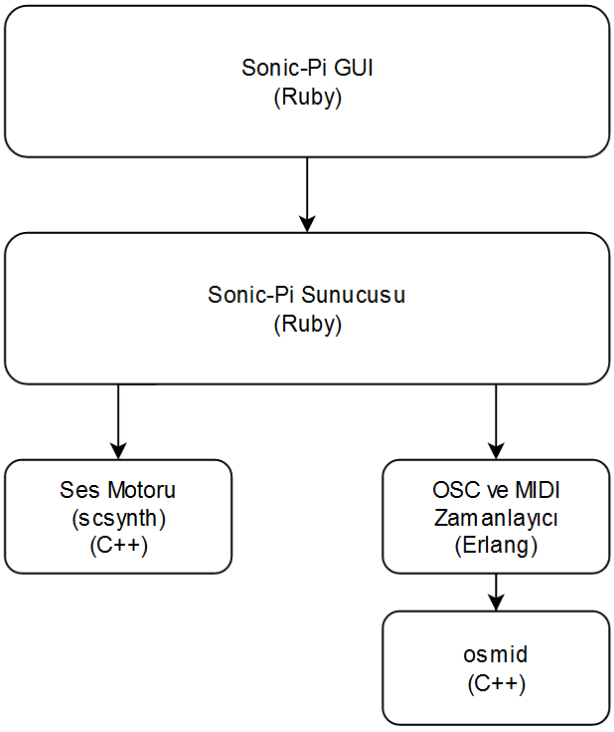
Eklentilerin ses yelpazelerinin oldukça geniş olmasının en önemli sebeplerinden biri parametrelerdir. Parametreler eklentilerin ses işlemleme şekillerini, dolayısıyla ortaya çıkardıkları sesi değiştiren ve çeşitlendiren değerlerdir. Bir eklentide yüzlerce farklı parametre olabilir ve eklentiden elde edilmesi istenen sesi karakterize eden, bu parametreler üzerinde yapılacak hassas ayarlamalardır. Öte yandan parametrelerin çeşitli cihazlarla performans esnasında değiştirilerek sesin çeşitlendirilmesi de, elektronik müzik içerisinde sıklıkla kullanılan bir yöntemdir. Dolayısıyla elektronik müzik üretimine ve performansına yönelik bir uygulamanın gerçek zamanlı parametre manipülasyonu yapabilmesi oldukça önemlidir.

SonicPluginHost’a gönderilen parametre değiştirme komutları, eklenti ekleme komutları ile aynı şekilde PluginManagementThread tarafından dinlenir ve işleme koyulur. Tıpkı diğer komutlar gibi, parametre değiştirme komutları da tek karakterlik bir anahtara sahiptir. Eklenti değiştirme mesajları için geçerli anahtar “p” karakteridir. Gelen mesajın ilk harfi p ise, mesaj PluginManagementThread tarafından

changeParamByName() fonksiyonuna yollanır. Bu fonksiyona yollanan mesaj, efekt eklentisi ekleme mesajlarına benzer olarak, beş parçadan meydana gelmektedir. Bu parçalar sırasıyla parametresi değiştirilecek eklentinin ID'si, "?" karakteri, değiştirilecek parametrenin ismi, "%" karakteri ve parametrenin yeni değeridir. Mesaj elde edildikten sonra, changeParamByName() fonksiyonu bu parçaları sınırlayıcı karakterleri saptamak suretiyle ayırır ve mesajı üç farklı bilgiye böler. İlk bilgi olan eklenti ID'si, PluginHostGraph'a bağlı AudioProcessorGraph grafiğinin getNodeforID() fonksiyonuna sunularak istenen node'un elde edilmesi için kullanılır. Ardından, getProcessor() fonksiyonu ile node'un ile ilişkili eklenti bilgilerine ulaşılır ve getParameters() fonksiyonu ile eklentinin parametrelerinin tamamından oluşan bir liste elde edilir. İkinci bilgi olan parametre ismi, eldeki tüm parametrelerden oluşan listeden istenen parametrenin bulunması için kullanılır. Parametre saptandıktan sonra ise setValue() fonksiyonu ile parametrenin değeri üçüncü bilgi olan yeni parametre değeri ile değiştirilir.

3.3 Sonic-Pi Yazılımı

3.3.1 Sonic-Pi'nin mimarisi



Şekil 3.7 Sonic-Pi yazılımının genel mimarisi

Sonic-Pi, temelde üç farklı katmandan oluşmaktadır. Bu üç katmanlı mimari, Şekil 3.7'de sunulmuştur. En üst – ve kullanıcıya en yakın – katmanda Sonic-Pi'nin grafik

kullanıcı arayüzü (*Graphical User Interface - GUI*), diğer bir isimlendirmeyle Bölüm 2.2'de bahsedilen Sonic-Pi IDE bulunmaktadır. Bu grafik kullanıcı arayüzü, kullanıcıların Sonic-Pi ile iletişime geçtiği katmandır ve Sonic-Pi'nin tüm görsel ögeleri bu katmana dahildir. Kullanıcının bu katmandaki metin düzenleyici vasıtasıyla girdiği tüm kodlar Sonic-Pi Sunucusu tarafından işlenmektedir.

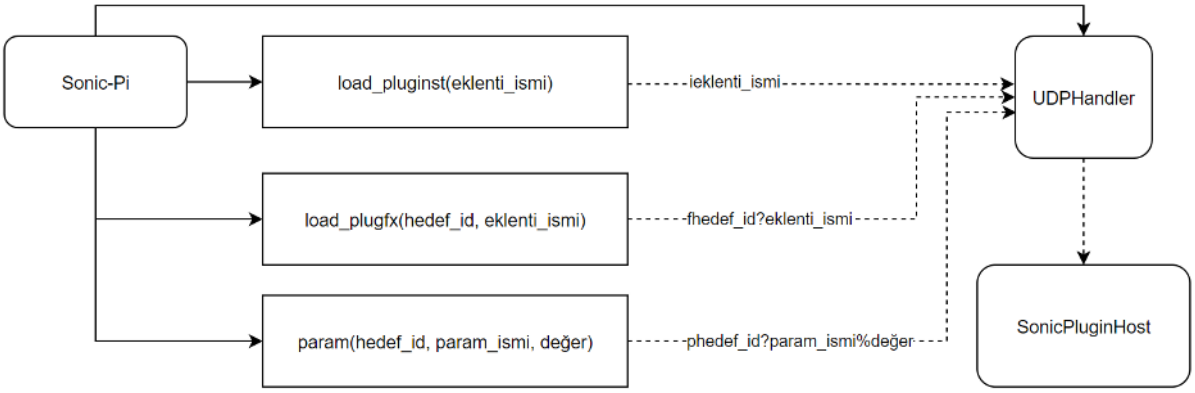
Sunucu ise, Sonic-Pi dilinin tüm özelliklerini barındıran katmandır. GUI vasıtasıyla girilen kodlar sunucuda ayrıştırılır, incelenir ve gerekli fonksiyonlar gerekli zamanlamalarla çağırılır. Sunucuda işlemlenen komutlar ses motoruna aktarılır. GUI ve sunucu, Ruby dilinde yazılmıştır.

Ses motoru Bölüm 2.1.1'de bahsedilen "SuperCollider Server", diğer ismiyle *scsynth*'dir ve C++ dilinde yazılmıştır. GUI vasıtasıyla elde edilen ve sunucuda işlemlenen mesajlar, ses bilgisine dönüşmek üzere ses motoruna gönderilir. Ses motorunun kendi içerisinde bir zamanlayıcısı (*scheduler*) vardır ve sunucudan gelen komutların zamanlamasını kendi ayarlar.

Bahsedilen üç katman, Sonic-Pi'nin temel katmanlarıdır. Ancak üçüncü sürüm itibariyle Sonic-Pi'ye harici uygulamalarla OSC ve MIDI yoluyla bağlantıya geçebilmesi için iki öge eklenmiştir. OSC, genellikle multimedya uygulamalarınca kullanılan bir ağ protokolü türüdür. Hassaslığı ve esnekliği dolayısıyla sıklıkla tercih edilmektedir. Sonic-Pi'de de Şekil 3.7'de oklarla gösterilen katmanlar arası iletişim OSC mesajları vasıtasıyla sağlanmaktadır.

Üçüncü sürüm ile eklenen MIDI ve OSC komutu gönderimi özelliği, bu komutların gönderim zamanlarının düzenlenmesi gerekliliğini beraberinde getirmiştir. Ruby'nin çöp toplama (*garbage collection*) özelliği, bir zamanlayıcı geliştirilmesini zorlaştırmaktadır. Bu sebeple MIDI ve OSC mesajlarının zamanlayıcısı Erlang dilinde yazılmıştır. Zamanlanan mesajlar MIDI ve OSC formatları arasında dönüşüm yapabilen "osmid" yazılımınca işlenir ve gönderilir ("Sonic Pi Architecture", 2017; "Use of Erlang in Sonic Pi", 2018).

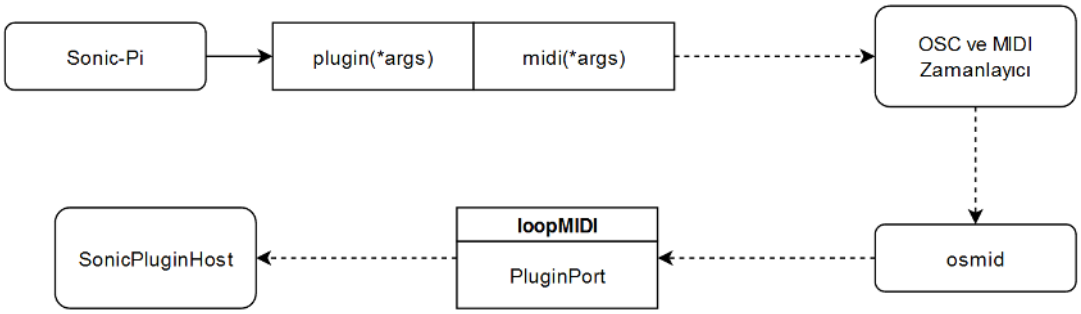
3.3.2 Eklentilere ilişkin komutların Sonic-Pi tarafından gönderilmesi



Şekil 3.8 Sonic-Pi'den gönderilen eklenti komutlarının işlemlenme şeması

Eklentilerin yüklenebilmesi, bağlanabilmesi ve parametrelerin değiştirilebilmesi, Sonic-Pi tarafından gönderilen ve belirli anahtarlara sahip mesajlar ile mümkün olmaktadır. Bu mesajlara dair işlemlenme şeması Şekil 3.8'de sunulmuştur. Eklenti komutlarının işlemlenmesinde Sonic-Pi'nin görevi, aslen komutları SonicPluginHost'un beklediği forma sokmaktan ibarettir. Bunun için SonicPluginHost ile paralel olarak üç fonksiyon tanımlanmıştır. Fonksiyonlar gerekli bilgileri kullanıcıdan fonksiyon parametresi olarak alır ve biçimlendirir. Biçimlendirme işlemi mesajın başına ilgili anahtarın koyulması ve bilgilerin belirli işaretçi karakterler kullanılarak birleştirilmesi aşamalarından meydana gelmektedir. Mesajlar biçimlendirildikten sonra Sonic-Pi içerisinde tanımlanmış olan UDPHandler vasıtasıyla SonicPluginHost'un dinleme noktası olan 54020 numaralı bağlantı noktasına gönderilir.

3.3.3 MIDI mesajlarının Sonic-Pi tarafından gönderilmesi



Şekil 3.9 Sonic-Pi'den gönderilecek MIDI mesajlarının işlemlenme şeması

Yüklenen enstrüman eklentilerinin kullanılabilmesi için, çalınması istenen MIDI mesajlarının Sonic-Pi'den SonicPluginHost'a doğru bir şekilde aktarılması gerekmektedir. Sonic-Pi'den gönderilen MIDI mesajları Şekil 3.9'da sunulan şemaya göre işlemlenmektedir. MIDI mesajlarının gönderilmesi eklenti komutlarının gönderilmesinden daha karmaşıktır. Çünkü eklenti komutları alındıkları anda işleme konurken, MIDI mesajları eserin ritmine uyacak şekilde işleme konmalıdır ve bu sebeple alındıktan sonra belirli bir süre beklenmelidir. MIDI mesajının gönderildiği cihazın bu ritmik yapıdan haberdar olması ve bu yapıya göre çalışacak bir zamanlayıcı barındırması beklenemez. Dolayısıyla bu mesajlara ilişkin bekleme süreleri, Sonic-Pi tarafından yönetilmektedir. Sonic-Pi'nin karmaşık bir zamanlama algoritması vardır (Aaron, Orchard ve Blackwell, 2014). Bu nedenle eklentilere gönderilecek MIDI mesajlarının yaratılacak yeni bir algoritmaya göre zamanlanması değil, Sonic-Pi'nin içerisindeki zamanlayıcı tarafından zamanlanması oldukça önemlidir. Bu sebeple Sonic-Pi'nin içindeki harici cihazlara MIDI mesajı göndermekte kullanılan midi() fonksiyonu kullanılmış, bu fonksiyonun yalnızca PluginPort'a mesaj göndermekte kullanılan bir uyarlaması oluşturulmuş ve plugin() olarak isimlendirilmiştir. Bu fonksiyon gönderilmesi istenen MIDI mesajına dair çeşitli bilgileri parametre olarak alır ve bu bilgileri barındıran bir OSC mesajı oluşturur. Bu OSC mesajı zamanlayıcıya gönderilir ve ritme uygun olarak tetiklenecek şekilde zamanlanır. Mesaj zamanlayıcıdan OSC mesajı olarak çıkar ve "osmid" yazılımına gönderilir. C++ ile yazılmış "osmid" yazılımı mesajı yüksek hızla MIDI'ye çevirir ve PluginPort'a gönderir. PluginPort'u dinlemekte olan SonicPluginHost MIDI mesajını alır ve işleme koyar.

4 BULGULAR

SonicPluginHost, mevcut sürümüyle çalışmaya yönelik belirlenen ihtiyaçların hepsini karşılamaktadır. Açık erişimli kaynak koda eklenen dosya yolu bilgisi ile belirtilen klasördeki eklentiler taranabilmekte ve her an yüklenmeye hazır halde tutulabilmektedir. Öte yandan yalnızca dinlemekle görevlendirilen ve sürekli olarak dinleme halinde bulunan bir iş parçacığı vasıtasıyla Sonic-Pi'den gelen komutlar her an, SonicPluginHost programının iş durumundan bağımsız olarak işleme sokulabilmekte, sesin işlemlenmesi esnasında dahi parametreler değiştirilebilmektedir. Ayrıca ses ve MIDI bağlantıları eklentilerin yüklenme sürecine dahil edilmiştir. Böylece kullanıcıdan alınacak hangi eklentinin nereye bağlanacağı bilgisi ile ayrı bir komuta gerek kalmaksızın bağlantılar tamamlanabilmektedir. Böylece çalışmanın başlangıç hedeflerine ulaştığı söylenebilmektedir.

Kullanım testleri 3.20GHz Intel Core i7-8700 işlemciyle ve 16 GB yüklü bellek (*RAM*) ile 64 bit tabanlı Windows 10 işletim sistemi üzerinde gerçekleştirilmiştir. Bu testlerde programın gerçek zamanlı ses sentezini aksatmayacak hızda çalıştığı ve herhangi bir çalışma zamanı hatası (*runtime error*) vermediği gözlemlenmiştir. SonicPluginHost'un açılması işlemi Sonic-Pi'nin açılış prosedürüne dahil edilmiştir. Sonic-Pi'nin değiştirilmemiş versiyonunun ortalama 10 saniye içerisinde, değiştirilmiş versiyon ve SonicPluginHost'un ortalama 27 saniye içerisinde açıldığı gözlemlenmiştir.

Programın kullanım testlerinde iki adet enstrüman (Tunefish4 ve Podolski) ve iki adet efekt (VerberateBasic2 ve Dystroyed) eklentisi kullanılmıştır. Kullanılan eklentiler ücretsiz ve basit arayüzlüdür. SonicPluginHost'un mevcut sürümünün Kontakt veya Reaktor gibi "eklenti editörü" eklentilerle başarılı bir şekilde çalışmadığı gözlemlenmiştir.

Kullanım testleri süresince Sonic-Pi'nin harici uygulamalara MIDI mesajı gönderme fonksiyonunun zaman zaman hatalı mesajlar gönderilebildiği tespit edilmiştir. Bu gözlem üç farklı eklenti hostu ile daha denenerek doğrulanmıştır.

MIDI mesajlarının aktarımı, yazılımın geliştirilme aşamasında karşılaşılan sorunlardan biri olmuştur. Bu sorunun sebebi Bölüm 3.3.3'te bahsedilen MIDI veri alışverişinin Windows vasıtasıyla gerçekleştirilmesidir. Sorunun çözümü için MIDI mesajlarının doğrudan osmid yazılımından alınması, UDP protokolü vasıtasıyla SonicPluginHost'a gönderilmesi ve SonicPluginHost içerisinde işlemlenmesi seçeneği düşünülmüştür. Ancak osmid yazılımının MIDI arabelleklerini (*buffer*) UDP yoluyla göndermeye uygun olmadığı anlaşılmıştır. Yazılım açık kaynaklı olduğu için

ihtiyaca göre değiştirilmesi mümkündür ancak bu durumun Sonic-Pi'nin ilerleyen sürümleri ile uyuşmazlık yaratacağı öngörülmüştür. Bu sebeple Sonic-Pi sistemine daha az müdahale gerektiren sanal MIDI bağlantı noktası çözümü tercih edilmiştir.

Yazılım geliştirilirken eklentilerin birbirlerine ve özel node'lara bağlanması işleminin özel bir komutla gerçekleştirilmesi öngörülmüştür. Ancak eklentilerin bağlanma şemalarının kategorileri içerisinde tutarlı olduğu gözlemlenmiştir. Test amaçlı kullanım süreçlerinde enstrüman eklentilerinin MidiInNode ve AudioOutNode noktalarına bağlanmasının, efekt eklentilerinin ise ses kaynağı herhangi bir node ile AudioOutNode noktalarına bağlanmasının sabit uygulamalar olduğu saptanmıştır. Bu sebeple, kullanıcıların bir eklentiyi çalışır hale getirmesi için girmeleri gereken kod miktarının mümkün olduğunca az tutulması adına, bağlantıların oluşturulmasının eklentilerin yüklenme süreçlerine dahil edilmesi uygun görülmüştür.

```
addFxPluginbyName invoked.
selectedplugin message is: Acon Digital Verberate Basic 2
Selected plugin index is: 5

addNode attempted. added plugin node: Acon Digital Verberate Basic 2
target processor: Zebra2
plugin sample rate: 48000
plugin block size: 1024
number of nodes in the graph:4

node0 :Midi Input (nodeID: 1)
node1 :Audio Output (nodeID: 2)
node2 :Zebra2 (nodeID: 101)
node3 :Acon Digital Verberate Basic 2 (nodeID: 102)

plugin node is not bypassed
plugin bus count:1
plugin 1st bus channel count:2
total num of input channels:2
total num of output channels:2
param no 1: Program
param no 2: Mix
param no 3: Bypass
Existing connection count:3
Existing target connection removed
Existing target connection removed
TargetNode channel 0 connection successful.
TargetNode channel 1 connection successful.
audInNode channel 0 connection successful.
audInNode channel 1 connection successful.
```

Şekil 4.1 SonicPluginHost komut satırı arayüzü

Yazılımın mimarisine dair ilk planlamalarda SonicPluginHost'un komut satırı arayüzünün (*Command Line Interface - CLI*) kullanıcıya sunulmaması öngörülmüş, yüklenebilecek eklentilere ve yüklü eklentilerin parametrelerine dair bilgilendirmelerin Sonic-Pi vasıtasıyla gerçekleştirilmesi tasarlanmıştır. Ancak test

amaçlı kullanımlar Sonic-Pi'deki mevcut bildirim panellerinin icra edilen komutlara dair sunduğu bilgilerin pek çok hatanın ayıklanmasında (*debugging*) oldukça işlevsel olduğunu göstermiştir. Bu sebeple hem tespit edilen ve yüklenebilecek eklentilerin, hem yüklenmiş eklentilerin çeşitli parametrelerinin hem de SonicPluginHost'a dair çalışma bilgilerinin komut satırı arayüzü ile (Şekil 4.1) kullanıcıya sunulmasına karar verilmiştir. Böylece kullanıcının hem bahsedilen bilgilere erişmesi, hem de olası sorunları tespit edebilmesi mümkün kılınmıştır. Ayrıca aslen geliştirme esnasında bir araç olarak kullanılan ses çıkış cihazı test sesi, programın kullanımında karşılaşılabilecek ses cihazlarına dair sorunların daha kolay tespit edilmesi amacıyla programın son haline dahil edilmiştir.

5 SONUÇ, TARTIŞMA VE ÖNERİLER

5.1 Sonuç ve Tartışma

Bu çalışmada canlı kodlamanın bir elektronik müzik aracı olarak yaygınlığının ve kullanılabilirliğinin artırılması hedeflenmiştir. Bu amaca yönelik olarak bilgisayar teknolojilerinden uzak insanların bile kolaylıkla öğrenebildiği ve kullanabildiği canlı kodlama uygulamalarından biri olan Sonic-Pi tercih edilmiştir. Sonic-Pi'nin bir elektronik müzik aracı olarak işlevinin artırılması için elektronik müzikte sıklıkla kullanılan yazılımlardan olan ses eklentilerini desteklemesinin oldukça faydalı olacağı çıkarımlanmıştır. Ses eklentilerini destekleyebilmesi için Sonic-Pi'ye yardımcı bir eklenti hostu programı olan SonicPluginhost geliştirilmiştir ve Sonic-Pi'ye bu programla iletişime geçip ses eklentilerini yönetebilmesini sağlayacak belirli araçlar eklenmiştir.

SonicPluginHost yazılımı vasıtasıyla, Sonic-Pi içerisinden ses eklentilerinin kontrol edilmesi mümkün hale gelmiştir. Sonic-Pi'nin açılması ile eşzamanlı olarak çalışmaya başlayan yazılım; belirlenen klasördeki eklentileri tarayıp tespit etme, belirlenen eklentileri içinde çalışacakları eklenti çizelgesine ekleme, eklentilerden elde edilen sesi çalınmak üzere ses cihazlarına yollama, eklentilere MIDI mesajı gönderme ve belirlenen eklentilerin parametrelerini değiştirme özelliklerine sahiptir. Yazılım, açık kaynak kodludur ve GNU GPL 3.0 ile lisanslanmıştır (https://github.com/ErenKaracan47/SonicPluginHost).

```
#load_pluginst "Zebra2"
#load_plugfx 101, "Acon Digital Verberate Basic 2"
live_loop :pod do
  param 101, "Cutoff", 0.3
  param 102, "Mix", 100
  sleep 1
  plugin 60, vel: 0.5, sustain: 1
  sleep 0.25
  plugin 60, vel: 0.5, sustain: 0.5
  sleep 1
  plugin 64, vel: 0.5, sustain: 0.5
  param 101, "Cutoff", 0.7
  sleep 0.5
  plugin 66, vel: 0.5, sustain: 2.5
  sleep 0.5
end
```

Şekil 5.1 Sonic-Pi'nin SonicPluginHost ile kullanımı

Bu yazılım ile birlikte Sonic-Pi'ye, performans esnasında ses eklentileri kullanabilme özelliği eklenmiştir (Şekil 5.1). Kullanıcıya tam yetki veren GNU GPL lisansı dolayısıyla üçüncü kişiler bu yazılımı indirebilir, çoğaltabilir, değiştirebilir ve dağıtabilir. Dolayısıyla yazılımın kişilerin beklentilerine göre modifiye edilmesi yahut başka projelerde bir modül olarak kullanılabilmesi mümkündür.

Bölüm 2.3'te bahsedildiği gibi, mevcut popüler elektronik müzik pratiklerinin içerisinde ses eklentilerinin oldukça önemli bir yeri vardır. Bu sebeple canlı kodlamanın popülerleşmesi ve kodlama bilmeyen müzisyenlere hitap edebilmesi için canlı kodlama programlarının müzisyenlerin alıştığı eklentilere yer verebilmesi oldukça önemlidir. Öte yandan sağlanan eklenti desteği ile Sonic-Pi'nin ses yelpazesi oldukça genişletilmiştir. Böylece canlı kodlamanın elektronik müzik veya programlamayla ilgili herhangi bir geçmişi olmayan insanlar için de daha cazip hale getirildiği söylenebilir. Bir başka sonuç olarak, bu şekilde Sonic-Pi'nin interaktif oyun müziği, oyun sesi tasarımı, web sitesi ses tasarımı gibi farklı alanlarda kullanılabilme ihtimali artırılmıştır.

Bölüm 2.1.3'te de bahsedildiği gibi, canlı kodlamanın elektronik müziğin performansına ilişkin bazı sorunlara çözüm sağlayabileceği öngörülmektedir. Ancak bu öngörünün denenebilmesi için, öncelikle canlı kodlamanın daha fazla kişi tarafından icra edilmesi ve daha çok ürünün ortaya konması gerekmektedir. Sunulan imkanların canlı kodlamanın daha büyük yaygınlık kazanmasında ve dolayısıyla elektronik müzik performansı için önemli bir vasıta olarak görülmesinde bir adım teşkil edeceği düşünülmektedir.

5.2 Öneriler

Bu tez ile geliştirilen yazılım, Sonic-Pi uygulamasının ses eklentileri ile entegrasyonunda bir prototip teşkil etmektedir. Dolayısıyla her ne kadar bahsedilen işlevi sağlasa da, kullanıcı odaklı bazı eklemeler programın geliştirilme amacını gerçekleştirmesinde oldukça faydalı olacaktır.

Uygulama mevcut sürümünde yalnızca tek bir MIDI cihazı kullanabilmektedir. Bu cihaz sayısının çizelgeye eklenecek MIDI kanalı ayrıştırıcıları vasıtasıyla artırılması, eş zamanlı olarak farklı MIDI komutları ile çalıştırılabilecek maksimum enstrüman eklentisi sayısını da artıracaktır.

Yazılımın kullanacağı ses cihazı ve içerisinde arama yapacağı eklenti klasörü, mevcut sürümde kaynak kodu içerisinden belirlenmektedir. Bu tercihlerin

düzenlenmesi amacıyla tasarlanacak bir arayüz, kullanıcı deneyimini kolaylaştıracaktır.

Yazılım mevcut olarak yalnızca Windows işletim sisteminde VST3 uzantılı eklentilerle denenmiştir. Ancak JUCE Framework vasıtasıyla yazıldığı için AU uzantılı eklentilerin desteklenmesi ve MacOS platformunda kullanılır hale getirilmesi oldukça kolay olacaktır. Bu amaçla yazılımın MacOS işletim sisteminde AU eklentilerle denenmesi ve ilgili ayarlamaların yapılması, hitap edilen kullanıcı havuzunu genişletecektir.

Mevcut sürümde yüklenen eklentilerin tüm parametreleri sunulsa da, kullanıcıların eklentilerin grafik arayüzleri vasıtasıyla bazı ayarlamalar yapmaları gerekebilmektedir. Bu nedenle yüklenen eklentilerin grafik arayüzlerine erişim sağlayacak fonksiyonlar, kullanım kalitesini artırabilir.

Yazılım mevcut sürümde sanal MIDI bağlantı noktası olarak harici yazılımlara ihtiyaç duymaktadır. Harici yazılımların kullanımı performansa yönelik kayda değer bir dezavantaja sebep olmasa da mevcut durumda kullanıcıya yüklenen bir sorumluluk, dolayısıyla bir kullanım zorluğu teşkil etmektedir. SonicPluginHost ve Sonic-Pi arasındaki iletişim için münhasıran yazılacak sanal MIDI bağlantı noktaları ile kullanıcı harici yazılım yükleme sorumluluğundan kurtarılabilir.

Birçok ses eklentisi çok fazla sayıda parametre ve dolayısıyla çok geniş bir ses yelpazesi sunmaktadır. Bu detay seviyesi yazılımın kullanım alanını artırarak bir avantaj sağlasa da istenen sese ulaşmak için çok fazla parametrenin değiştirilmesi gerekliliğiyle bir kullanım zorluğuna da sebep olmaktadır. Bu sorunun üstesinden gelmek amacıyla birçok eklenti *preset* ya da *program* olarak adlandırılan hazır parametre kurulumlarını desteklemektedir. Böylece kullanıcılar kendilerinin ya da başka kullanıcıların kaydetmiş olduğu sesleri kolaylıkla tekrar çağırabilmekte ve kullanabilmektedir. Bu preset sistemine ilişkin desteğin SonicPluginHost'a dahil edilmesiyle kullanıcıların kolaylıkla ulaşabileceği ses çeşitliliği çok büyük oranda artırılabilir.

Bahsedilen geliştirmeler ile kullanıcıların SonicPluginHost çizelgesine ekleyeceği eklenti sayısı, mevcut eklenti kimliklendirme sistemini kullanışsız kılacak kadar artabilecektir. Bu durumda yüklenen eklentilerin ve aralarındaki bağlantıların ifade edildiği mevcut eklenti sinyal akışını gösterecek bir şema ile kullanım kolaylığı sağlanabilir.

KAYNAKÇA

3rd Party Developer. (t.y.).*Https://www.steinberg.net/*. Steinberg Media Technologies GmbH. 2 Ağustos 2020 tarihinde https://www.steinberg.net/en/company/developers.html adresinden erişildi.

Aaron, S. (2016). *Code Music with Sonic Pi.pdf*. Cambridge: Raspberry Pi.

Aaron, S. ve Blackwell, A. F. (2013). From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages. *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design* içinde (ss. 35-46). FARM, sunulmuş bildiri, Boston, Massachusetts, USA: ACM Press. doi:10.1145/2505341.2505346

Aaron, S. ve Judge, J. (2012). Snapshots: New Possibilities for Social Digital Music-Making arising from the Storage of History. *Non-Cochlear Sound: Proceedings of the 38th International Computer Music Conference, ICMC 2012, Ljubljana, Slovenia, September 9-14, 2012* içinde (ss. 228-235). ICMC, sunulmuş bildiri, Ljubljana, Slovenya: Michigan Publishing. http://hdl.handle.net/2027/spo.bbp2372.2012.041 adresinden erişildi.

Aaron, S., Orchard, D. ve Blackwell, A. F. (2014). Temporal semantics for a live coding language. *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design—FARM '14* içinde (ss. 37-47). the 2nd ACM SIGPLAN international workshop, sunulmuş bildiri, Gothenburg, Sweden: ACM Press. doi:10.1145/2633638.2633648

Algorave. (t.y.).*Algorave*. 21 Temmuz 2020 tarihinde https://algorave.com/ adresinden erişildi.

Alliance Partner Program | AAX Connectivity Toolkit | Avid. (t.y.). 2 Ağustos 2020 tarihinde https://www.avid.com/alliance-partner-program/aax-connectivity-toolkit adresinden erişildi.

Artut, S. (2016). Live Coding for Incorporating Self Expression. *International Conference on Social Sciences and Humanities "Challenges: From Diversity to Synergy"* içinde (ss. 625-630). ICSSH, sunulmuş bildiri, Skopje, Macedonia.

AudioUnit | Apple Developer Documentation. (t.y.). 2 Ağustos 2020 tarihinde https://developer.apple.com/documentation/audiounit adresinden erişildi.

Brown, A. R. (2016). Performing with the other: The relationship of musician and machine in live coding. *International Journal of Performance Arts and Digital Media*, *12*(2), 179-186. doi:10.1080/14794713.2016.1227595

Brown, C. ve Bischoff, J. (2002). Indigenous to the Net: Early Network Music Bands in the San Francisco Bay Area. 21 Temmuz 2020 tarihinde http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html adresinden erişildi.

Burgess, R. J. (2014). *The History of Music Production*. New York: Oxford University Press.

Cascone, K. (2003). Grain, Sequence, System: Three Levels of Reception in the Performance of Laptop Music. *Contemporary Music Review*, *22*(4), 101-104. doi:10.1080/0749446032000156900

Cocker, E. (2016). Performing thinking in action: The meletē of live coding. *International Journal of Performance Arts and Digital Media*, *12*(2), 102-116. doi:10.1080/14794713.2016.1227597

Collins, M. (2012). *A Professional Guide to Audio Plug-ins and Virtual Instruments*. Focal Press. doi:10.4324/9780080497846

Collins, N. (2003). Generative Music and Laptop Performance. *Contemporary Music Review*, *22*(4), 67-79. doi:10.1080/0749446032000156919

Collins, N. ve McLean, A. (2014). Algorave: Live Performance of Algorithmic Electronic Dance Music. *Proceedings Of The International Conference On New Interfaces For Musical Expression 2014* içinde (ss. 355-358). New Interfaces For Musical Expression, sunulmuş bildiri, Londra.

Collins, N., McLEAN, A., Rohrhuber, J. ve Ward, A. (2003). Live coding in laptop performance. *Organised Sound*, *8*(3), 321-330. doi:10.1017/S135577180300030X

Dannenberg, R. B., Gold, N. E., Liang, D. ve Xia, G. (2014). Methods and Prospects for Human–Computer Performance of Popular Music. *Computer Music Journal*, *38*(2), 36-50. doi:10.1162/COMJ_a_00238

Doornbusch, P. (2004). Computer Sound Synthesis in 1951: The Music of CSIRAC. *Computer Music Journal*, *28*(1), 10-25. doi:10.1162/014892604322970616

Doornbusch, P. (2009). Early Hardware and Early Ideas in Computer Music: Their Development and Their Current Forms. *The Oxford Handbook of Computer Music* içinde (ss. 44-84). New York: Oxford University Press.

Evans, H. (1992). Out of sight, out of mind. *Out Of Sight, Out of Mind: An Analysis of Rave Culture*. 21 Temmuz 2020 tarihinde http://hehe.org.free.fr/hehe/texte/rave/ adresinden erişildi.

Flašar, M. (2016). Listening with the Eyes: Remarks on Live Coding Performance. *MAP - Media / Archive / Performance*, *2016*(7), 7.

Goudard, V. ve Muller, R. (2003). *Real-time Audio Plugin Architectures* (s. 24). IRCAM.

Herrera Machuca, M., Lobato Cardoso, J. A., Torres Cerro, J. A. ve Lomelí Bravo, F. J. (2016). Live coding for all: Three creative approaches to live coding for

non-programmers. *International Journal of Performance Arts and Digital Media*, *12*(2), 187-194. doi:10.1080/14794713.2016.1227598

International Conference on Live Coding—Home. (t.y.). 3 Ağustos 2020 tarihinde https://iclc.toplap.org/ adresinden erişildi.

In_thread. (t.y.).*In_thread*. 24 Temmuz 2020 tarihinde https://in-thread.sonic-pi.net/ adresinden erişildi.

Julian, R. ve de Campo, A. (2009a). Improvising Formalisation—Conversational Programming and Live Coding. *New Computational Paradigms for Computer Music* içinde (ss. 113-124). Delatour.

Julian, R. ve de Campo, A. (2009b). Improvising Formalisation—Conversational Programming and Live Coding. *New Computational Paradigms for Computer Music* içinde (ss. 113-124). Delatour.

Kaiser, J. G. (2013). *Improvising Technology: Configuring Identities and Interfaces in Contemporary Electro-acoustic Music*. https://escholarship.org/uc/item/6bg2c417 adresinden erişildi.

Keislar, D. (2009). A Historical View of Computer Music Technology. *The Oxford Handbook of Computer Music* içinde (ss. 11-43). New York: Oxford University Press.

Kjus, Y. ve Danielsen, A. (2016). Live mediation: Performing concerts using studio technology. *Popular Music*, *35*(3), 320-337. doi:10.1017/S0261143016000568

Knotts, S. (2016). *Exploring Virtuosity in Live Coding in Flow*. International Conference on Live Coding 2016, sunulmuş bildiri, Hamilton, Canada. https://macdrive.mcmaster.ca/seafhttp/files/a121034d-2173-43eb-92a1-52efa398f990/iclc2016-knotts.pdf adresinden erişildi.

Lawson, S. ve Smith, R. R. (2019). What Am I Looking At?: An Approach to Describing the Projected Image in Live-Coding Performance. *Proceedings of the International Conference on Live Coding* içinde . ICLC, sunulmuş bildiri, Madrid: Zenodo. doi:10.5281/ZENODO.3946290

Magnusson, T. (2011a). Algorithms as Scores: Coding Live Music. *Leonardo Music Journal*, *21*(21), 19-23. doi:10.1162/LMJ_a_00056

Magnusson, T. (2011b). The IXI Lang: A SuperCollider Parasite for Live Coding. *Proceedings of the 2011 International Computer Music Conference* içinde (C. 2011, ss. 503-506). International Computer Music Conference, sunulmuş bildiri, Huddersfield, UK: Michigan Publishing. http://hdl.handle.net/2027/spo.bbp2372.2011.101 adresinden erişildi.

Magnusson, T. (2014). Herding Cats: Observing Live Coding in the Wild. *Computer Music Journal*, *38*(1), 8-16. doi:10.1162/COMJ_a_00216

ManifestoDraft—Toplap. (t.y.). 21 Temmuz 2020 tarihinde https://toplap.org/wiki/ManifestoDraft adresinden erişildi.

Manning, P. (2004). *Electronic and computer music* (Rev. and expanded ed.). Oxford ; New York: Oxford University Press.

Manning, P. (2009). Sound Synthesis Using Computers. *The Oxford Handbook of Computer Music* içinde (ss. 85-105). New York: Oxford University Press.

Mathews, M. V. (1963). The Digital Computer as a Musical Instrument. *Science, New Series*, *142*(3592), 553-557.

Mazierska, E. (2018). Improvisation in Electronic Music—The Case of Vienna Electronica. *Open Cultural Studies*, *2*(1), 553-561. doi:10.1515/culture-2018-0050

McCartney, J. (1996). SuperCollider—A New Real-time Sound Synthesis Engine. *Proceedings of the 1996 International Computer Music Conference* içinde

(C. 1996, ss. 257-258). International Computer Music Conference, sunulmuş bildiri, Hong Kong: Michigan Publishing. http://quod.lib.umich.edu/i/icmc/bbp2372.1996 adresinden erişildi.

McCartney, J. (1998). Continued Evolution of the SuperCollider Real Time Synthesis Environment. *Proceedings of the 1998 International Computer Music Conference* içinde . ICMC, sunulmuş bildiri, Michigan, USA: Michigan Publishing. http://hdl.handle.net/2027/spo.bbp2372.1998.262 adresinden erişildi.

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, *26*(4), 61-68. doi:10.1162/014892602320991383

McLean, A. (2003). Angry. *Cream12*. 25 Temmuz 2020 tarihinde https://jesis.home.xs4all.nl/artcriticism/cream/back_issues/cream12.html adresinden erişildi.

McLean, A. (2008). Live Coding for Free. *Floss + Art* içinde (ss. 224-231). Poitiers, Fransa.

McLean, A. (2011). *Artist-Programmers and Programming Languages for the Arts*. University of London, London.

McLean, A. (2014). Making programming languages to dance to: Live coding with tidal. *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design—FARM '14* içinde (ss. 63-70). the 2nd ACM SIGPLAN international workshop, sunulmuş bildiri, Gothenburg, Sweden: ACM Press. doi:10.1145/2633638.2633647

McLean, A., Griffiths, D., Collins, N. ve Wiggins, G. (2010). Visualisation of Live Code. *Proceedings of the 2010 International Conference on Electronic Visualisation and the Arts* içinde , EVA'10 (ss. 26–30). Swindon, GBR: BCS Learning & Development Ltd.

Mehackit Creative programming workshop with Sonic Pi. (t.y.). 26 Temmuz 2020 tarihinde http://sonic-pi.mehackit.org/ adresinden erişildi.

Moura da Silva, P. M., Cavalcante Mattos, C. L. ve de Souza Junior, A. H. (2019). Audio Plugin Recommendation Systems for Music Production. *8th Brazilian Conference on Intelligent Systems (BRACIS)* içinde (ss. 854-859). BRACIS, sunulmuş bildiri, Salvador, Brazil: IEEE. doi:10.1109/BRACIS.2019.00152

Musikinformatik/SuperDirt. (2020). SuperCollider, musikinformatik. https://github.com/musikinformatik/SuperDirt adresinden erişildi.

Nash, C. ve Blackwell, A. (2011). Tracking Virtuosity and Flow in Computer Music. *Proceedings of the 2011 International Computer Music Conference* içinde (C. 2011, ss. 575-582). ICMC, sunulmuş bildiri, Huddersfield, UK: Michigan Publishing. http://hdl.handle.net/2027/spo.bbp2372.2011.116 adresinden erişildi.

Nilson, C. (2007). Live Coding Practice. *Proceedings of the 7th International Conference on New Interfaces for Musical Expression* içinde . NIME, sunulmuş bildiri, New York, New York: ACM Press. doi:10.1145/1279740.1279760

Pejrolo, A. ve Metcalfe, S. B. (2017). *Creating sounds from scratch: A practical guide to music synthesis for producers and composers*. New York, NY: Oxford University Press.

Peters, N., Choi, J. ve Lei, H. (2012). Matching Artificial Reverb Settings to Unknown Room Recordings: A Recommendation System for Reverb Plugins. *Proceedings of the 133th Audio Engineering Society Convention* içinde (s. 9). AES, sunulmuş bildiri, San Francisco, California.

Pirkle, W. C. (2019). *Designing Audio Effect Plugins in C++. For AAX, AU and VST3 with DSP Theory* (2. bs.). New York: Routledge.

Puckette, M. (1991). Something Digital. *Computer Music Journal*, *15*(4), 65. doi:10.2307/3681075

Raspberry Pi Foundation—About Us. (t.y.).*Raspberry Pi*. https://www.raspberrypi.org/about/ adresinden erişildi.

Roads, C. (1986). The Second STEIM Symposium on Interactive Composition in Live Electronic Music. *Computer Music Journal*, *10*(2), 44. doi:10.2307/3679484

Rohrhuber, J., de Campo, A., Wieser, R., van Kampen, J.-K., Ho, E. ve Hölzl, H. (2007). Purloined letters and distributed persons. *Music in the Global Village Conference (Budapest)* içinde .

Sayer, T. (2015). Cognition and Improvisation: Some Implications for Live Coding. *Proceedings of the First International Conference on Live Coding* içinde (ss. 87–92). International Conference on Live Coding, sunulmuş bildiri, Leeds, UK: ICSRiM, University of Leeds. doi:10.5281/zenodo.19328

Sayer, T. (2016). Cognitive load and live coding: A comparison with improvisation using traditional instruments. *International Journal of Performance Arts and Digital Media*, *12*(2), 129-138. doi:10.1080/14794713.2016.1227603

Short History of Computer Music. (t.y.). 25 Temmuz 2020 tarihinde http://artsites.ucsc.edu/EMS/Music/equipment/computers/history/history.html adresinden erişildi.

Sonic Pi Architecture. (2017).*In_thread*. 27 Temmuz 2020 tarihinde https://in-thread.sonic-pi.net/t/sonic-pi-architecture/329 adresinden erişildi.

Sorensen, A. (2005). Impromptu—An Interactive Programming Environment for Composition and Performance. *Proceedings of the Australasian Computer Music Conference 2009* içinde . Australasian Computer Music Conference, sunulmuş bildiri, Brisbane: Queensland University of Technology.

Sorensen, A. C. (2018). *Extempore: The design, implementation and application of a cyber-physical programming language*. (Yayımlanmamış phd thesis). College of Engineering and Computer Science, The Australian National University.

Stasis, S., Jillings, N., Enderby, S. ve Stables, R. (2017). Audio Processing Chain Recommendation. *Proceedings of the 20th International Conference on Digital Audio Effects* içinde (ss. 103-109). DAFx, sunulmuş bildiri, Edinburgh, UK: University of Edinburgh.

SuperCollider. (t.y.). 13 Ağustos 2020 tarihinde https://supercollider.github.io/ adresinden erişildi.

Technologies. (t.y.).*Https://www.steinberg.net/*. Steinberg Media Technologies GmbH. 2 Ağustos 2020 tarihinde https://www.steinberg.net/en/company/technologies.html adresinden erişildi.

The latest round of Google Open Source Peer Bonus winners. (t.y.).*Google Open Source Blog*. https://opensource.googleblog.com/2017/03/the-latest-round-of-google-open-source.html adresinden erişildi.

TOPLAP nodes. (2016, 12 Mayıs).*TOPLAP*. 21 Temmuz 2020 tarihinde https://toplap.org/nodes/ adresinden erişildi.

Turkle, S. ve Papert, S. (1990). Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs*, *16*(1), 128-157.

Upton, E. (2020, 28 Mayıs). 8GB Raspberry Pi 4 on sale now at $75. *Raspberry Pi*. https://www.raspberrypi.org/blog/8gb-raspberry-pi-4-on-sale-now-at-75/ adresinden erişildi.

Use of Erlang in Sonic Pi. (2018, 2 Şubat).*In_thread*. 24 Temmuz 2020 tarihinde https://in-thread.sonic-pi.net/t/use-of-erlang-in-sonic-pi/701/2 adresinden erişildi.

Wang, G. ve Cook, P. R. (2003). ChucK: A Concurrent, On-the-fly, Audio Programming Language. *Proceedings of the 2003 International Computer Music Conference* içinde (C. 2003, ss. 1-8). ICMC, sunulmuş bildiri, Singapore: Michigan Publishing. http://hdl.handle.net/2027/spo.bbp2372.2003.055 adresinden erişildi.

Wang, G. ve Cook, P. R. (2004). On-the-Fly Programming: Using Code as an Expressive Musical Instrument. *Proceedings of the 2004 Conference on New Interfaces for Musical Expression* içinde , NIME '04 (ss. 138–143). SGP: National University of Singapore.

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N. ve Alexander, A. (2004). Live Algorithm Programming and a Temporary Organisation for its Promotion. *Proceedings of the README Software Art Conference* içinde (C. 289, s. 290). read_me --- Software Art and Cultures, sunulmuş bildiri, Århus, Denmark: Digital Aesthetics Research Centre, University of Aarhus.

Weingarten, C. R. (2016, 23 Mayıs). Moogfest 2016: Was It Actually the Future of Music? *Rolling Stone*. https://www.rollingstone.com/music/music-live-reviews/moogfest-2016-was-it-actually-the-future-of-music-58300/ adresinden erişildi.

Zmölnig, I. ve Eckel, G. (2007). Live coding: An overview. *International Computer Music Conference*, 295-298.

EKLER

EK A Main.cpp içeriği

```
#include "F:\VSProj\SonicPluginHost\Builds\VisualStudio2019\PluginHost.h"
#include <Windows.h>

class PluginHostApp : public JUCEApplicationBase

{
public:
	PluginHostApp() {}
	~PluginHostApp() {}

	const String getApplicationName() override
	{
		return "PluginHost";
	}

	const String getApplicationVersion() override
	{
		return "1.0";
	}

	bool moreThanOneInstanceAllowed() override
	{
		return false;
	}

	void initialise(const String& cmdLineParam) override
	{
		std::cout << "initialized" << std::endl;
	}

	void shutdown() override
	{

	}

	void anotherInstanceStarted(const String& cmdLineParam) override
	{
		std::cout << "Only one instance allowed" << std::endl;
	}

	void systemRequestedQuit() override
	{

	}

	void suspended() override
	{

	}

	void resumed() override
	{

	}
```

EK A (devam) Main.cpp içeriği

```
void unhandledException(const std::exception*, const String& sourceFilename,
int lineNum) override
	{
		std::cout << "Encountered an unhandled exception." << std::endl;
	}

	void memoryWarningReceived() override
	{
		std::cout << "OS memory warning recieved." << std::endl;
	}
private:

	PluginHost host;

};

START_JUCE_APPLICATION(PluginHostApp)
```

EK B PluginHost.h içeriği

```
#pragma once
#include "JuceHeader.h"
#include "PluginHostGraph.h"
#include "UDPHandler.h"
#include "PluginManagementThread.h"

class PluginHost {

public:

	PluginHost();
	~PluginHost();

	PluginHostGraph hostGraph;



private:

	AudioDeviceManager deviceManager;
	AudioProcessorPlayer graphPlayer;
	AudioPluginFormatManager formatManager;
	KnownPluginList knownPluginList;
	Array<PluginDescription> pluginDescriptions;

	UDPHandler udpin;
	UDPHandler udpout;
	PluginManagementThread pmthread;

	void deviceManagerInit();
	void init();
	void scanAndAddPlugins();
	void sendPluginNum();
	void coutPluginNames();
	void startPMThread();

};
```

EK C PluginHost.cpp içeriği

```
#include "PluginHost.h"

PluginHost::PluginHost()
	: deviceManager(),
	  hostGraph(formatManager, knownPluginList),
	  udpin(UDPHandler(true, 54010)),
	  udpout(UDPHandler(false, 54000)),
	  pmthread(hostGraph)
{

	init();
	scanAndAddPlugins();
}

PluginHost::~PluginHost() {

}

void PluginHost::deviceManagerInit() {

	auto& devicetypes = deviceManager.getAvailableDeviceTypes();
	for (int i = 0; i < devicetypes.size(); i++) {
		std::cout << "availible device: " <<
		devicetypes[i]->getTypeName() << std::endl;
	}

	auto s = deviceManager.initialise(0, 2, nullptr, true);
	std::cout << "current device: " <<
	deviceManager.getCurrentAudioDeviceType() << std::endl;

	auto setup = deviceManager.getAudioDeviceSetup();
	setup.bufferSize = 1024;
	deviceManager.setAudioDeviceSetup(setup, true);
	deviceManager.playTestSound();
	std::cout << "device sample rate: " <<
	deviceManager.getCurrentAudioDevice()->getCurrentSampleRate() << std::endl;
	std::cout << "device block size: " <<
	deviceManager.getCurrentAudioDevice()->getCurrentBufferSizeSamples() <<
	std::endl;


	auto samplerates =
	deviceManager.getCurrentAudioDevice()->getAvailableSampleRates();

	for (int i = 0; i < samplerates.size(); i++) {
		std::cout << "availible sample rate: " << samplerates[i] << std::endl;
	}

	if (s.isNotEmpty()) {
		std::cout << "Error while initializing Device Manager." << std::endl;
		std::cout << s << std::endl;
	}

	deviceManager.addAudioCallback(&graphPlayer);
```

EK C (devam) PluginHost.cpp içeriği

```
	for (int i = 0; i < samplerates.size(); i++) {
		std::cout << "availible sample rate: " << samplerates[i] <<
		std::endl;
	}

	if (s.isNotEmpty()) {
		std::cout << "Error while initializing Device Manager." <<
		std::endl;
		std::cout << s << std::endl;
	}

	deviceManager.addAudioCallback(&graphPlayer);

	auto mididevicelist = MidiInput::getAvailableDevices();
	MidiDeviceInfo pluginportdevice;
	std::cout << "Availible midi device count:" << mididevicelist.size() <<
	std::endl;
	for (int i = 0; i < mididevicelist.size(); i++) {
		std::cout << "Availible midi device:" << mididevicelist[i].name <<
		std::endl;
		if (mididevicelist[i].name.equalsIgnoreCase("PluginPort")) {
			pluginportdevice = mididevicelist[i];
		}
	}

	deviceManager.setMidiInputDeviceEnabled(mididevicelist[0].identifier,
                                            true);
	std::cout << "Enabling midi device:" << mididevicelist[0].name <<
	std::endl;
	if (deviceManager.isMidiInputDeviceEnabled(mididevicelist[0].identifier))
	{
		std::cout << "Successfully enabled." << std::endl;
	}
	else {
		std::cout << "Device could not be enabled." << std::endl;
	}

	deviceManager.addMidiInputDeviceCallback(pluginportdevice.identifier,
                                             &graphPlayer);
}

void PluginHost::init() {

	deviceManagerInit();

	graphPlayer.setProcessor(&hostGraph.graph);

	std::cout << "name of current processor to play: " <<
	graphPlayer.getCurrentProcessor()->getName() << std::endl;

	formatManager.addDefaultFormats();
	auto format = formatManager.getFormat(0);
	auto formatName = format->getName();

	std::cout << "\nAvailible format count: " <<
	formatManager.getNumFormats() << "    Availible format:" << formatName <<
	"\n" << std::endl;
}
```

EK C (devam) PluginHost.cpp içeriği

```
void PluginHost::scanAndAddPlugins() {

	String pathName = "F:\\VST3";
	auto searchPath = FileSearchPath(pathName);
	auto pedalFl = File("F:\\VST3");

	auto dirScanner = new PluginDirectoryScanner(knownPluginList,
                                                 *formatManager.getFormat(0),
                                                  searchPath,
                                                  true,
                                                  pedalFl);
	bool scantrue = true;
	String beingscanned;
	while (scantrue)
	{
		std::cout << "Scanning..." << std::endl;
		std::cout << dirScanner->getNextPluginFileThatWillBeScanned() <<
		std::endl;
		scantrue = dirScanner->scanNextFile(true, beingscanned);
	}
	std::cout << "\n" << std::endl;

	pluginDescriptions = knownPluginList.getTypes();

	for (int i = 0; i < pluginDescriptions.size(); i++) {
		std::cout << "Loaded plugin:" << pluginDescriptions[i].name <<
		std::endl;
	}

	coutPluginNames();
	startPMThread();
}

void PluginHost::sendPluginNum() {

	std::cout << "Sending number of possible plugins..." << std::endl;
	String confirmmessage;
	udpout.UDPSend(String(knownPluginList.getNumTypes()));
	while (confirmmessage != "count received")
	{
		std::cout << "Awaiting confirmation" << std::endl;
		confirmmessage = udpin.UDPReceive();
	}
}

void PluginHost::coutPluginNames() {
	for (int i = 0;i < knownPluginList.getNumTypes(); i++)
	{
		std::cout << "Availible plugin: " << pluginDescriptions[i].name <<
std::endl;
		String msg = "Availible plugin: ";
	}
}

void PluginHost::startPMThread() {
	pmthread.startThread();
}
```

EK D PluginHostGraph.h içeriği

```
#pragma once

#include "JuceHeader.h"

class PluginHostGraph {

public:

	PluginHostGraph(AudioPluginFormatManager&, KnownPluginList&);
	~PluginHostGraph();



	AudioProcessorGraph graph;
	void addInstPlugin(const PluginDescription& desc);
	void addInstPluginByName(String pluginname);
	void addFxPlugin(const PluginDescription& desc,
			const AudioProcessorGraph::NodeID& target);
	void addFxPluginByName(String pluginname);
	void changeParamByName(String paramchange);

	std::unique_ptr<AudioProcessorGraph::AudioGraphIOProcessor>
midiInProcessor;
	std::unique_ptr<AudioProcessorGraph::AudioGraphIOProcessor>
audioOutProcessor;

private:

	AudioPluginFormatManager& formatManager;
	KnownPluginList& kpList;

};
```

EK E PluginHostGraph.cpp içeriği

```
#include "PluginHostGraph.h"

PluginHostGraph::PluginHostGraph(AudioPluginFormatManager& fm, KnownPluginList&
kp)
                    :formatManager(fm),
                     kpList(kp)
{
      audioOutProcessor =
std::make_unique<AudioProcessorGraph::AudioGraphIOProcessor>
(AudioProcessorGraph::AudioGraphIOProcessor::audioOutputNode);

      midiInProcessor =
std::make_unique<AudioProcessorGraph::AudioGraphIOProcessor>
(AudioProcessorGraph::AudioGraphIOProcessor::midiInputNode);

      auto midiInNode = graph.addNode(std::move(midiInProcessor));
      auto audioOutNode = graph.addNode(std::move(audioOutProcessor));

}

PluginHostGraph::~PluginHostGraph() {}

void PluginHostGraph::addInstPlugin(const PluginDescription& desc) {

      String err = String();
      auto instance = formatManager.createPluginInstance(desc,
                                                         graph.getSampleRate(),
                                                         graph.getBlockSize(),
                                                         err);


      instance->enableAllBuses();
      instance->prepareToPlay(graph.getSampleRate(), graph.getBlockSize());
      auto pluginnode = graph.addNode(std::move(instance),
AudioProcessorGraph::NodeID((graph.getNumNodes() + 99)));

      std::cout << "\naddNode attempted. added plugin node: " <<
      pluginnode->getProcessor()->getName() << std::endl;
      std::cout << "plugin sample rate: " <<
      pluginnode->getProcessor()->getSampleRate() << std::endl;
      std::cout << "plugin block size: " <<
      pluginnode->getProcessor()->getBlockSize() << std::endl;
      std::cout << "number of nodes in the graph:" << graph.getNumNodes() <<
      "\n" << std::endl;

      for (int i = 0; i < graph.getNumNodes(); i++) {
            std::cout << "node" << i << " :" <<
            graph.getNode(i)->getProcessor()->getName() << " (nodeID: " <<
            graph.getNode(i)->nodeID.uid << ")" << std::endl;
      }

      if (pluginnode->isBypassed()) {
            std::cout << "\nplugin node is bypassed" << std::endl;
      }
      else {
            std::cout << "\nplugin node is not bypassed" << std::endl;
      }
```

EK E (devam) PluginHostGraph.cpp içeriği

```
std::cout << "plugin bus count:" <<
pluginnode->getProcessor()->getBusCount(true) << std::endl;
std::cout << "plugin 1st bus channel count:" <<
pluginnode->getProcessor()->getChannelCountOfBus(true, 0) << std::endl;
std::cout << "total num of input channels:" <<
pluginnode->getProcessor()->getTotalNumInputChannels() << std::endl;
std::cout << "total num of output channels:" <<
pluginnode->getProcessor()->getTotalNumOutputChannels() << std::endl;

auto paramlist = pluginnode->getProcessor()->getParameters();
for (int i = 0; i < paramlist.size(); i++) {
      std::cout << "param no " << (i + 1) << ": " <<
      paramlist[i]->getName(20) << std::endl;
}


AudioProcessorGraph::NodeAndChannel sourcenodechannel1;
AudioProcessorGraph::NodeAndChannel destnodechannel1;
AudioProcessorGraph::NodeAndChannel sourcenodechannel2;
AudioProcessorGraph::NodeAndChannel destnodechannel2;

sourcenodechannel1.nodeID = AudioProcessorGraph::NodeID(1);
sourcenodechannel1.channelIndex = AudioProcessorGraph::midiChannelIndex;
destnodechannel1.nodeID = pluginnode->nodeID;
destnodechannel1.channelIndex = AudioProcessorGraph::midiChannelIndex;

AudioProcessorGraph::Connection midiconnection =
AudioProcessorGraph::Connection(sourcenodechannel1, destnodechannel1);
bool midiconfirm = graph.addConnection(midiconnection);
if (midiconfirm) {
      std::cout << "midiInNode connection successful." << std::endl;
}
else {
      std::cout << "midiInNode connection failed." << std::endl;
}



sourcenodechannel1.nodeID = pluginnode->nodeID;
sourcenodechannel1.channelIndex = 0;
destnodechannel1.nodeID = AudioProcessorGraph::NodeID(2);
destnodechannel1.channelIndex = 0;
AudioProcessorGraph::Connection aud1connection =
AudioProcessorGraph::Connection(sourcenodechannel1, destnodechannel1);
bool aud1confirm = graph.addConnection(aud1connection);
if (aud1confirm) {
      std::cout << "audInNode channel 0 connection successful." <<
      std::endl;
}
else {
      std::cout << "audInNode channel 0 connection successful." <<
      std::endl;
}
```

EK E (devam) PluginHostGraph.cpp içeriği

```
        sourcenodechannel2.nodeID = pluginnode->nodeID;
        sourcenodechannel2.channelIndex = 1;
        destnodechannel2.nodeID = AudioProcessorGraph::NodeID(2);
        destnodechannel2.channelIndex = 1;
        AudioProcessorGraph::Connection aud2connection =
        AudioProcessorGraph::Connection(sourcenodechannel2, destnodechannel2);
        bool aud2confirm = graph.addConnection(aud2connection);
        if (aud2confirm) {
              std::cout << "audInNode channel 1 connection successful." <<
              std::endl;
        }
        else {
              std::cout << "audInNode channel 1 connection successful." <<
              std::endl;
        }

}

void PluginHostGraph::addFxPlugin(const PluginDescription& desc,
                                  const AudioProcessorGraph::NodeID& target)
{
        String err = String();

        auto instance = formatManager.createPluginInstance(desc,
              graph.getSampleRate(),
              graph.getBlockSize(),
              err);

        instance->enableAllBuses();
        instance->prepareToPlay(graph.getSampleRate(), graph.getBlockSize());
        auto pluginnode = graph.addNode(std::move(instance),
        AudioProcessorGraph::NodeID((graph.getNumNodes() + 99)));
        std::cout << "\naddNode attempted. added plugin node: " <<
        pluginnode->getProcessor()->getName() << std::endl;
        std::cout << "target processor: " <<
        graph.getNodeForId(target)->getProcessor()->getName() << std::endl;
        std::cout << "plugin sample rate: " <<
        pluginnode->getProcessor()->getSampleRate() << std::endl;
        std::cout << "plugin block size: " <<
        pluginnode->getProcessor()->getBlockSize() << std::endl;
        std::cout << "number of nodes in the graph:" << graph.getNumNodes() <<
        "\n" << std::endl;

        for (int i = 0; i < graph.getNumNodes(); i++) {
              std::cout << "node" << i << " :" <<
              graph.getNode(i)->getProcessor()->getName() << " (nodeID: " <<
              graph.getNode(i)->nodeID.uid << ")" << std::endl;
        }

        if (pluginnode->isBypassed()) {
              std::cout << "\nplugin node is bypassed" << std::endl;
        }
        else {
              std::cout << "\nplugin node is not bypassed" << std::endl;
        }
```

EK E (devam) PluginHostGraph.cpp içeriği

```
std::cout << "plugin bus count:" <<
pluginnode->getProcessor()->getBusCount(true) << std::endl;
std::cout << "plugin 1st bus channel count:" <<
pluginnode->getProcessor()->getChannelCountOfBus(true, 0) << std::endl;
std::cout << "total num of input channels:" <<
pluginnode->getProcessor()->getTotalNumInputChannels() << std::endl;
std::cout << "total num of output channels:" <<
pluginnode->getProcessor()->getTotalNumOutputChannels() << std::endl;
auto paramlist = pluginnode->getProcessor()->getParameters();
for (int i = 0; i < paramlist.size(); i++) {
      std::cout << "param no " << (i + 1) << ": " <<
      paramlist[i]->getName(20) << std::endl;
}

auto connectionlist = graph.getConnections();
std::cout << "Existing connection count:" << connectionlist.size() <<
std::endl;
for (int i = 0; i < connectionlist.size(); i++) {
      if ((connectionlist[i].source.nodeID == target) &&
          (connectionlist[i].destination.nodeID ==
           AudioProcessorGraph::NodeID(2)))
      {
            graph.removeConnection(connectionlist[i]);
            std::cout << "Existing target connection removed" <<
            std::endl;
      }
}

AudioProcessorGraph::NodeAndChannel sourcenodechannel1;
AudioProcessorGraph::NodeAndChannel destnodechannel1;
AudioProcessorGraph::NodeAndChannel sourcenodechannel2;
AudioProcessorGraph::NodeAndChannel destnodechannel2;

sourcenodechannel1.nodeID = target;
sourcenodechannel1.channelIndex = 0;
destnodechannel1.nodeID = pluginnode->nodeID;
destnodechannel1.channelIndex = 0;

AudioProcessorGraph::Connection targ1connection =
AudioProcessorGraph::Connection(sourcenodechannel1, destnodechannel1);
bool targ1confirm = graph.addConnection(targ1connection);
if (targ1confirm) {
      std::cout << "TargetNode channel 0 connection successful." <<
      std::endl;
}
else {
      std::cout << "TargetNode channel 0 connection successful." <<
      std::endl;
}

sourcenodechannel2.nodeID = target;
sourcenodechannel2.channelIndex = 1;
destnodechannel2.nodeID = pluginnode->nodeID;
destnodechannel2.channelIndex = 1;
```

EK E (devam) PluginHostGraph.cpp içeriği

```
	AudioProcessorGraph::Connection targ2connection =
	AudioProcessorGraph::Connection(sourcenodechannel2, destnodechannel2);
	bool targ2confirm = graph.addConnection(targ2connection);
	if (targ2confirm) {
		std::cout << "TargetNode channel 1 connection successful." <<
		std::endl;
	}
	else {
		std::cout << "TargetNode channel 1 connection successful." <<
		std::endl;
	}


	sourcenodechannel1.nodeID = pluginnode->nodeID;
	sourcenodechannel1.channelIndex = 0;
	destnodechannel1.nodeID = AudioProcessorGraph::NodeID(2);
	destnodechannel1.channelIndex = 0;

	AudioProcessorGraph::Connection aud1connection =
	AudioProcessorGraph::Connection(sourcenodechannel1, destnodechannel1);
	bool aud1confirm = graph.addConnection(aud1connection);
	if (aud1confirm) {
		std::cout << "audInNode channel 0 connection successful." <<
		std::endl;
	}
	else {
		std::cout << "audInNode channel 0 connection successful." <<
		std::endl;
	}


	sourcenodechannel2.nodeID = pluginnode->nodeID;
	sourcenodechannel2.channelIndex = 1;
	destnodechannel2.nodeID = AudioProcessorGraph::NodeID(2);
	destnodechannel2.channelIndex = 1;
	AudioProcessorGraph::Connection aud2connection =
	AudioProcessorGraph::Connection(sourcenodechannel2, destnodechannel2);
	bool aud2confirm = graph.addConnection(aud2connection);
	if (aud2confirm) {
		std::cout << "audInNode channel 1 connection successful." <<
		std::endl;
	}
	else {
		std::cout << "audInNode channel 1 connection successful." <<
		std::endl;
	}

}
```

EK E (devam) PluginHostGraph.cpp içeriği

```
void PluginHostGraph::addFxPluginByName(String pluginmessage) {

	std::cout << "addFxPluginbyName invoked." << std::endl;
	auto firstindex = pluginmessage.indexOfChar('?');
	auto targetID = pluginmessage.substring(0, firstindex).getIntValue();
	auto pluginname = pluginmessage.substring((firstindex + 1),
                                                  pluginmessage.length());
	int selectedpluginindex = 22;
	for (int i = 0; i < kpList.getNumTypes(); i++) {
		if (pluginname.equalsIgnoreCase(kpList.getType(i)->name)) {
			selectedpluginindex = i;
		}
	}
	std::cout << "selectedplugin message is: " << pluginname << std::endl;
	std::cout << "Selected plugin index is: " << selectedpluginindex <<
	std::endl;
	auto descp = kpList.getType(selectedpluginindex);

	addFxPlugin(*descp, AudioProcessorGraph::NodeID(targetID));
}

void PluginHostGraph::addInstPluginByName(String pluginname) {

	std::cout << "addInstPluginbyName invoked." << std::endl;

	int selectedpluginindex = 22;
	for (int i = 0; i < kpList.getNumTypes(); i++) {
		if (pluginname.equalsIgnoreCase(kpList.getType(i)->name)) {
			selectedpluginindex = i;
		}
	}
	std::cout << "selectedplugin message is: " << pluginname << std::endl;
	std::cout << "Selected plugin index is: " << selectedpluginindex <<
	std::endl;

	auto descp = kpList.getType(selectedpluginindex);
	addInstPlugin(*descp);
}

void PluginHostGraph::changeParamByName(String paramchange) {
	auto firstindex = paramchange.indexOfChar('?');
	auto pluginID = paramchange.substring(0, firstindex).getIntValue();
	auto secondindex = paramchange.indexOfChar('%');
	auto paramname = paramchange.substring((firstindex + 1), secondindex);
	auto paramval = paramchange.substring((secondindex + 1),

                                      paramchange.length()).getFloatValue();
	std::cout << "Plugin Name: " << pluginID << std::endl;
	std::cout << "Param Name: " << paramname << std::endl;
	std::cout << "Param Val: " << paramval << std::endl;

	auto params = graph.getNodeForId
	(AudioProcessorGraph::NodeID(pluginID))->getProcessor()->getParameters();
	for (int i = 0; i < params.size(); i++) {
		if (params[i]->getName(20) == paramname) {
			params[i]->setValue(paramval);
		}
	}
}
```

EK F PluginManagementThread.h içeriği

```
#pragma once


#include "JuceHeader.h"
#include "UDPHandler.h"
#include "PluginHostGraph.h"
#pragma once



class PluginManagementThread : public Thread


{
public:
	PluginManagementThread(PluginHostGraph& hstgrphref);

	void run() override;

	UDPHandler pmtudp;
	PluginHostGraph& pmthstgrph;

};
```

EK G PluginManagementThread.cpp içeriği

```
#include "PluginManagementThread.h"
#include <string>



PluginManagementThread::PluginManagementThread(PluginHostGraph& hstgrphref)
	: Thread("PluginManagementThread"),
	  pmtudp(UDPHandler(true, 54020)),
	  pmthstgrph(hstgrphref)
{}

void PluginManagementThread::run()
{

	while (!threadShouldExit()) {
		String rubpluginmessage = pmtudp.UDPReceive();
		std::cout << "VmThread received: " << rubpluginmessage << std::endl;
		std::cout << rubpluginmessage.substring(0, 4) << std::endl;

		if (rubpluginmessage.substring(0, 1) == "i") {
			std::cout << "ins if passed" << std::endl;
			rubpluginmessage = rubpluginmessage.substring(1);

			if (rubpluginmessage.getLastCharacters(1) == "\n") {
				std::cout << "Second if passed" << std::endl;
				rubpluginmessage =
rubpluginmessage.dropLastCharacters(1);
				std::cout << "Message chomped." << std::endl;
			}

			MessageManager::callAsync([this, rubpluginmessage]()
			{pmthstgrph.addInstPluginByName(rubpluginmessage); });

		}

		if (rubpluginmessage.substring(0, 1) == "f") {
			std::cout << "fxp if passed" << std::endl;
			rubpluginmessage = rubpluginmessage.substring(1);

			if (rubpluginmessage.getLastCharacters(1) == "\n") {
				std::cout << "Second if passed" << std::endl;
				rubpluginmessage =
rubpluginmessage.dropLastCharacters(1);
				std::cout << "Message chomped." << std::endl;
			}

			MessageManager::callAsync([this, rubpluginmessage]()
			{pmthstgrph.addFxPluginByName(rubpluginmessage); });

		}
```

EK G (devam) PluginManagementThread.cpp içeriği

```
            if (rubpluginmessage.substring(0, 1) == "p") {

                  std::cout << "prm if passed" << std::endl;
                  auto rubparammessage = rubpluginmessage.substring(1);

                  if (rubparammessage.getLastCharacters(1) == "\n") {
                        std::cout << "Second if passed" << std::endl;
                        rubparammessage =
                        rubparammessage.dropLastCharacters(1);
                        std::cout << "Message chomped." << std::endl;
                  }

                  MessageManager::callAsync([this, rubparammessage]()
                  {pmthstgrph.changeParamByName(rubparammessage);});
            }
      }
}
```

EK H UDPHandler.h içeriği

```
#pragma once
#ifndef UDPHANDLER
#define UDPHANDLER

#include "JuceHeader.h"
#pragma comment (lib, "ws2_32.lib")
#include <WS2tcpip.h>
#include <string>


class UDPHandler
{
public:
	WORD wversionreq;
	WSADATA Data;
	int rStartUp;

	SOCKET sockin;
	SOCKET sockout;
	sockaddr_in in_serverHandler;
	sockaddr_in out_serverHandler;


	UDPHandler(bool isInbound, int port);
	~UDPHandler();
	void UDPSend(const String& str);
	String UDPReceive();
};
```

EK I UDPHandler.cpp içeriği

```
#include "UDPHandler.h"
#include <iostream>
#pragma once

UDPHandler::UDPHandler(bool isInbound, int port) {

	wversionreq = MAKEWORD(2, 2);
	rStartUp = WSAStartup(wversionreq, &Data);

	if (rStartUp != 0) {
		std::cout << "WSAStartup sikintili." << std::endl;
	}
	if (rStartUp == 0) {
		std::cout << "WSAStartup basarili." << std::endl;
	}
	if (isInbound == true) {

		sockin = socket(AF_INET, SOCK_DGRAM, 0);

		in_serverHandler;
		in_serverHandler.sin_addr.S_un.S_addr = ADDR_ANY;
		in_serverHandler.sin_family = AF_INET;
		in_serverHandler.sin_port = htons(port);

		if (bind(sockin, (sockaddr*)&in_serverHandler,
		sizeof(in_serverHandler)) == SOCKET_ERROR)
		{
			std::cout << "Binding isleminde sikinti cikti." <<
			WSAGetLastError() << std::endl;
		}
	}
	sockout = socket(AF_INET, SOCK_DGRAM, 0);

	out_serverHandler;
	out_serverHandler.sin_family = AF_INET;
	out_serverHandler.sin_port = htons(port);
	inet_pton(AF_INET, "127.0.0.1", &out_serverHandler.sin_addr);
};

UDPHandler::~UDPHandler() {
	WSACleanup();
}

void UDPHandler::UDPSend(const String& str)
{
	std::string s = str.toStdString();
	int sendOk = sendto(sockout,
			s.c_str(),
			s.size() + 1,
			0,
			(sockaddr*)&out_serverHandler,
			sizeof(out_serverHandler));

	if (sendOk == SOCKET_ERROR) {
		std::cout << "Gonderimde bi sikinti cikti" << WSAGetLastError() <<
		std::endl;
	}
};
```

EK I UDPHandler.cpp içeriği

```
String UDPHandler::UDPReceive()
{

	sockaddr_in clientHandler;
	int clientHandlerLength = sizeof(clientHandler);
	ZeroMemory(&clientHandler, clientHandlerLength);

	char buffer[1024];
	while (true) {

		ZeroMemory(buffer, 1024);

		int incomingBytes = recvfrom(sockin,
					buffer,
					1024,
					0,
					(sockaddr*)&clientHandler,
					&clientHandlerLength);

		if (incomingBytes == SOCKET_ERROR) {

			std::cout << "Client'tan alirken sikinti cikti." <<
			WSAGetLastError() << std::endl;
			continue;
		}

		char client_Ip[256];
		ZeroMemory(client_Ip, 256);

		inet_ntop(AF_INET,
				&clientHandler.sin_addr,
				 client_Ip, 256);

		std::cout << "Message recv from " << client_Ip << " : " <<
		buffer << std::endl;

		return String(buffer);
	}

}
```

EK J Sonic-Pi kaynak kodundaki sound.rb dosyasına yapılan ekleme

```
@@udphndlrthrd = UDPHandler.new(54020, 54080)
      def load_pluginst(plugin_name)
        plugin_name = plugin_name.to_s
        @@udphndlrthrd.UDPOut("i" + plugin_name)
      end

      def load_plugfx(target_plugin_id, plugin_name)
        target_plugin_id = target_plugin_id.to_s
        plugin_name = plugin_name.to_s
        @@udphndlrthrd.UDPOut("f" + target_plugin_id + '?' + plugin_name)
      end

      def param (plugin_id, param_name, param_val)
        @@udphndlrthrd.UDPOut("p" + plugin_id.to_s + '?' + param_name + '%' +
param_val.to_s)
      end
```

EK K Sonic-Pi kaynak kodundaki midi.rb dosyasına yapılan ekleme

```
      def plugin(*args)
        params, opts = split_params_and_merge_opts_array(args)
        opts         = current_midi_defaults.merge(opts)
        n, vel = *params

        if rest? n
          __midi_rest_message "midi :rest"
          return nil
        end

        n = normalise_transpose_and_tune_note_from_args(n, opts)

        on_val = opts.fetch(:on, 1)

        if truthy?(on_val)
          return midi_all_notes_off(opts) if n == :off

          channels = __resolve_midi_channels(opts)
          ports    = __resolve_midi_ports(opts)
          vel      = __resolve_midi_velocity(vel, opts)
          sus      = opts.fetch(:sustain, 1).to_f
          rel_vel  = opts.fetch(:release_velocity, 127)
          n        = n.round.min(0).max(127)
          chan     = pp_el_or_list(channels)
          port     = "PluginPort"

          ports.each do |p|
            channels.each do |c|
              __midi_send_timed_pc("/note_on", p, c, n, vel)
              time_warp sus - 0.01 do
                __midi_send_timed_pc("/note_off", p, c, n, rel_vel)
              end
            end
          end
          __midi_message "midi #{@@number_cache[n]}, #{@@number_cache[vel]},
sustain: #{sus}, port: #{port}, channel: #{chan}"
        else
          __midi_rest_message "midi #{@@number_cache[n]},
#{@@number_cache[vel]}, sustain: #{sus}, port: #{port}, channel: #{chan}, on:
0"
        end
        nil
      end
```

ÖZGEÇMİŞ

Adı Soyadı **: Doğuhan Eren KARACAN**

Doğum Tarihi ve Yeri **: 28.06.1995 / Ankara**

E-posta **: deren.karacan@mgu.edu.tr**

ÖĞRENİM DURUMU:

- **Lisans** **:** 2017, Ankara Üniversitesi, Hukuk Fakültesi, Hukuk Bölümü

MESLEKİ DENEYİM:

2019 –	Ankara Müzik ve Güzel Sanatlar Üniversitesi Müzik Bilimleri ve Teknolojileri Fakültesi Araştırma Görevlisi

YABANCI DİL:

İngilizce