Chunity: Integrated Audiovisual Programming in Unity

Jack Atherton
CCRMA, Stanford University
Stanford, CA, United States
lja@ccrma.stanford.edu

Ge Wang
CCRMA, Stanford University
Stanford, CA, United States
ge@ccrma.stanford.edu

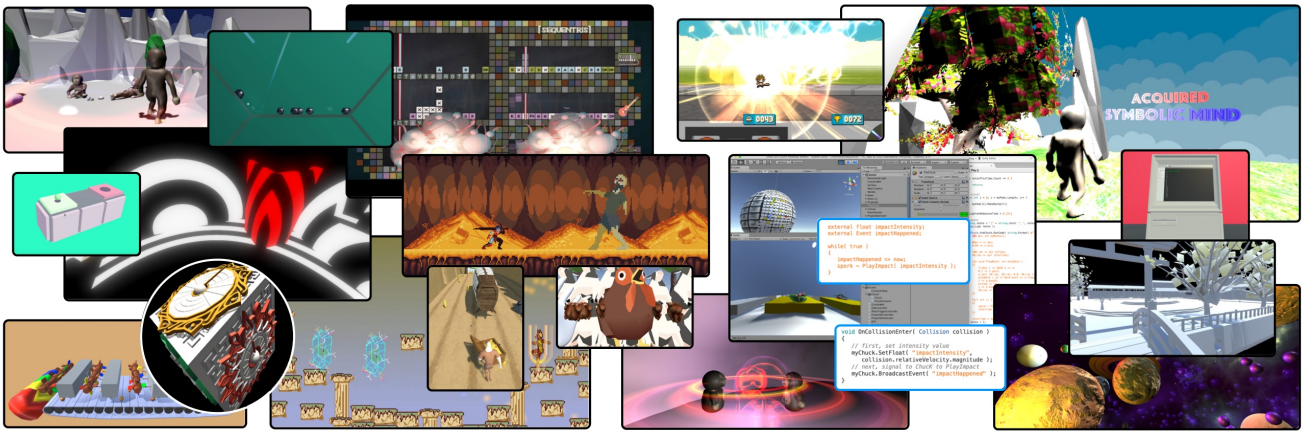

Figure 1: Chunity is a programming environment for the creation of interactive audiovisual software. It combines the strongly-timed audio synthesis of the ChucK language with the high-performance graphics of the Unity game engine.

ABSTRACT

Chunity is a programming environment for the design of interactive audiovisual games, instruments, and experiences. It embodies an audio-driven, sound-first approach that integrates audio programming and graphics programming in the same workflow, taking advantage of strongly-timed audio programming features of the ChucK programming language and the state-of-the-art real-time graphics engine found in Unity. We describe both the system and its intended workflow for the creation of expressive audiovisual works. Chunity was evaluated as the primary software platform in a computer music and design course, where students created a diverse assortment of interactive audiovisual software. We present results from the evaluation and discuss Chunity's usability, utility, and aesthetics as a way of working. Through these, we argue for Chunity as a unique and useful way to program sound, graphics, and interaction in tandem, giving users the flexibility to use a game engine to do much more than "just" make games.

Author Keywords

audiovisual interaction, ChucK, Unity, programming

CCS Concepts

•**Applied computing** → **Sound and music computing;**
•**Human-centered computing** → **Interaction design;**
•**Software engineering** → Domain specific languages;

NIME'18, June 3-6, 2018, Blacksburg, Virginia, USA.

1. INTRODUCTION

This paper describes the Chunity project, which combines the audio programming language ChucK with the game engine Unity for the creation of artful and interactive audiovisual applications. Chunity is both a tool and a workflow for creating tools, games, toys, instruments, and experiences.

As music researchers who work with the interplay between sound and graphics, we seek to create tools that prioritize audio and allow systems to be *audio-driven* when helpful (e.g. for precise, musically-timed graphics).

We chose to work with ChucK and Unity to combine the respective affordances of each. For example, ChucK is designed to enable a temporally deterministic way to control sound synthesis, whereas Unity is designed for high-level control over real-time graphics and physics simulations. Chunity creates a single environment combining the capabilities of both tools.

Tools, and new tools especially, always suggest particular ways of working. While we find it important to design tools with usability in mind, we believe it is equally important to examine the *aesthetics* of using the tool: what ways of working does it encourage? What are the specific ways in which it allows you to accomplish certain tasks? How does using it make you feel? In this context, Chunity is both a tool as well as the paradigmatic ways of working with that tool; the sum of these parts implicitly shapes what one can create with it. As such, we have evaluated Chunity by examining how students use it to create interactive audiovisual applications of their own design.

In the rest of this paper, we outline various related approaches to audiovisual programming. We articulate a design ethos in creating Chunity and discuss its workflow, highlighting a few concrete examples, as well as providing notes on Chunity's implementation. Finally, we present our qualitative evaluation and discuss its implications.

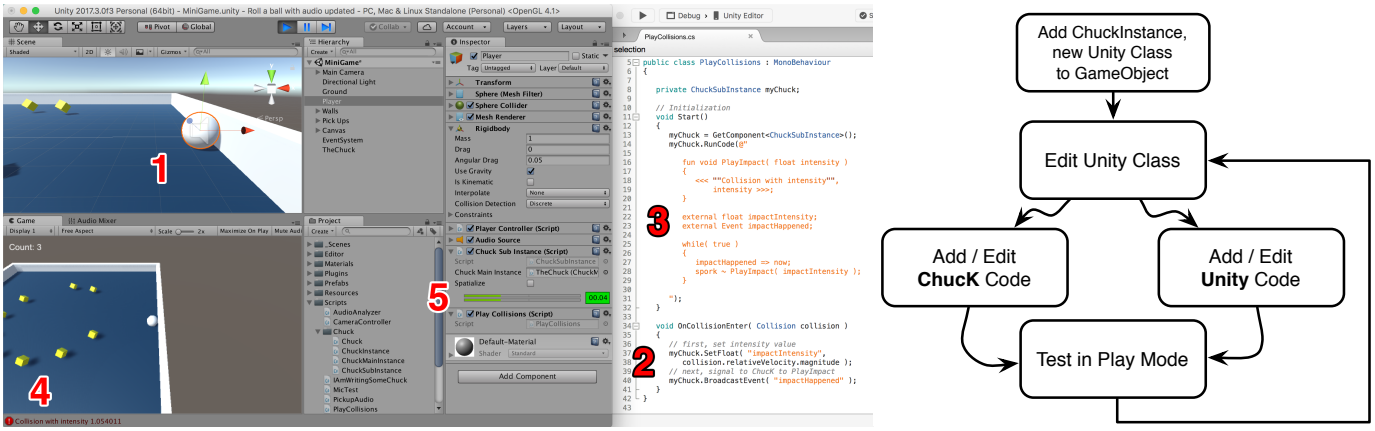

Figure 2: Workflow: (1) Users make changes in the Unity Scene, (2) Unity C# code, and (3) ChucK code, then test their game to see how it currently looks and sounds (4, 5). **§4.1 Runtime:** (1) The Player object collides into another game object. (2) The `OnCollisionEnter` Unity callback is called. The ChucK float `impactIntensity` is set, then the ChucK Event `impactHappened` is broadcast. (3) The ChucK code listening for `impactHappened` sporks the ChucK function `PlayImpact`. (4) This function prints to the Unity console, and (5) plays an impact sound according to the intensity.

2. RELATED WORKS

In contextualizing this work, we have identified three main approaches for creating interactive audiovisual applications. The first approach involves programming audio and graphics in a low-level language like C++. This approach uses tools with basic affordances, such as callback functions that directly compute audio samples [12] and low-level graphics primitives like the OpenGL API [10]. Audiovisual applications created with this approach can be expressive, but often require a lot of work or the use of external libraries to assert high-level control over audio and graphics. Examples of this approach also include works using the Synthesis ToolKit, OpenFrameworks, and Cinder [4, 6, 9, 2, 3].

The second approach involves working in a game engine such as Unity or Unreal Engine. Game engines have powerful tools for interactive graphics such as physics engines, but usually limit audio programming to the playback of audio files through a few simple filters [14, 7]. This approach is used by independent ("indie") video games with musical aspects, such as Night in the Woods and Sentris [15, 1, 16].

The third approach involves linking an audio engine to a graphics engine via a network communication protocol such as Open Sound Control [18]. This approach enables the integration of audio programming languages like ChucK, SuperCollider, and Pure Data with game engines, as in UDK-OSC [5]. Using the network is flexible, but can introduce new complexities (e.g. scheduling granularity, distributed mindset) that make tight integration of audio and graphics difficult. This approach is used in works by the Virtual Human Interaction Lab, the Stanford Laptop Orchestra, and many works in the NIME community [11, 17, 4, 6].

There are existing environments that combine elements of these approaches. For example, Max/MSP/Jitter couples high-level control of audio with graphics in a tight integration that does not rely on network communication [8]. While Max/MSP lends itself to certain ways of working, its graphical patching paradigm does not inherently support clear reasoning about time and sequential operations.

3. APPROACH AND DESIGN ETHOS

In creating Chunity, we were guided by two main principles. First, that tools should be *audio-driven*. Audio should be prioritized as a first-class component, enabling implementation of complex synthesis techniques and other high-level controls. In such a regime, audio can drive graphics events as needed to achieve robust, precise control over time.

Second, that audio and graphics should be as tightly integrated as possible. The two should be programmed together in the same context in the programmer's workflow; communication between them should be reliable and fast.

4. WORKFLOW

Since Chunity is used to design graphics and audio in tandem, a typical workflow involves iteration and testing on graphics and audio together. Figure 2 shows how a user would program and test the code example of Section 4.1.

4.1 Physics-Driven Procedural Audio

This code plays a ChucK function to generate the sound for a collision, informed by the speed of that collision.

```
1  public class PlayCollisions : MonoBehaviour {
2    private ChuckSubInstance myChuck;
3
4    // Initialization
5    void Start() {
6      myChuck = GetComponent<ChuckSubInstance>();
7      myChuck.RunCode(@"
8        fun void PlayImpact( float intensity ) {
9          // play a collision sound...
10       }
11
12       external float impactIntensity;
13       external Event impactHappened;
14
15       while( true ) {
16         impactHappened => now;
17         spork ~ PlayImpact( impactIntensity );
18       }
19     ");
20   }
21
22   // Run on every Unity physics collision
23   void OnCollisionEnter( Collision collision ) {
24     // first, set ChucK intensity value
25     myChuck.SetFloat( "impactIntensity",
26       collision.relativeVelocity.magnitude );
27
28     // next, signal to ChucK to PlayImpact
29     myChuck.BroadcastEvent( "impactHappened" );
30   }
31 }
```

Every time a Unity physics collision occurs, this script sets the value of the float `impactIntensity`, then broadcasts the event `impactHappened` (lines 25-29), which signals to ChucK to spork (run concurrently) a function that plays a sound using the value of `impactIntensity` (line 17).

4.2 ChucK as Strongly-Timed Clock

This code rotates an object every 250 ms, with the timing being generated exactly via ChucK.

```
1  public class EventResponder : MonoBehaviour {
2    private ChuckSubInstance myChuck;
3
4    void Start() {
5      myChuck = GetComponent<ChuckSubInstance>();
6
7      // broadcast "notifier" every 250 ms
8      myChuck.RunCode( @"
9        external Event notifier;
10       while( true ) {
11         notifier.broadcast();
12         250::ms => now;
13       }
14     " );
15
16     // create a ChuckEventListener
17     ChuckEventListener listener = gameObject
18       .AddComponent<ChuckEventListener>();
19
20     // call MyCallback() during Update()
21     //   after every broadcast from "notifier"
22     listener.ListenForEvent( myChuck, "notifier",
23       MyCallback );
24   }
25
26   void MyCallback() {
27     // react to event (rotate my object)
28     transform.Rotate( new Vector3( 5, 10, 15 ) );
29   }
30 }
```

Every time the `notifier` Event is broadcast (line 11), the `ChuckEventListener` (lines 17-23) stores a message on the audio thread that the broadcast happened. Then, the user's callback `MyCallback` (line 26) is called on the next visual frame. `ChuckEventListener` is part of a growing body of helper components that encapsulate basic patterns using external variables. Note also that this architecture works for Events that fire on any schedule, not just a simple regular schedule as defined in the above ChucK code.

5. IMPLEMENTATION

Chunity is a C++ Unity Native Audio Plugin that is accessed via C# scripts. Figure 3 shows how user-written classes and the Unity audio engine interact with Chunity.

5.1 External Variables

We have added the new `external` keyword to enable integrated communication between ChucK code and the outside environment that ChucK is embedded in (the *embedding host*). The external keyword is used when declaring the type of a variable, such as in Section 4.2 (line 9). The main guiding principle in the design of this keyword is that it is not necessary for ChucK to know anything about the embedding host, or whether it is embedded at all. Instead, external variables appear like normal variables within their own ChucK script, but can be inspected, edited, or listened to by other ChucK scripts or by the embedding host.

So far, the `external` keyword is enabled for three types of variables. The first type of external variable is primitives: `ints`, `floats`, and `strings`. The embedding host can get and set their values. The get operation requires the use of a callback because the embedding host often runs on a different thread than the audio thread.

The second type of external variable is `Events`. ChucK Events are used to pause execution in a ChucK script until the Event signals that it has occurred. The embedding host can *signal* or *broadcast* an external Event (i.e. trigger *one* or *all* ChucK scripts waiting on the event). The embedding host can also register a C# callback to be called every time an external Event is broadcast, as in Section 4.2 (line 22). This callback to user code occurs on the audio thread and thus is timed with sample-level accuracy; a tighter integration of timing between audio and visuals is not achievable.

The third type of external variable is `UGens` (unit generators). ChucK UGens are signal processing elements that generate streams of audio. The embedding host can fetch an external UGen's most recent samples.

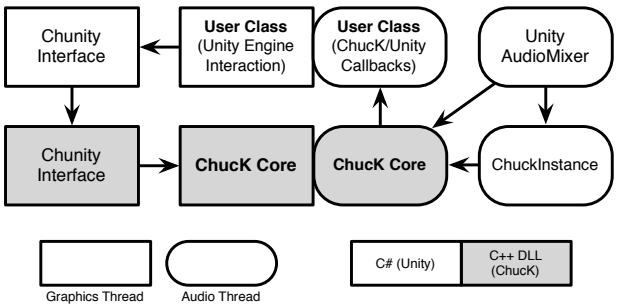

Figure 3: The architecture of Chunity. Users write classes in C# that send code and external variable requests to the Chunity interface, which passes them on to ChucK. When necessary, ChucK calls callbacks in the user class from the audio thread. The Unity AudioMixer and ChuckInstance classes call ChucK's audio callback, causing sound to be computed and time to be advanced.

5.2 Internal Rearchitecture

The desire to embed ChucK in Unity motivated the wider libChucK rearchitecture project, which enables ChucK to act as an embeddable component in any C++ project.

The ChucK source was separated into *core* and *host* codebases. The core comprises the language parser, which compiles code, and virtual machine (VM), which translates audio inputs to outputs. One embeds ChucK in a new project by simply writing a new host that calls these functions.

The rearchitecture allowed multiple VMs to exist in the same address space (useful for contexts where the number of channels is limited and multiple outputs are desired, such as in a digital audio plugin or Unity's spatial audio system). It also enabled the redirection of all ChucK error messages to an optional callback (e.g. the Unity debug console).

5.3 Interface with Unity

Chunity can be added to a Unity project in two ways: as a channel strip plugin, or placed directly on a game object.

As a plugin, a ChucK VM acts as a digital effect. This method is efficient, implemented entirely in C++, but each plugin must be added manually, and plugins cannot receive both microphone input and data for sound spatialization.

Through a `ChuckInstance` C# component on a game object, a ChucK VM acts as a virtual sound source that can be spatialized within the game world. This method also enables new ChucK VMs to be constructed programmatically with the use of Unity *prefabs* (archetypes of objects).

To address the inefficiency of including multiple ChucK VMs just to spatialize audio from multiple locations, we introduced `ChuckMainInstance` and `ChuckSubInstance`. A ChuckMainInstance fetches microphone input from Unity and advances time in its VM. Each ChuckSubInstance has a reference to a shared ChuckMainInstance and fetches its output samples from an external UGen in that VM, perhaps spatializing the result along the way. This way, many spatialized ChucK scripts can all rely on the same VM and microphone, saving some computational overhead.

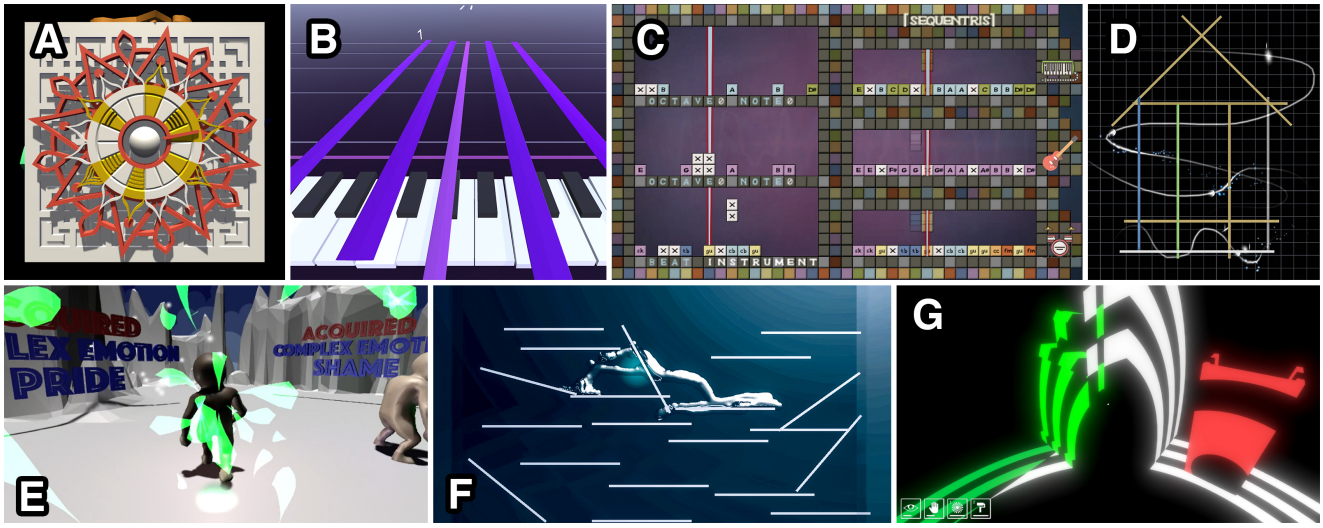

Figure 4: Student work. A: MandalaBox. **B:** Keyboreal. **C:** Sequentris. **D:** Stroquencer. **E:** Music and Evolution: From Grunts to Songs. **F:** Vessel: Liquid Choreography. **G:** Unblind. **(See video at `https://vimeo.com/263613454`.)**

6. EVALUATION

Chunity is both a tool and a way of working. The success of such a tool lies in what people can create with it. Therefore, we believe that the best evaluation of this project is a qualitative one wherein users engage with the tool and its workflow to realize projects they have designed themselves.

A class of 24 students used Chunity to design projects throughout a ten-week course at Stanford University, including a step sequencer assignment and a final project, for which they created any interactive audiovisual software of their own design. Below are some examples of the students' work; see also Figure 4 for screenshots and video.

6.1 Student Work

MandalaBox (Figure 4A). Users manipulate an ornate artifact covered in mandalas to sequence the intricate patterns of a Balinese gamelan. Different mandalas control the base melody, the percussive rhythm, and ornamentations on top of the melody. The MasterMandala acts as a meta-sequencer, allowing the user to program switches between different patterns they have saved.

Keyboreal (Figure 4B). A tool for keyboard recording and playback. Users play a 3D keyboard in real time, then edit individual notes, scroll through the recording at their own speed, set loop boundaries, and quantize. Here, ChucK affords flexible timing, as the recording can be scrubbed through in real time and played back at any rate.

Sequentris (Figure 4C). A sequencer game where melody, bassline, and percussion are set by clearing a row from a game of Tetris. Users select the pitch or timbre of each block before placing it in their game. The game also features alternate timing modes like "Jazz Mode" (swung timing).

Stroquencer (Figure 4D). Users arrange lines on a grid to represent different sounds, then draw paths across the lines. Small sprites travel along the paths at the same speed they were drawn. The sprites stroke each line they cross to play its sound. The position of the line crossing is mapped to pitch, and the color of the line is mapped to a variety of timbres in ChucK or to sounds recorded by the user.

Music and Evolution: From Grunts to Songs (Figure 4E). A game and interactive "essay" exploring how music might have driven pre-humans to evolve their minds. Players interact with other apes to compete in music contests (and acquire "complex emotion: shame"), communicate ("musilanguage"), and make music together ("pattern sense"). Unity and ChucK are used in tandem to create fluid animations tightly coupled to generative soundtracks. For example, once the player has acquired "rhythm sense" and "pitch sense", each step their ape avatar takes is accompanied by a note and percussive sound from a melody.

Vessel: Liquid Choreography (Figure 4F). An artifact where the user guides a sentient liquid through a series of obstacles. This exploration of the aesthetics of fluid modeling links complex Unity fluid simulations to a granular synthesis algorithm in ChucK, allowing virtual space to "choreograph" the simulated liquid. If the user is lucky, the liquid may tell them "Good night!" during the experience.

Unblind (Figure 4G). A game in which the protagonist sees through sending out integrated audiovisual sound waves to interact with the world. The narrative follows the protagonist's journey through five levels to reintegrate with their community following the loss of their vision. Abilities in addition to seeing through sound waves include "Resonance" (only see similar objects), "Beacon" (several objects remain lit) and "Concentration" (slow time).

6.2 Reported Workflow

The students volunteered their thoughts on using Chunity in an extended, qualitative, open-ended questionnaire.

Most students preferred to work in an integrated way, as described in Figure 2.

- "Usually I want to wrap all the ChucK code in C# functions as quickly as possible so I can abstract away all the nitty-gritty audio details."
- "I normally start with a big idea, and start building the gameflow chronologically. Then I hit walls or discover cool tools or functions within Chunity. Then the small parts within the big picture get changed. There are a lot of improvisations on the way to a finished design."

	Mean ± S.D.	[Min,Max]
Years Music Training	10.02 ± 6.30	$[0, 23]$
Years Programming	5.30 ± 2.96	$[2, 14]$
Years ChucK	0.34 ± 0.52	$[0, 2]$
Years Unity	0.28 ± 0.51	$[0, 2]$

Table 1: Student Demographics. Students had considerable training in music and programming, but most were new to ChucK and Unity.

A number of students preferred to prototype the initial version of their interactive audio in miniAudicle, the integrated development environment (IDE) for ChucK [13]. Then, they would move this first version into Chunity and work in an integrated way from there.

- "I tinker and make desired sounds and code logic in miniAudicle, write it in Chunity, then test, iterate, and refine within Chunity."

A couple students preferred to prototype their visuals first.

- "I build my environment first, and then create my sound objects with a hierarchy designed to streamline ChucK as much as possible. However, the sound (ChucK) is usually secondary to visual / mechanical concerns."

6.3 Reported Experience

The students found it satisfying that Chunity enabled one to start working quickly,

- "It just 'works' – like sound spatialization comes with it, it's not too hard to set up, and it's fun."

that it was straightforward to connect Unity and ChucK,

- "The ability to control the audio content in exact relation to the visuals, programmatically, is great."
- "I liked the overall simplicity of mapping interaction / behavior of game elements to sound."
- "Setting ChucK values from Unity was straightforward. Getting ChucK values was usually satisfying."

that Chunity could be used for timing graphical events,

- "It's nice to have a separate, strongly-timed assistant. I don't like relying on frame rate."
- "As an audio mechanism, it was amazing for getting precise timing."
- "Made it easy to trigger events and time Unity movements."

that Chunity enabled access to procedural audio,

- "It is very useful if you want to create some arbitrary sounds or dynamic melodies because you don't need to pre-record them."
- "I liked creating a class and being able to spawn many versions of the class and get cool sounds!"

that Chunity enabled on-the-fly addition of new audio code,

- "I liked the ability to use a RunCode to insert a line in a ChucK instance at any time."

and that Chunity fostered a well-integrated workflow between ChucK and Unity.

- "I once connected Supercollider and Unity using OSC messages to create a simple audio puzzle game, and Chunity was much easier to use. Using OSC made me go back and forth between Unity and Supercollider, but with Chunity, I only had to worry about Unity."

Students had a number of requested features for the future of the tool, including improved error messages,

- "Chunity's ChucK error messages were fairly vague, making debugging difficult."
- "I debugged ChucK code separately in miniAudicle since it's easier there."

external array variables,

- "Getting ChucK values was a bit cluttered when many values were being polled."
- "Doesn't support external array"
- "Want arrays!"

improved performance,

- "Instantiating multiple VMs quickly chewed up CPU resources, although ChuckMainInstance and ChuckSubInstance helped."

and better ways to code ChucK in-line in the Unity editor.

- "Writing ChucK code inline was sometimes painful."
- "Code editor in Unity doesn't highlight errors or useful things, and errors are a little ambiguous to know what line they refer to."

Overall, the students generally appreciated Chunity as a tool, even despite its current limitations.

- "It is a great tool that enables you to break down audio and make it your own."
- "I feel like I'm starting to get good at it! And I feel more *powerful*."
- "I was ok with some of the bugs / lacks of functionality (i.e. no external arrays) because it forced me to think in different ways / learn deeply. :)"
- "It's amazing. Even though it does sometimes crash, I would be much worse off without it."
- "ChucK → Chunity
  Batman → Batmobile."
- "Don't really know if I like what I made, but I made it."
- "There is definitely a learning curve, since you need to know ChucK. But if I had to write the audio / timing code from scratch, it would be a lot worse."

Meanwhile, other students noted that Chunity did not fully support their own preferred ways of working; this may be attributed to the idiosyncrasies of both Unity and ChucK.

- "It mostly meshes well with Unity's aesthetics, but I also don't really care for Unity's aesthetics."
- "If the aesthetic of your product works well with ChucK-generated sound, it's excellent. If the aesthetic is different, it works, but can be challenging."

The questionnaire also contained a series of statements where the students marked "Strongly Disagree - Disagree - Neutral - Agree - Strongly Agree". We codified these responses to represent numbers 1 through 5. This was not intended as a quantitative assessment, but rather as another useful qualitative way to gauge how people felt in using the tool.

- 4.59/5: I felt empowered to make things I wouldn't have otherwise
- 4.54/5: I had new opportunities to express myself
- 4.50/5: I was proud of what I made
- 4.50/5: I improved my ability to be creative
- 4.09/5: UGens were satisfying to use
- 4.05/5: Controlling **audio** timing was satisfying
- 3.68/5: Chunity allowed me to prototype quickly
- 3.09/5: Controlling **graphical** timing was satisfying

Ultimately, our students seemed empowered by this tool. At the same time, it is clear that much can be improved both in terms of existing features and in terms of making Chunity more satisfying to use. We will consider both of these takeaways as we continue to evolve Chunity.

7. DISCUSSION

So far, we have presented Chunity's approach, workflow, implementation, and evaluation as an environment for creating interactive audiovisual artifacts. It embodies an audio-first, strongly-timed, and high-level ethos integrated into the workflow of a high-performance game development tool.

We have seen people make diverse use of Chunity to create sophisticated and artful systems. In this section, we discuss some aesthetic considerations of designing an integrated tool like Chunity.

Through this process, we sought both to create a new tool and also to understand its implications. Since all tools encourage particular ways of working (often resulting from values embedded in the design of the tool), our evaluation attempted to better understand the ways of working that an unconventional hybrid such as Chunity suggests.

Understanding the ways of working encouraged by such a large system is not straightforward, for it is not always readily reducible to (or susceptible to study from) its constituent parts. Such understanding involves the overall aesthetics of using the tool, what it allows one to do, and the manners in which it suggests itself, as well as the domain(s) it engages with. Interactive audiovisual design is an inherently complex and messy domain. It entails working simultaneously with interaction, sound, and graphics to create a single coherent experience. It asks the designer to reconcile their conceptions and intentions with the idiosyncrasies of the underlying tools, while working with two different programming paradigms.

As a programming paradigm, Unity encourages ways of working that mesh well with its conception as a state-of-the-art graphics engine and game development platform. Unity's workflow, while complex, has become something of an industry standard that is widely used and understood. Meanwhile, ChucK provides a specific way of thinking about time and concurrency as musical constructs. In our design of Chunity, we wanted to find an amalgam that takes advantage of Unity and ChucK's respective ways of working instead of creating something entirely new.

In thinking critically about Chunity as such a hybrid tool, we have observed both limitations and useful and expressive affordances not found elsewhere. The inherent tension and sense of complexity in mixing two disparate paradigms (e.g., graphics/game vs. audio; GUI+C# vs. text-based/ChucK) is evident in the students' feedback. In spite of this tension, the integration of ChucK into Unity allowed people to craft audio synthesis code from the ground up, and to programmatically use it to drive graphics and interaction.

More importantly, Chunity's affordances empowered developers to create artful systems that interoperated tightly, and to reason about such systems as one cohesive entity. The "inline" integration of ChucK and Unity was valuable in this regard because it allowed users to work in a way previously not possible — this is a clear break from the distributed computing model used by solutions that link two engines together with network communication. In particular, users of Chunity were able to adopt an audio-first, strongly-timed workflow in places where it served their need (e.g., "I want my ape's animations to be tightly coupled to the generated music!"), while continuing to take advantage of more "traditional" Unity workflows. These affordances were not originally present in Unity alone, which has no means to synthesize audio on-the-fly and relies on the graphics frame rate and system timers as timing mechanisms.

Presently, we have begun to address known limitations of Chunity by adding features to improve efficiency (e.g., ChuckMain / SubInstance for better spatial audio performance) and ease of use (e.g., the helper component ChuckEventListener of Section 4.2 for abstracting away communication complexities between audio and graphics threads). Moving forward, we hope to better visualize internal Chunity state in real-time (such as the values of external variables); we also hope to further improve the quality of life of writing ChucK code embedded in another context.

Ultimately, as we continue to explore its design and implications, we see Chunity as two things: a unique *tool* for integrating sound, graphics, and interaction — and a new *way of working* that gives users the flexibility to use a game engine to do much more than "just" make games.

Download Chunity and view further documentation at `http://chuck.stanford.edu/chunity/`.

8. ACKNOWLEDGEMENTS

Thanks to all the students of Music 256A / CS 476A at Stanford University. We would also like to thank Spencer Salazar and Romain Michon for their support. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1656518.

9. REFERENCES

[1] L. Alexander. Art and tech come full-circle in Sentris. In *Gamasutra*, 2014.

[2] Cinder. `https://libcinder.org/`. Accessed: 2018-01-11.

[3] P. R. Cook and G. P. Scavone. The Synthesis ToolKit (STK). In *ICMC*, 1999.

[4] N. Correia and A. Tanaka. Prototyping audiovisual performance tools: A hackathon approach. In *NIME*, June 2015.

[5] R. Hamilton. UDKOSC: An immersive musical environment. In *International Computer Music Conference*, Aug. 2011.

[6] J. Hochenbaum, O. Vallis, D. Diakopoulos, J. W. Murphy, and A. Kapur. Designing expressive musical interfaces for tabletop surfaces. In *NIME*, 2010.

[7] M. Lanham. *Game Audio Development with Unity 5.X*. Packt Publishing Ltd, June 2017.

[8] Max Software Tools for Media | Cycling '74. `https://cycling74.com/products/max/`. Accessed: 2018-01-11.

[9] openFrameworks. `http://openframeworks.cc/`. Accessed: 2018-01-11.

[10] OpenGL - The Industry Standard for High Performance Graphics. `https://www.opengl.org/`. Accessed: 2018-01-11.

[11] R. S. Rosenberg, S. L. Baughman, and J. N. Bailenson. Virtual Superheroes: Using Superpowers in Virtual Reality to Encourage Prosocial Behavior. *PLOS ONE*, 8(1):e55003, Jan. 2013.

[12] The RtAudio Home Page. `https://www.music.mcgill.ca/~gary/rtaudio/`. Accessed: 2018-01-11.

[13] S. Salazar, G. Wang, and P. R. Cook. miniAudicle and ChucK Shell: New Interfaces for ChucK Development and Performance. In *ICMC*, 2006.

[14] Unity - Audio. `https://unity3d.com/learn/tutorials/s/audio`. Accessed: 2018-01-11.

[15] Unity awards 2017. `https://awards.unity.com/`. Accessed: 2018-01-11.

[16] Unreal Engine. `https://www.unrealengine.com`. Accessed: 2018-01-17.

[17] G. Wang, N. Bryan, J. Oh, and R. Hamilton. Stanford Laptop Orchestra (SLOrk). In *International Computer Music Conference*, Jan. 2009.

[18] M. Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.