CollideFx: A Physics-Based Audio Effects Processor

Chet N. Gnegy
Center for Computer Research in Music and Acoustics (CCRMA)
Stanford University
chet@ccrma.stanford.edu

ABSTRACT

CollideFx is a real-time audio effects processor that integrates the physics of real objects into the parameter space of the signal chain. Much like a traditional signal chain, the user can choose a series of effects and offer realtime control to their various parameters. In this work, we introduce a means of creating tree-like signal graphs that dynamically change their routing in response to changes in the location of the unit generators in a virtual space. Signals are rerouted using a crossfading scheme that avoids the harsh clicks and pops associated with amplitude discontinuities. The unit generators are easily controllable using a click and drag interface that responds using familiar physics. CollideFx brings the interactivity of a video game together with the purpose of creating interesting and complex audio effects. With little difficulty, users can craft custom effects, or alternatively, can fling a unit generator into a cluster of several others to obtain more surprising results, letting the physics engine do the decision making.

Keywords

digital signal processing, effects generator, dynamic routing, physics

1. INTRODUCTION

From guitar stompboxes to the effects chains in digital audio workstations, it is taken for granted that we can cascade audio effects together and provide realtime control to some subset of the effect parameters. A notable example of each includes the wah-wah pedal and MIDI envelopes, the latter of which can be either drawn into the software or controlled via an external MIDI interface. Scrambling the order of the effects in the signal chain on the fly, however, is a much less conventional feature in an effects processor. The combinations of nonlinear audio effects, such as chorusing, distortion, or realtime granulation can produce interesting and unexpected results when given this type of freedom. Additionally, we do not restrict our signal chains to a single path, we allow tree-like graphs to be constructed, branching the signal through an arbitrary number of paths. To encourage exploration of this dynamic environment, we assign each unit generator to a virtual object, a disc. These discs are placed within a square world and given properties of real objects; they move and rotate with realistic physics, collide with one another, and are subject to friction. The notion of pairing a unit generator with an object and routing signals based on proximity is also used in the popular reacTable [1], which uses physical objects and computer vision to interact with the audio system. Visual programming languages Pure Data [2] and Max/MSP [3] also feature a convenient means of rerouting effects, however, avoiding clicks and pops due to discontinuities in the audio signal is the responsibility of the user. In a later section, we discuss the rerouting scheme for CollideFx, which uses crossfades to automatically eliminate these undesirable effects.

The graphical interface is shown in Figure 1. Here we see a complex signal graph, and the connections between are visualized by colored, partially transparent orbs that originate at source unit generators and flow through the graph until they reach an output unit generator and vanish. Like the discs, the orbs move with physics. Due to a combination of repulsive and attractive forces, similar to electrons around a nucleus, the orbs form a cloud surrounding their parent disc. Each type of source creates orbs of a unique color scheme, allowing the path for any particular source to be traced through the graph.

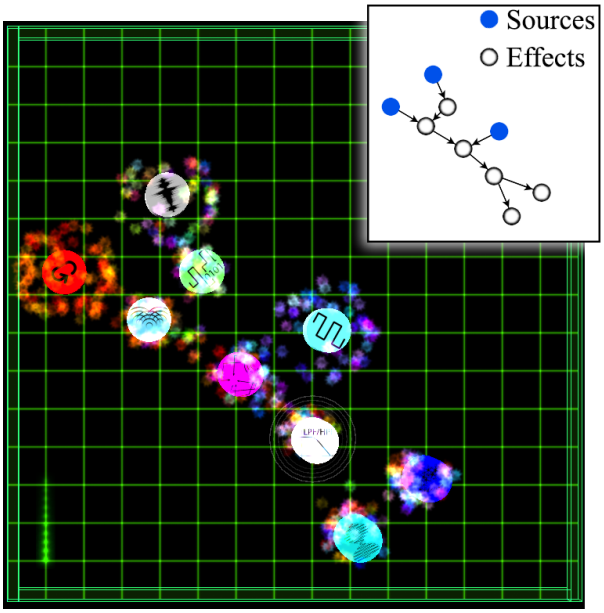


Figure 1: A screenshot of CollideFx. The `Input`, `Square`, and `Looper` modules are connected to network of effects. The connectivity of the signal flow graph is shown in the upper right corner. The edges in the graph are represented by a stream of continuously flowing orbs that originate at sources and travel along the path of the signal.

NIME'14, 30 June–3 July 2014, London, England.

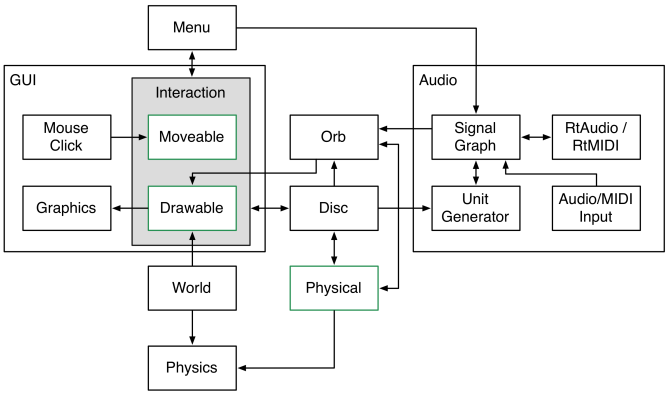


Figure 2: System Diagram

2. SYSTEM ARCHITECTURE

CollideFx is implemented entirely in C++ and utilizes the OpenGL libraries for the graphical interface and user interaction. A basic block diagram is shown in Figure 2. This interface allows users to click and drag to create, move, and delete discs with very little effort. Discs first enter the world when they are dragged from their corresponding menu icon onto the world. At this point, discs containing source unit generators create a cloud of orbs and begin interacting with the other unit generators via the signal graph. The mouse click information is relayed through the graphical interface to the discs. A clicked disc calculates a spring force between itself and the mouse location as it is moved. Collisions between discs and against the boundaries of the world are also handled within the physics engine, which updates the discs positions inside of the graphics thread. The menu, which can be seen in the Appendix, is also used to change audio parameters and delete discs. The main interaction with the audio thread is through the discs and their unit generators. It is important to note that a disc is always paired with a unit generator. While it is usually clear to use the terms "disc" and "unit generator" synonymously, we will reserve the term "disc" for the discussion the physical or graphic properties of the object, and "unit generator" when discussing the audio properties of the object.

The RtAudio/RtMIDI [4] framework is at the heart of the audio system. Using RtAudio/RtMIDI callback functions, MIDI events and buffers of audio input are sent to the signal graph. All audio and MIDI data are then passed to the audio or MIDI source unit generators, if any currently exist. The signal graph uses the positions of the discs and their respective unit generators to build a graph of connections. This graph passes these buffers between the unit generators for every audio buffer and sends the result to RtAudio. Furthermore, if the connections of the graph have changed since the previous buffer, the graph performs a crossfade between the previous graph topology and the current one. To prevent multiple graph crossfades from occurring at once, only one graph change is allowed per audio buffer. An audio buffer size of 512 samples therefore produces a short crossfade of the same length. On a regular interval, any orbs that are currently floating around a disc are passed downstream to another disc.

2.1 Audio Processing and Dynamic Routing

2.1.1 Unit Generators

CollideFx features a diverse variety of unit generators, subdivided into two categories: source and effects. The source unit generators include the `Input`, taken from the microphone or line in, and several classic waveforms that respond to MIDI events, namely the `Sine`, `Square`, `Tri`, and `Sawtooth` waves. All classic waveforms (with the exception of the sine wave) are generated using the first 15 terms of their Fourier series representations. This is done to avoid the harsh sound caused by sharp discontinuities in amplitude of their first derivative that are present in the ideal waveforms.

The effect unit generators featured are the `BitCrusher`, `Chorus`, `Delay`, `Distortion`, `Filter`, `Granulator`, `Looper`, `Ring Modulator`, `Reverb`, and `Tremolo`. The `Looper` unit generator that is routed like an effect while in Off/Recording mode, but acts like a source once it is in Playback mode.

All unit generators have a state, which can be saved and recalled at a later point in time. The state objects contain every piece of information necessary to uniquely describe a unit generator at a given point in time. Depending on the type of unit generator, this can involve storing anywhere from a couple of variables to an entire buffer of audio data. For example, the `Delay` module must store a whole buffer, but the `Ring Modulator` must only recall the phase and frequency of the modulating signal. As mentioned in the previous section, we avoid sharp discontinuities by crossfading between graphs as the routing changes. It is for this reason that we must store and recall states; the crossfades require a buffer of audio from two different graphs, the previous and the current graph, for the same set of samples.

2.1.2 Signal Graph

The signal graph is constructed based on the locations of the discs to one another. A minimal spanning tree connecting the discs is computed using a modified version of Prim's Algorithm [5], an algorithm for creating a fully connected graph with minimal edge lengths. This provides an intuitive looking graph, (i.e., the discs closest to each other are connected). It also prevents the formation of cycles, which would cause the system to possibly become unstable. The algorithm is modified in three ways. First, we prevent two source discs from being connected to each other. Next, we do not allow edges to be created if the distance between two discs, λ, exceeds some maximum distance, L_{max}. This may result in a disjoint graph. Because Prim's algorithm doesn't terminate until the graph is no longer disjoint, we include a final modification to allow the algorithm to terminate if the only remaining possible connections are greater in length than L_{max}. An additional complication involving the `Looper` unit generator is that it switches from an effect to a source once it is in Record mode; this causes the graph to be rerouted. The sound that the `Looper` recorded will then be repeated as its output.

Any unit generator at the end of a branch is implicitly declared to be a sink and therefore sends its output to RtAudio, as seen in Figure 2. Once the edges of the graph have been established we must establish their directionality, which is done iteratively. We start by ensuring that any discs labeled as sources are on the "from" end of the edge. The discs on the "to" end of these edges must naturally be upstream from any connected non-source discs. We iteratively change the directionality of edges from the beginning to the ends of the chain to make sure that paths are leading away from the sources.

Once the graph is computed, each disc stores its current input and output connections in a data structure. In order to obtain the audio output for the entire graph, we poll the unit generators of the discs that have been designated to be sinks for their audio buffer. We recursively work our way from the sinks back to the sources and then apply each effect on the return journey through the tree. At each step, we use each disc and its inputs (which can be sources or other

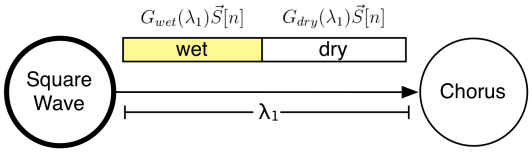


Figure 3: The wet/dry mix is computed based on the proximity between two discs. The distance λ_1 cannot exceed L_{max} or the connection will not be made. The Square Wave unit generator is shown with a thick outline because it is a source.

effects) to determine the wet/dry mix using Equations 1 and 2. Two touching discs have a mix level of 100%. As we increase the distance, λ, between two discs to L_{max}, we reduce the mix level to 0%, at which point orbs no longer flow between the two discs. For each of the inputs to any particular unit generator, we create two N-sample buffers, a wet and a dry buffer. We use buffer lengths of $N = 512$. The summation of only the wet buffers is processed by the unit generator and then recombined with the summation of the dry buffers. Equation 3 generalizes the calculation of the output for some effect, $f(\vec{x}[n])$, for M inputs, $\vec{x}_i[n]$. Additionally, to maintain the perceived volume of the system when the graph has many branches, we scale down input contributions by a factor of $1/\sqrt{k}$, where k is the fanout of the previous unit generator in the path.

$$G_{wet}(\lambda) = \left(1 - \frac{\lambda}{L_{max}}\right) \tag{1}$$

$$G_{dry}(\lambda) = \frac{\lambda}{L_{max}} \tag{2}$$

$$\vec{y}[n] = f\left(\sum_i^M G_{wet}(\lambda_i)\vec{x}_i[n]\right) + \sum_i^M G_{dry}(\lambda_i)\vec{x}_i[n] \tag{3}$$

For example, consider the system in Figure 3. The Square Wave unit generator produces a signal $\vec{S}[n]$, which is passed to a Chorus effect. The discs are a distance λ_1 apart. We use Equations 1 and 2 to decompose the buffer into wet and dry components, $\vec{S}_{wet}[n]$ and $\vec{S}_{dry}[n]$ respectively. Only the wet mix is processed by the Chorus, $c(\vec{x}[n])$. The output of the Chorus is then $c(\vec{S}_{wet}[n]) + \vec{S}_{dry}[n]$; this is the simple result of using Equation 3 for a single input. We introduce an `Input` in Figure 4 to detail how a single effect can have multiple sources.

We then check to see if the connections of the graph have changed since the last iteration. For the case where no changes have occurred, we ask each sink disc to compute its audio buffer and sum them together, as previously detailed. When a change has been made, however, a crossfade is necessary and the graph must be processed twice. Before doing this, we store the state of each unit generator. As the graph is processed the first time, we store the incoming audio buffer for each unit generator. These stored buffers will be used in the second processing step. Following this operation, we recall the states of all of the unit generators and proceed to reprocess the graph using the new, changed set of edges. This time, we perform a crossfade using the stored audio buffer from the first pass and the new incoming audio buffer prior to applying an effect. Computing the crossfade on the incoming buffer keeps the internal effect buffers (in the case of the `Looper`, `Granulator`, etc.) free of discontinuities.

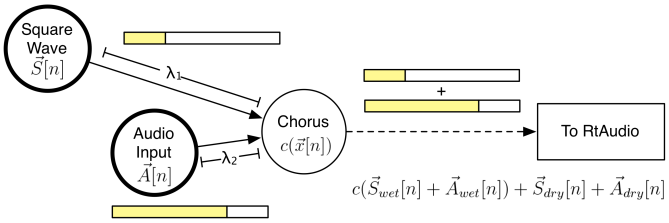


Figure 4: An example of an audio input and a Square Wave being fed into a Chorus effect. Each input has its own mix level. The wet buffers are summed together and processed by effect $c(\vec{x}[n])$. The processed output is summed with the dry buffers as described by Equation 3. If there is not another node in the graph, the output is sent to RtAudio.

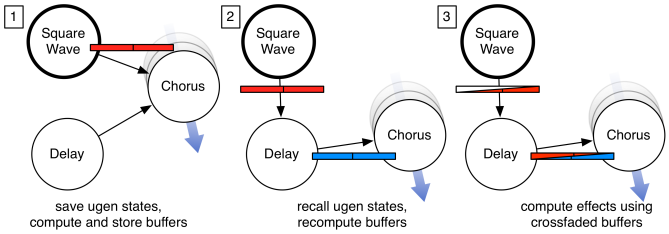


Figure 5: When the moving Chorus disc causes the signal graph to change, we crossfade buffers to prevent discontinuities. Buffers colors are associated with the discs that provided their audio data. (1) First, all unit generators store their current state and a copy of their incoming buffer. The delay is not driven by anything and only contributes an empty buffer (not shown for simplicity). (2) Then, the states are recalled and the buffers are recomputed using the new signal graph. (3) Prior to applying an effect, we crossfade between the old incoming buffer and the new one. The diagonal line through the buffer symbolizes a crossfade.

This process is illustrated in Figure 5. As we see in this example, the Chorus unit generator stores its incoming buffer (from the Square Wave, shown in red). On the second pass, it receives a buffer from the Delay (shown in blue). Prior to processing the its effect, it crossfades between the old and new buffers (shown as a red buffer that tapers off into the blue buffer). The Delay was not driven by anything initially, so upon receiving the new Square Wave input, it simply fades in to the buffer from the Square Wave (shown as an empty, white buffer fading to the red buffer).

2.2 Physics

The translational and rotation motion of the discs is computed using numerical integration and is updated 20 times for every refresh of the graphics (which is nominally 50fps). 20 iterations per frame was determined experimentally as a compromise between choppy looking movement and excessive amounts of computation. We consider all external forces and torques from the individual discs and orbs. This includes the spring force due to mouse dragging, the forces that keep the orbs hovering around the discs, air drag, and the damping torque that opposes the angular velocity of discs. The instantaneous acceleration is obtained using Newton's second law. We update the translational velocities and positions of each object according to the formulae $\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t)\Delta t$ and $\vec{s}(t + \Delta t) = \vec{s}(t) + \vec{v}(t)\Delta t$,

respectively. Likewise, we recalculate the angular components, $\vec{\omega}(t + \Delta t) = \vec{\omega}(t) + \alpha(\vec{t})\Delta t$ and $\vec{\theta}(t + \Delta t) = \theta(\vec{t}) + \omega(\vec{t})\Delta t$.

Whenever the discs come close to the wall, or close to another disc, a completely elastic collision occurs. In this case, the velocity component normal to the collision changes sign. Without going into the vector arithmetic, we use this change in velocity to find the impulse of the collision. This impulse, along with the component of the velocity tangential to the collision, is used to slow the disc down and to cause it to rotate. This occurs because of friction between the two colliding objects. In future versions, disc rotation may be mapped to an audio parameter, but it is currently only a feature of the physics engine.

3. CONCLUSIONS

CollideFx offers a simple and intuitive means of building complex effects chains. Furthermore, it allows the user to change order of the unit generators in real time without the introduction of amplitude discontinuities. It is convenient to build effects using multiple input sources and arbitrary numbers of branches that would normally require more complex routing and the use of bus channels. This type of flexibility is not typically offered in effects processors and can create very interesting and unexpected results.

4. LINKS

The CollideFx homepage (with video demo):
`http://www.chetgnegy.com/design/CollideFx.html`

5. ACKNOWLEDGMENTS

Thanks to Julius Smith and Ge Wang for advice and support.

6. REFERENCES

[1] S. Jorda, M. Kaltenbrunner, G. Geiger, and R. Bencina, "The reacTable*," in *In Proceedings of the International Computer Music Conference (ICMC 2005*, pp. 579–582, 2005.

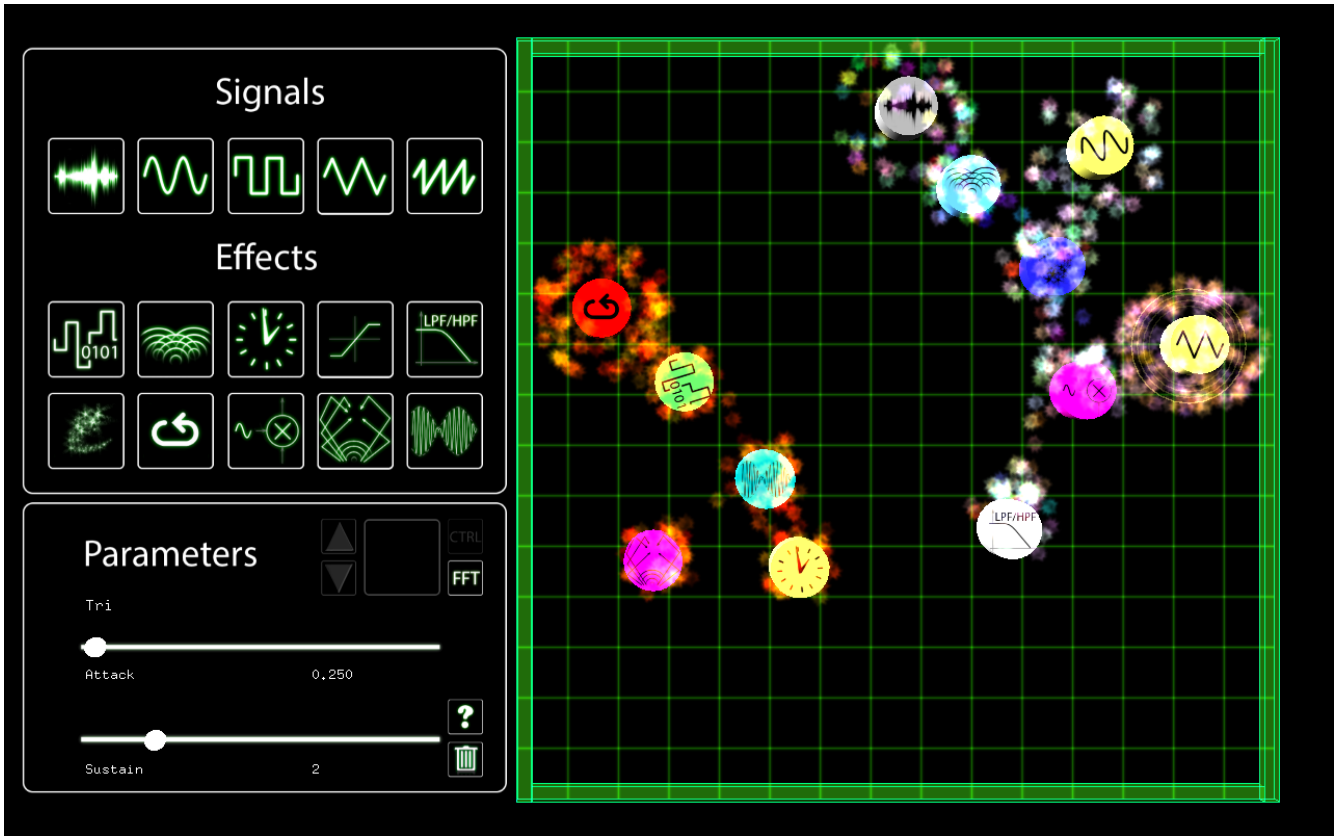
[2] M. Puckette, "Pure data: Another integrated computer music environment.," in *Proceedings, Second Intercollege Computer Music Concerts, Tachikawa, Japan*, pp. 37–41, 1996.

[3] M. Puckette, "Combining event and signal processing in the max graphical programming environment," *Computer Music Journal*, vol. 15 (3), pp. 67–77, 1991.

[4] G. Scavone, "RtAudio and RtMIDI." `http://www.music.mcgill.ca/~gary/`, 2013. Accessed: 2014-01-17.

[5] R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, vol. 36, pp. 1389–1401, 1957.

APPENDIX



Two disjoint graphs, the leftmost being driven by the Looper, and the upper right graph being driven by the `Sine`, `Input`, and `Tri` (triangle wave) unit generators. The `Tri` unit generator is currently selected. This allows users to edit the `Tri` parameters using the lower half of the menu.