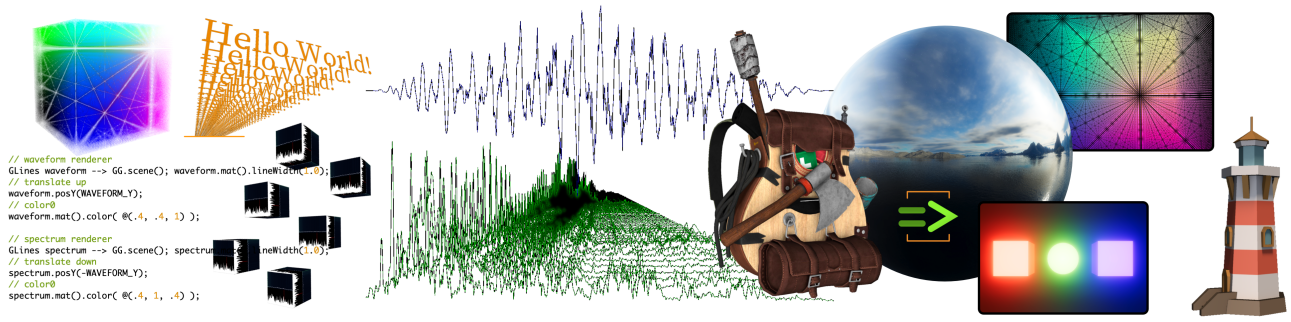
ChuGL: Unified Audiovisual Programming in ChucK

Andrew Zhu Aday
CCRMA, Stanford University
Stanford, California, United States
azaday@ccrma.stanford.edu

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



ABSTRACT

ChuGL (sounds like "chuckle"; rhymes with "juggle") is a unified audiovisual programming framework built into the ChucK language. It extends ChucK's strongly-timed, concurrent programming model with a 3D rendering engine and a new paradigm for coding real-time graphics and audio. ChuGL introduces the notion of a Graphics Generator (GGen) that can be manipulated sample-synchronously alongside audio unit generators (UGens) to unify graphics and audio within a single strongly-timed language. Under the hood, this is made possible by a multithreaded scenegraph architecture that provides low-latency, high performance audiovisual synchronization. In this paper we present the design ethos of ChuGL, describe its integrated graphics-and-audio workflow, highlight architectural decisions, and present an evaluation of ChuGL as a tool for expressive audiovisual design, used in a computer music programming course at Stanford University. ChuGL transforms ChucK into a standalone audiovisual programming language, and argues for a way of thinking and doing in which audio and graphics are given equal importance.

Author Keywords

Audiovisual interaction, 3D/2D graphics, ChucK, strongly-timed, programming

CCS Concepts

•**Applied computing** → *Sound and music computing;*
•**Software and its engineering** → *Domain specific languages;*
•**Human-centered computing** → *Systems and tools for interaction design;*

NIME'24, 4–6 September, Utrecht, The Netherlands.

1. INTRODUCTION

ChuGL is a high-performance 3D graphics tool and workflow built into the ChucK programming language [26]. It is designed around ChucK's existing strongly-timed, concurrent programming model such that users can create interactive audiovisual applications with sample-synchronous timing of both graphics and audio.

As computer musicians who also care about graphics and gaming, we designed ChuGL holistically as an *audio-driven* graphics API. Being *audio-driven* [10] applies to ChuGL both as a tool and a workflow. As a tool, ChuGL is implemented within and driven by the ChucK virtual machine; this placement of a graphics engine within the context of an audio engine is the inverse of typical game engine architecture, and results in a programmer experience uniquely different from other methods of audiovisual programming. As a workflow, ChuGL seamlessly integrates into a ChucK-ian style of using musical events and precise sample-synchronous timing to drive audio, graphics, and the structure of the audiovisual experience as a whole; graphics in ChuGL are "strongly-timed" in the same way audio is strongly-timed in ChucK. This workflow—the ability to think about and program graphics parallel to how one already uses ChucK to program sound—leads us to propose that ChucK+ChuGL is (one of) the first programming languages with native support for unified audiovisual programming.

Put another way, ChuGL is designed for people who care deeply about interactive sound programming and are familiar with traditional audio workflows. This paper presents a unified workflow where Graphics Generators (GGens) parallel Unit Generators (UGens), scenegraphs parallel signal networks, graphical events are treated like any other timing event, and wherever else possible the graphics API naturally extends the pre-existing ways of using ChucK for sound-synthesis. ChuGL is for audiovisual designers, computer musicians, game programmers interested in audio, and students of all the above.

As hands-on designers ourselves, we believe the merit of a tool comes not from its complexity or feature set, but from what one can actually make with it, and the quality of one's moment-to-moment experience during that creative process. Therefore, we have chosen to evaluate ChuGL as the primary programming environment used in Stanford

University's Music, Computing, Design course of Fall 2023.

The remainder of this paper is organized as follows: we compare related tools for audiovisual programming, explain the motivation behind our design decisions, demonstrate a workflow for strongly-timed graphics, highlight architectural decisions, and evaluate ChuGL's use in a classroom setting. Lastly, we describe ongoing work and reflect on our journey thus far.

2. RELATED WORK

There are many methods for programming interactive audiovisual applications. Some involve working directly with graphics and audio APIs in a systems language such as Rust or C++. This means building tools and abstractions from first principles. While flexible and performant, this also entails significant programming overhead, and requires the programmer to be knowledgeable about low-level implementation details. Relevant libraries include OpenGL, Vulkan, RtAudio, SynthesisToolKit, and OpenFrameworks [4, 13, 7, 23].

Another method is to use two separate applications—one for audio, another for graphics—and synchronize data between them over a network protocol such as Open Sound Control [28]. Using this method, one can coordinate audio synthesis from tools such as ChucK or Pure Data [20] with rendering abilities of applications such as Processing [5]. This is a popular approach used by many, including the Stanford Laptop Orchestra [25]. However, synchronizing data between separate processes introduces both a technical limitation in terms of the latency and bandwidth of data transfer, as well as programming overhead in terms of having to structure code around these data-sharing requirements.

A third method is to use a game engine equipped with a sound effects (SFX) engine. Options include Unity, Unreal Engine, and Godot, which have plugins or builtin integrations with SFX engines such as FMOD [21] and Wwise [8]. These game engines have renderers optimized for complex 3D graphics and audio APIs which are designed around the requirements typical of 3D games: optimized sound file playback and spatialization. SFX engines offer additional tools for offline sound authoring but are not meant for real-time sound synthesis; they are generally incapable of precise musical timing, generative composition, or other requirements typical of computer music.

Yet another approach is to use a graphical patching tool such as Max/MSP/Jitter [3] or TouchDesigner [6]. These visual interfaces are great for non-programmers and enable rapid experimentation of the mapping schemes between both graphics and audio generators. However, they also share the typical shortcomings of graphical patching languages, which include the dependence on existing high-level objects (or externals programmed in low-level implementation languages) and the inability to programmatically manipulate patch structure. As a result, these tools typically are not used to develop general audiovisual systems such as video games, which is a design goal of ChuGL.

Live-coding tools offer their own style of audiovisual programming. There are separate tools for graphics [2, 1] and music [16, 17], which can communicate over network or virtual audio channels. These systems are built for live performance, prioritizing terseness and expressivity along very specific directions. Consequently, they are designed for use only by the performer, and are not intended for general audiovisual software development.

The final method we observe is embedding a programmable audio engine inside a game engine. This is the architecture of Chunity [9] and Chunreal[1], which embed the ChucK VM inside Unity and Unreal, respectively. This approach offers better performance and tighter program integration than using separate applications connected over network (here the network layer is removed), but shares the drawback of needing to code in multiple languages and coordinate data and timing across two separate processes. Significant programmer effort is often required to properly synchronize data across audio and graphics code, which becomes a design and performance bottleneck for large-scale systems.

3. DESIGN ETHOS

Audiovisual programming is inherently complex. Audio and graphics have traditionally been separate subfields of computing, each with their own established tools, techniques, concepts, workflows, etc. Historically, the audiovisual designer who aspires to integrate synthesized sound with real-time rendered visuals must code each side independently, oftentimes in separate programming environments, and somehow bridge the two.

In our experience, the disparity between audio and graphics programming paradigms (spanning code to hardware) limits the scope of the overall application and tends to pull the designer away from writing code that actually *matters*, i.e. code that contributes to an actual sound, visual, or interaction in the final experience.

Our insights from working extensively with existing tools led us to rethink audiovisual programming from first principles: we designed ChuGL not as a bridge between separate audio and graphics pipelines, but as a programming framework wherein audio and graphics are unified at the root, both in terms of user workflow and underlying implementation. ChuGL is designed to be:

- **Unified**: interactive audiovisuals can be coded in a single programming environment, without need to explicitly pass data across separate graphics and audio contexts
- **Strongly-timed**: a graphical workflow integrates into ChucK's strongly-timed concurrent programming model; the designer thinks about programming graphics in the same way that they already think about programming sound.
- **Dual-performance**: real-time sound synthesis and 3D graphics running at high frame rates; neither audio nor graphics is sacrificed for the other.[2]

ChuGL's development was greatly accelerated by an unexpected turn of events in the games industry. In September 2023, Unity Technologies—formerly the champion of indie game development—announced a new "runtime fee" which would charge developers a per-install fee for games made in their Unity engine. The terms of the pricing model were seen as extreme and even predatory for game developers (certain price-per-install to install-count ratios would make the game developer owe Unity *more* money than the game had even made). As educators we realized ChuGL held new cultural importance: to be a free, open-source platform dedicated to protecting the interests of students and creatives.

[1] `https://github.com/ccrma/chunreal`

[2] Historically, game developers rarely prioritize audio (which typically is relegated to file playback with a minimal CPU budget). Researchers such as Perry Cook have argued for a cultural shift in games and other interactive applications towards prioritizing audio via parametric sound synthesis and simulation sharing between audio and graphics. For example, in a boxing game, mesh deformation of someone's face getting punched can be used for graphics *and* to inform audio synthesis [12].

We therefore chose to replace Chunity (ChucK + Unity, in use since 2017) [9] with ChuGL (ahead of our original timetable) in CCRMA's Music, Computing, Design course. It is in this context that we develop ChuGL, not only to provide a new workflow, but from motivations that are educational and cultural.

4. WORKFLOW

ChuGL integrates graphics programming into the strongly-timed audio workflow of ChucK by designing abstractions around the many parallels we observe between graphics rendering and audio synthesis, while also taking advantage of their differences. Informally, we want to make graphics programming in ChuGL "fun", and fun to program in tandem alongside audio.

ChucK	ChuGL	Purpose
UGen	GGen	Synthesis Unit
UGen graph	GGen graph	Audio/graphics state
`=>`	`-->`	Graph connection operator
`dac`	`GG.root()`	Global sink/source nodes
`tick()`	`update()`	Synthesis function for custom generators

Table 1: Parallels between ChucK and ChuGL workflows.

4.1 Graphics Generators (GGens)

ChuGL's first core abstraction is the *Graphics Generator*, or GGen for short. Analogous to how *Unit Generators* form the building blocks for audio signal networks, graphics generators are the base class for all graphical entities which exist in virtual 3D space. A GGen by itself contains transform data (position, rotation, scale) and has no corresponding visual display. Expressivity comes from 1) ChuGL's class library of primitives which includes 3D geometries, lighting, cameras, text, etc. and 2) extending these GGens with custom geometry, materials, behavior, and even subgraphs of other GGens.

4.2 The Scenegraph

GGens can be connected to form *scenegraphs*, another core abstraction in ChuGL. Just as ChucK unit generators may be connected together to form a signal processing network that outputs sound, ChuGL graphics generators can be connected together to form a scenegraph that is drawn to the screen as a 3D scene. While UGen networks might contain oscillators, filters, and envelope generators, a ChuGL scenegraph contains descriptions of transforms, geometry, materials, and other graphical state required for drawing. And just as ChucK processes audio samples for UGens connected to a global `dac` UGen, ChuGL similarly will render GGens connected to a global scene root, `GG.root()`.

```
// connecting a UGen to the audio network
SqrOsc square => dac;

// connecting a GGen to the scenegraph
GCube cube --> GG.root();
```

Listing 1: Similarities between ChucK audio patching and the ChuGL scenegraph.

UGens are connected to each other via the `=>` "chuck" operator, denoting an audio sample input-output relationship; likewise, GGens are connected to each other with the `-->` "gruck" operator, denoting a parent-child transform hierarchy. Through these abstractions, programming graphical systems can be clearly delineated in code similar to programming audio synthesis networks. Listing 1 illustrates these parallels.

ChuGL provides an API to freely modify the scenegraph in user-written code, as well as a custom 3D renderer that draws each frame based on current scenegraph state. Supported graphics features include:

- Transform hierachies
- Various geometry, materials, and lighting types
- Texture mapping and environment mapping
- Transparency
- 3D model loading of several popular file formats
- World-space text rendering
- Custom vertex/fragment/screen shaders
- Post processing effects, including HDR colors, tonemapping, and bloom
- UI library

4.3 The Gameloop

The abstractions for making 3D scenes dynamic also parallel the audio workflow. While sound is shaped by explicitly passing time and changing the state of various UGens over the course of many audio samples, graphics are animated by passing time explicitly in ChucK and changing the state of the scenegraph over time. We call the infinite loop of frame-by-frame updates the *gameloop*, and a per-frame timing event is exposed in ChuGL via the global event `GG.nextFrame()`. Furthermore, graphical behavior can be separated across multiple concurrent ChucK shreds precisely like audio control in ChucK, each running according to its own timing logic. This gives rise to the strongly-timed audiovisual programming paradigm in ChuGL. Shreds A and B in Listing 2 demonstrate these parallels.

```
// Shred A: Updating oscillator frequency
while (true) {
    Math.random2(220, 880) => square.freq;
    100::ms => now;
}

// Shred B: Updating cube rotation
while (true) {
    GG.dt() => cube.rotateY;
    GG.nextFrame() => now;
}

// Shred C: Mapping cube scale to
   oscillator frequency
while (true) {
    square.freq() / 220.0 => cube.scale;
    100::ms => now;
}
```

Listing 2: Similarities between audio and graphics updates (Shreds A and B); mapping a graphics parameter to an audio parameter (Shred C).

4.4 Syncing Graphics and Audio

Because ChuGL is designed to be a *unified* audiovisual programming environment, sharing data between graphics and audio states is trivial—all state lives in the same program so it's already shared by default! Shred C in Listing 2 shows how making the size of a cube change according to

```
// window and scene setup
GG.windowTitle( "sndpeek (ChuGL version)" );
GG.fullscreen();
GG.scene().backgroundColor( @(0,0,0) );

// scenegraph setup (graphics)
GLines waveform --> GG.scene();
GLines spectrum --> GG.scene();

// audio analysis network setup (audio)
adc => Accum accumulator => blackhole;  // audio waveform accumulator
adc => PoleZero dcbloke => FFT fft => blackhole;  // spectrum analyzer

// omitted: initialization code for GGens, UGens, UAnas
// ...

// audio processing
fun void doAudio() {
    while( true ) {
        // waveform: get the most recent audio samples
        accum.upchuck(); accum.output(samplesArray);
        // spectrum: take FFT of waveform data, get magnitude response
        fft.upchuck(); fft.spectrum(responseArray);
        // jump by FFT hop size
        HOP_SIZE::samp => now;
    }
}
// run doAudio() as concurrent shred
spork ~ doAudio();

// graphics loop
while( true ) {
    // map waveform samples to vertex positions
    waveform.geo().positions(map2waveform(samplesArray));
    // map spectrum magnitude response to vertex positions
    spectrum.geo().positions(map2spectrum(responseArray));
    // onto the next graphics frame
    GG.nextFrame() => now;
}
```

Listing 3: Abbreviated implementation of sndpeek in ChuGL. `Lines 2-4`: Setting application window appearance and background color. `Lines 7-8`: Connecting two line renderers to the scenegraph. `Lines 11-12`: Building an audio analysis network to capture waveform and spectrum data from mic input. `Lines 18-29`: Looping an audio shred to regularly fetch new audio analysis data. `Lines 32-29`: Looping the main shred every frame to map the latest audio analysis data to vertex positions in the line renderers.

the pitch of an oscillator is as simple as reading and writing a variable—all the mundane bookkeeping necessary to integrate audio and graphics is implicitly handled by the ChuGL backend so that the programmer can focus on writing creative, meaningful code.

Listing 2 also demonstrates two distinct methods of updating graphical state. Shred B updates graphical state every frame via passing `GG.nextFrame() => now`. This generates smooth frame-rate animations and can be thought of as *continuous updates*. Alternatively, ChucK's strongly-timed scheduler allows making changes under arbitrary timing, shown by Shred C only updating state every 100ms. We call these *discrete updates*, which are especially useful for reacting to one-shot events like musical downbeats. Changes to the scenegraph via *discrete updates* appear in next rendered frame.

4.5 Custom GGens

GGens can be subclassed in ChucK code to group graphical primitives, data, and behavior under a single abstraction that works like any other GGen. In addition, all custom GGens can implement a user-provided `update()` function, automatically called every graphics frame by the ChuGL backend. This programming model parallels writing custom unit generators by extending UGens and overriding the base `tick()` function.

In summary, ChuGL's programming abstractions build upon the many parallels we have observed between graphics rendering and audio synthesis. Where Chuck traverses the UGen graph every sample to compute values for an audiobuffer, ChuGL traverses the scenegraph every graphics frame to compute color values for the display buffer. Finding these parallels allowed us to design ChuGL in a way that naturally integrates with the existing ChucK workflow. These are summarized in the Table 1.

4.6 Putting it All Together

As a more comprehensive example, we demonstrate recreating the sndpeek audio visualizer [18]. Originally written in C++, sndpeek visualizes the current audio waveform alongside a waterfall plot of spectrum history. Listing 3 contains the abbreviated implementation in ChuGL, showing basic window setup, scenegraph and audio graph initialization, and audio visualization with concurrent shreds. Note that audio analysis is performed with *Unit Analyzers* [27], which in the scope of this example can be thought of analogously to UGens. As a quick comparison, the complete ChuGL code[3] for this example is 232 lines counting comments and

[3] `https://chuck.stanford.edu/chugl/examples/sndpeek/sndpeek.ck`

whitespace, while the main C++ sndpeek program[4] is over 1500 lines *not* counting code for thread management, audio analysis, and other necessary bookkeeping. Moreover, ChuGL benefits from using a more modern graphics API and runs at C++ sndpeek's framerate or faster.

This example provides a blueprint of ChuGL's unified workflow for programming graphics and audio, including how each can be mapped to the other. It can be extended to an arbitrary extent and complexity.

5. ARCHITECTURE

ChuGL is implemented as a ChuGin (ChucK plugin) [22]. The ChuGin interface functions as a dynamic binding between ChucK and native compiled code by providing an API to important VM functionality including shred management and garbage collection. At runtime, the VM dynamically loads ChuGL to make its functionality available to all ChucK programs.

ChuGL's unified programming model (as illustrated in Section 4), is made possible by three key aspects of its system-design: 1) the API is retained-mode, 2) audio and graphics run in parallel and 3) a non-blocking synchronization architecture maintains scenegraph state across threads.

5.1 Retained-mode API

ChuGL is a retained-mode graphics API [15], meaning that the library implementation—and not the ChucK programmer—is responsible for handling graphics-related state. The programmer instead controls graphics by indirectly modifying this state via API functions to freely read and modify the ChuGL scenegraph. Note that retained-mode APIs are in contrast to immediate-mode APIs where the programmer has to manage their own state and explicitly issue commands to "redraw" a graphical entity every frame. The benefits of this retained-mode approach are threefold:

1. The programmer does not need to concern themselves with the implementation details of 3D rendering. Like ChucK does for audio, ChuGL hides the mundane while exposing the expressive.

2. ChuGL's hardware-accelerated backend is far more performant than an immediate-mode approach using ChucK for software rendering.

3. Retaining state internally enables the implementation of ChuGL's other key architectures, namely gameloop parallelization and scenegraph synchronization.

5.2 Parallelizing the Gameloop

ChuGL renders graphics on a separate thread that runs in parallel with the ChucK VM thread, which synthesizes audio. We call these the *graphics thread* and *audio thread* respectively, and describe ChuGL's gameloop as implicitly *parallel*. In practice, we found the implementation of a parallel gameloop—where audio and graphics do not block on each other—to be a necessary optimization for running real-time audiovisual applications.

To coordinate scheduling these separate threads, ChuGL takes advantage of the fact that real-time audio has more stringent timing requirements than graphics. Dispatching audio at frame rate from a graphics-driven gameloop lacks the resolution for precise audio-rate timing. Conversely, Chuck's sample-synchronous scheduler has no difficulty with frame-rate timing resolution. Therefore, ChuGL dispatches the graphics thread from the audio thread and provides ChucK with a non-blocking frame event that shreds can wait on to execute once every frame. This two-way coordination process is diagrammed in Figure 1. Note that shreds waiting on the frame event are only allowed to run once the graphics thread has finished drawing the *previous* frame; this enforces that both threads remain at most 1 frame apart, preventing desync bugs and capping the latency for audiovisual correspondence to 1 frame (in practice only a few milliseconds).

[4] `https://soundlab.cs.princeton.edu/software/sndpeek/`

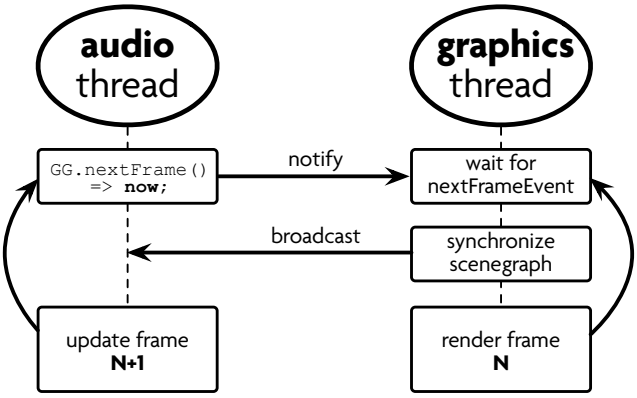


Figure 1: ChuGL's thread coordination process. The audio thread computes scenegraph state for the next frame while the graphics thread renders the current frame.

Under this architecture, performing significant amounts of computation on the audio thread or increasing the audio buffer size can reduce graphical framerate, but long draw times in the renderer will never slow down audio synthesis. This behavior follows our guiding principle of designing an *audio-driven* engine.

5.3 Scenegraph Synchronization

A parallel gameloop necessitates both threads can access scenegraph state safely and performantly. To avoid all the performance and correctness issues associated with data sharing (race conditions, memory safety, etc.) ChuGL instead *duplicates* the scenegraph across both threads. The audio thread controls the source-of-truth scenegraph, which the programmer can freely read and modify. Meanwhile, the graphics thread owns a scenegraph copy that is incrementally updated every frame to stay in sync. No locks are required for reading or writing to one's local copy, ensuring stable performance. Figure 2 diagrams this synchronization architecture.

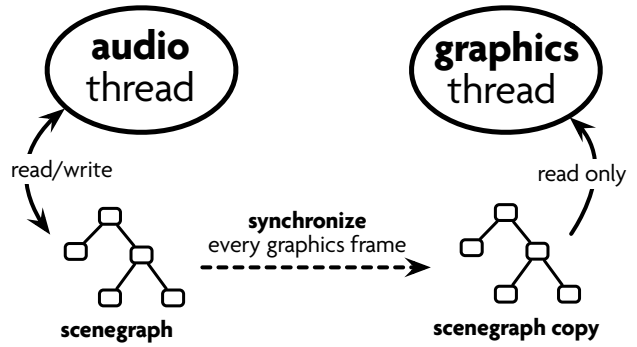


Figure 2: Scenegraph synchronization architecture.

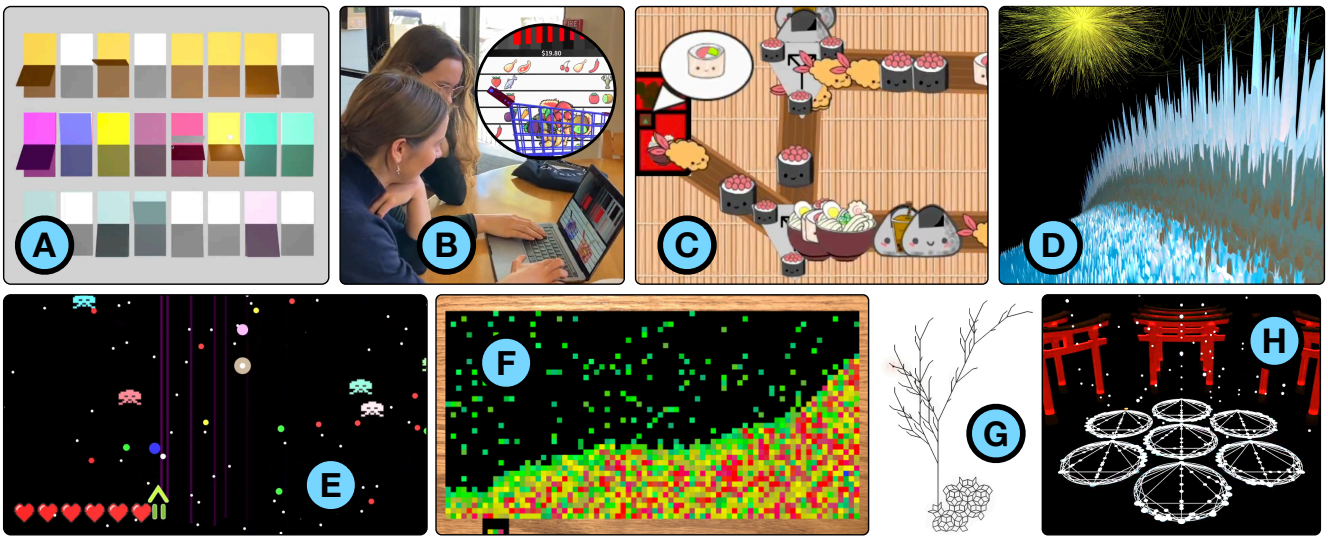


Figure 3: Screenshots from the student showcase. **A:** Splitflap. **B:** Rhythm Market. **C:** Sushi Game. **D:** OceanSide. **E:** ChucK Invaders. **F:** GrainBox. **G:** l-systems & whimsy. **H:** Sounds Like Rain. **See video at** `https://vimeo.com/909845445`.

6. CASE STUDY

ChuGL was the primary teaching tool for the Fall 2023 edition of Stanford's Music, Computing, and Design course. In this course on interactive audiovisual design, 25 students used ChuGL to create audio visualizers, sequencers, musical games, and more. Their projects and feedback gave us real-world insight into what students can create with the tool and, more importantly, how using it made them *feel*. ChuGL's feature set will grow with time; the programming paradigm and way of thinking, however, are core to its identity.

In this qualitative evaluation we showcase select student work and discuss student feedback on an anonymous, optional course exit survey. Our survey is modeled on the questionnaire previously used to evaluate Chunity in 2018, allowing us to directly measure the strengths and weaknesses of ChuGL in comparison to another audiovisual tool.

6.1 Student Showcase

Our showcase contains descriptions of select student projects, images of those projects in action, and a video demo reel. The images and video are provided in Figure 3.

- **Splitflap**: An audio interaction/instrument that explores stochasticism, time and change.
- **Rhythm Market**: A competitive two-player, grocery shopping rhythm game.
- **Sushi Game**: A game, instrument, and sequencer about operating a conveyor belt sushi restaurant.
- **OceanSide**: Explore a world where sound is transformed into mountains, ocean waves, and lighting.
- **Chuck Invaders**: Musical space invaders! All weapon pickups correspond to musical instruments, and a generative soundtrack evolves as the player progresses through levels.
- **GrainBox**: A granular sequencer and sand simulation where the grains of sand are also grains of sound.
- **l-systems & whimsy**: An instrument for constructing musical sequences with Lindenmayer Systems.
- **Sounds Like Rain**: A horror game built around an "Irrational Rhythm" sequencer which depicts the sound of rain.

	Mean ± S.D.	**Min, Max**
Years Music Training	12.41 ± 5.88	[0, 20]
Years Coding	5.5 ± 2.64	[2, 11]
Years ChucK	$.40 \pm 0.87$	[0, 3]
Years Graphics	0.16 ± 0.32	[0, 1]

Table 2: Student Demographics. Most had prior musical and general programming experience but were new to graphics and audio programming.

6.2 Reported Experience

The exit survey included several questions asking students to freely reflect on aspects of the class, including:

Did ChuGL encourage a way of thinking about audiovisual programming that made sense?

- "It felt like I could think about audio and visuals equally when sculpting a narrative."
- "Being able to think critically about the audio and visual timing of whatever art I was creating helped with integrating the two, especially since they were in the same language.
- "Using ChuGL actually felt incredibly intuitive once I was familiar with how it worked. It really encouraged me to simultaneously think about the audio and visual components of a given project, instead of only one or the other."

How does using ChuGL compare to other tools you've used for audiovisual programming?

- "ChuGL felt way more intuitive and easier to use, since there wasn't any extra back-and-forth between ChucK and Unity that I had to account for; I could just dive right into audiovisual programming straight away with ChuGL."
- "Rigorous, learner-focused documentation was the largest shortcoming by far when it came to moving ideas from my head into reality. I found myself struggling a lot with what would be easily google-able questions about another language"
- "Felt much more primitive and first principle, but very approachable whereas when I first used Unity I felt like the learning curve was more difficult."

In summary, students found ChuGL intuitive, and typically began their assignments by designing the graphical aspects first. Once familiarized with the workflow, students with prior audiovisual programming experience found ChuGL easier to use than other tools; however, many felt that the lack of online learning resources made ramping up more difficult than necessary.

6.3 Comparison with Chunity

	Chunity	ChuGL
I could prototype quickly	**3.38**	**3.92**
Controlling graphical timing was satisfying	**3.09**	**3.92**
UGens were satisfying	**4.09**	**3.67**
I felt empowered	4.59	4.67
Controlling audio timing was satisfying	4.05	4.0
GGens were satsifying	N/A	4.33

Table 3: Exit survey results for Chunity and ChuGL (from 2018 and 2023, respectively). Statements where values differ significantly are in bold. A value of N/A means the statement was not present on the survey.

In the course survey, students were given statements about their class experience to which they could respond "Strongly Disagree - Disagree - Neutral - Agree - Strongly Agree." These statements were a superset of those used in the Chunity questionnaire and were quantified in identical fashion: responses were assigned values 1 (strongly disagree) through 5 (strongly agree) and averaged. Table 3 presents those averages, comparing scores between ChuGL and Chunity.

The two areas where ChuGL scores higher are rapid prototyping and controlling graphical timing, validating our design goals. ChuGL is simpler than Unity and does not require lengthy build times to test code; these factors facilitate prototyping. Furthermore, responses show ChuGL's unified workflow is more satisfying to students than Chunity's global events and callback handlers.

The one statement where ChuGL scores lower is using UGens for audio synthesis. This is surprising because UGens are no different between Chunity and ChuGL. We attribute this decrease in satisfaction to our own pedagogy—creating a graphics library while teaching the course meant we naturally placed greater emphasis on graphics, and had less time to guide students through audio synthesis.

These survey responses are not intended to make any objective statements of quality; rather, they serve to distinguish ChuGL from Chunity and other tools for programming audiovisual applications. We plan to create more structured learning content to teach students the novel style of audiovisual programming which ChuGL offers.

7. REFLECTIONS

Why are audio and graphics historically separate? Conway's Law, introduced by computer scientist Melvin Conway in 1967, states the structure of a product mirrors the communication structure of the organization which produced it [11]. When organizations delegate work to teams and individuals, they partition a holistic idea into subfields and impose barriers between them. Audio and graphics are almost always delegated to separate teams, and because there is greater communication friction across teams than within, the end product exhibits this separation as well. Hence operating systems provide separate graphics and audio APIs, game engines feature separate rendering and audio systems, and, to the user, audio and graphics are understood as separate entities.

The extent to which audio and graphics are separated is the extent to which the potential for a shared design is diminished. Separation increases the likelihood that one side is prioritized at the expense of the other (historically audio has often been shortchanged wherever graphics are also involved, especially video game development). In addition, this separation is a fundamental contributor to complexity in audiovisual programming, which requires working against the communication friction inherent between separate software and hardware systems.

ChuGL, in contrast, emerges from a belief that the boundaries between audio and graphics are arbitrary, and that the unification of both entities into a single audiovisual tool and workflow is not only possible, but also holds technical and artistic potential. As *Artful Design* Principle 3.1 states, "Design sound, graphics, and interaction together." [24] This design ethos is the overarching goal of ChuGL. As a tool, ChuGL's architecture, including its retained-mode API and efficient synchronization mechanisms, integrate real-time graphics and audio without sacrificing capabilities of either. And as a workflow, ChuGL reveals the benefits of this closer integration, which include a unified audiovisual programming model and implicit time and data synchronization across audio and graphics contexts. Furthermore, ChuGL programs have access to the entirety of ChucK's growing toolset–for example, audio synthesis, analysis [27], and interactive AI[5]–along with the potential for shared design therein.

In its pilot course at Stanford University, students used ChuGL to craft rich audiovisual experiences. These experiences pointed to limitations of the tool and inspired directions for future development. But more importantly, they demonstrated that ChuGL's unified audiovisual workflow—as a way of thinking and doing—empowered a diverse student body with varying experience in either medium.

Although ChuGL is still in its infancy, it builds on a history of work arising from shared premises about audiovisual integration. What we present in this paper is the third iteration in a lineage beginning 20 years ago with Philip Davidson's GlucK in 2004, followed by Spencer Salazar's ChuGL in 2014. Each iteration was made possible by engineering lessons from the prior, as well as advancements in modern programming languages, hardware, and graphics APIs.

We are working towards the long-term goal of making ChuGL a mature platform for audiovisual application development by computer music practitioners and indie game developers alike. This means creating more learning resources, developing new features, and using the tool ourselves to build larger-scale applications, including audio-driven video games and instruments that integrate audio synthesis, graphics, and human interaction for live performance. In fact ChuGL has already been used by Celeste Betancur for her audiovisual live-coding sets in venues worldwide. Currently, we are migrating the underlying graphics API from OpenGL to WebGPU, adding native support for 3D spatial audio in ChucK, and enabling ChuGL to run on the web via integration with WebChucK IDE [14, 19]. Other promising directions for ChuGL include intermedia art installations and live-coding audiovisual performances.

Looking forward, we aspire for ChuGL to not merely simplify making audiovisual applications, but to empower a fundamental difference in kind. There is a horizon of audiovisual design waiting to be explored as tools are increas-

[5] `https://chuck.stanford.edu/chai/`

ingly unified, communication friction is reduced, and multimodal, multi-disciplinary ways of thinking are encouraged. Exploring this world means exploring our own aural, visual selves. Tools (and their design) matter.

8. ACKNOWLEDGMENTS

Thanks to all the students of Music 256a / CS 476a at Stanford University for their creativity, feedback and patience being the first users of ChuGL, to Perry R. Cook for inspiring the authors to prioritize expressiveness and play in audiovisual programming (and for PongViewGL), and to Philip Davidson and Spencer Salazar for their respective past efforts integrating graphics with ChucK, without which this work would not be possible.

9. ETHICAL STANDARDS

ChuGL has been developed with the support of CCRMA's departmental funding, curricular student research, and volunteer contributions. The authors are aware of no potential conflicts of interest. The ChuGL course survey was anonymous and optional, with no implication for course grades. Students were made aware that their responses on the survey could be used anonymously in a research paper.

An alpha version of ChuGL is included as part the standard ChucK distribution (as of language version 1.5.2.1). For documentation and more, visit:
`https://chuck.stanford.edu/chugl/`

10. REFERENCES

[1] fluxus. `https://www.pawfal.org/Software/fluxus_/`. Accessed: 2024-02-03.

[2] hydra, live coding video synth. `https://hydra.ojack.xyz/`. Accessed: 2024-02-03.

[3] Jitter | cycling '74. `https://cycling74.com/products/jitter`. Accessed: 2024-02-03.

[4] openframeworks. `https://openframeworks.cc/`. Accessed: 2024-02-03.

[5] Processing. `https://processing.org/`. Accessed: 2024-02-03.

[6] Touchdesigner. `https://derivative.ca/`. Accessed: 2024-02-03.

[7] Vulkan - cross platform 3d graphics. `https://www.vulkan.org/`. Accessed: 2024-02-03.

[8] Wwise. `https://www.audiokinetic.com/en/products/wwise/`. Accessed: 2024-02-05.

[9] J. Atherton and G. Wang. Chunity: Integrated audiovisual programming in unity. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 102–107, June 2018.

[10] J. Atherton and G. Wang. Doing vs. being: A philosophy of design for artful vr. *Journal of New Music Research*, 49(1):35–59, 2020.

[11] M. E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[12] P. R. Cook. *Real Sound Synthesis for Interactive Applications*. A. K. Peters, Ltd., USA, 2002.

[13] P. R. Cook and G. P. Scavone. The synthesis toolkit (stk). In *ICMC*, 1999.

[14] T. Feng, C. Betancur, M. R. Mulshine, C. Chafe, and G. Wang. Webchuck ide: A web-based programming sandbox for chuck. 2023.

[15] H. Igehy, G. Stoll, and P. Hanrahan. The design of a parallel graphics interface. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 141–150, 1998.

[16] T. Magnusson. The ixi lang: A supercollider parasite for live coding. In *ICMC*, 2011.

[17] A. McLean and G. Wiggins. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*, pages 331–334, 2010.

[18] A. Misra, G. Wang, and P. Cook. Sndtools: Real-time audio dsp and 3d visualization. In *ICMC*, 2005.

[19] M. R. Mulshine, G. Wang, J. Atherton, C. Chafe, T. Feng, and C. Betancur. Webchuck: Computer music programming on the web. In *New Interfaces for Musical Expression*, 2023.

[20] M. Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.

[21] C. Robinson. *Game Audio with FMOD and Unity*. Routledge, 2019.

[22] S. Salazar and G. Wang. Chugens, chubgraphs, chugins: 3 tiers for extending chuck. In *ICMC*, 2012.

[23] G. P. Scavone. Rtaudio: A cross-platform c++ class for realtime audio input/output. In *ICMC*. Citeseer, 2002.

[24] G. Wang. *Artful Design: Technology in Search of the Sublime, A MusiComic Manifesto*. Stanford University Press, 2018.

[25] G. Wang, N. J. Bryan, J. Oh, and R. Hamilton. Stanford laptop orchestra (slork). In *ICMC*, 2009.

[26] G. Wang, P. R. Cook, and S. Salazar. ChucK: A Strongly Timed Computer Music Language. *Computer Music Journal*, 39(4):10–29, 12 2015.

[27] G. Wang, R. Fiebrink, and P. R. Cook. Combining analysis and synthesis in the chuck programming language. In *ICMC*, 2007.

[28] M. Wright. Open sound control: an enabling technology for musical networking. *Org. Sound*, 10(3):193–200, dec 2005.