Physically Based Sound for Computer Animation and Virtual Environments

Tips for real-time audiovisual simulation

Doug James
Stanford University

Context: HW1

HW#1: Acceleration Noise

http://graphics.stanford.edu/courses/cs448z/HW_Sp21/HW1/CS448Z_2021sp_HW1_RBD_AccelNoise.pdf

PART I: Rigid-body Simulation

- "Spheres in a box"
- Discrete *or* continuous detection (*your choice*)
- Rigid-body Impact: Use collision time and impulse magnitude for...

PART II: Synthesizing acceleration noise

- Acceleration noise model
- Contact timescale, τ, definition
- Acceleration profile in time; smooth approximation

Simple demo of click sounds

Using ChucK: http://chuck.stanford.edu

Chuck Demo File:

https://www.dropbox.com/s/wwbvmq23sgt6uzk/expClickDemo.ck

Plays acceleration sounds $\dot{S}(t;\tau)$ (10) with varying τ values:

$$S(t;t_0,\tau) \approx exp\left(-\frac{6}{\tau^2}\left(t-t_0-\frac{\tau}{2}\right)^2\right), \qquad (9)$$

which has the derivative

$$\dot{S}(t;t_0,\tau) = -\frac{12}{\tau^2}\left(t-t_0-\frac{\tau}{2}\right)\ S(t;t_0,\tau), \qquad (10)$$

Just this part

Linear superposition of "click" sounds

For each sphere, for each impact event, play a physical "click" sound at the appropriate delayed time:

$$p(\boldsymbol{x}, t) \approx \frac{\rho_0 a^3 \cos\theta}{2c\|\boldsymbol{x} - \boldsymbol{x}_0\|} \frac{\partial^2 V}{\partial t^2}\left(t - \frac{\|\boldsymbol{x} - \boldsymbol{x}_0\|}{c}\right), \qquad (1)$$

where $\boldsymbol{x}$ is the listening position, $\boldsymbol{x}_0$ is the center of mass of the sphere, c is the speed of sound in air ($\approx 330 m/s$), ρ_0 is the density of air at STP (at 20 ^oC and 101.325 kPa, dry air has a density of 1.20 kg/m^3), and $\cos\theta = \cos(\hat{V}, \boldsymbol{x} - \boldsymbol{x}_0)$ is the cosine of the angle between the direction of oscillation and the direction to the listener.

Synchronizing Graphics & Audio Updates

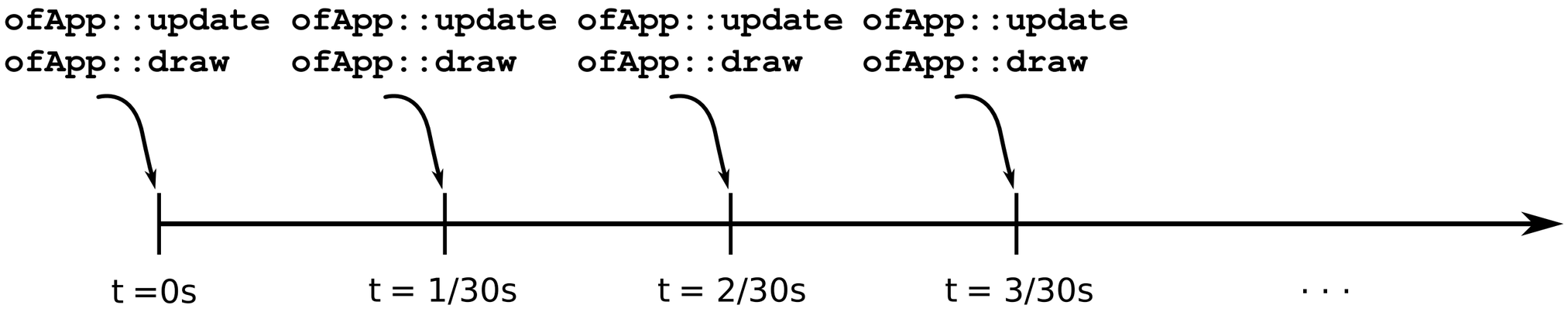
Synchronizing Graphics & Audio Updates

Fixed-rate audio buffer updates

Fixed or variable graphics frame rates. (consider fixed for now)

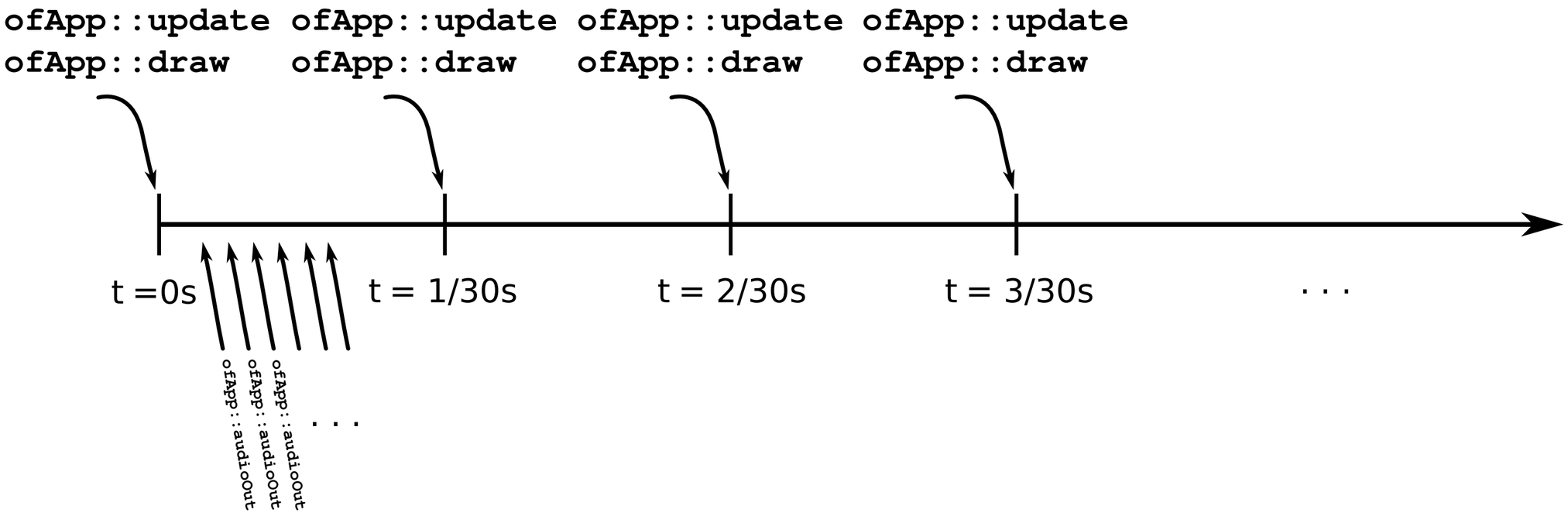
Synchronizing Graphics & Audio Updates

Example: Multi-threaded ofApp



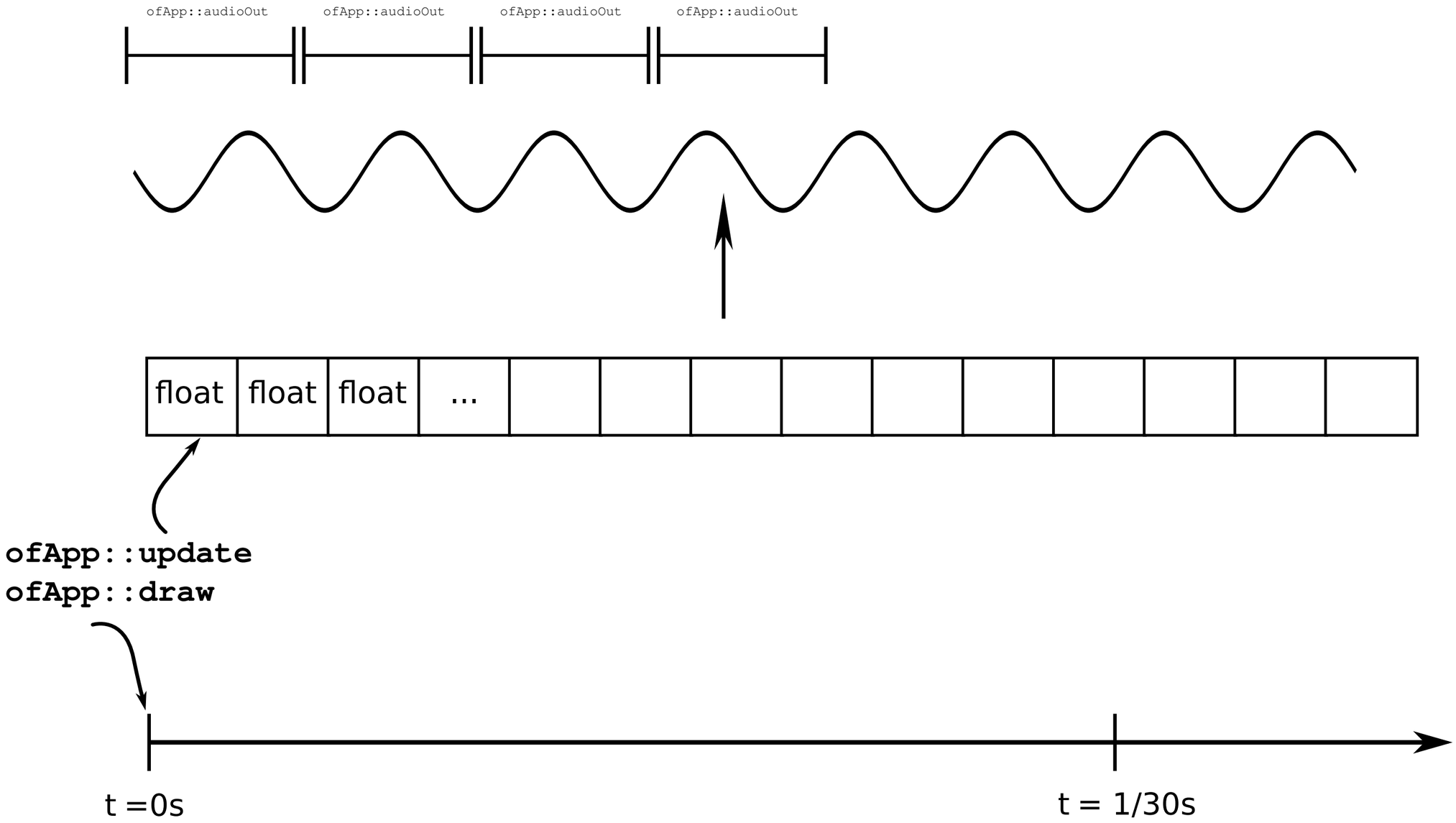
Synchronizing Graphics & Audio Updates

Example: Multi-threaded ofApp



Synchronizing Graphics & Audio Updates

Example: Multi-threaded ofApp with audio buffer



Synchronizing Graphics & Audio Updates

Example: ofApp

Different synchronization strategies with various pros & cons:

1. **Just-in-time audio:**
   - Graphics `update()` builds & maintains a list of collision events and sound-related data needed to evaluate a function `sound(t)`
   - `audioOut()` evaluates `sound(t)` to generate sound samples on the fly
   - PRO: No sound buffering required by graphics `update()`.
   - CON:
     - Potentially high `sound(t)` evaluation overhead in `audioOut()`
     - Only for simple `sound(t)` implementations
2. **Cache audio on graphics loop:**
   - Graphics `update()` fills audio buffer (summing all current "click" sounds)
   - `audioOut()` copies from the buffer.

Buffering Audio Updates

Fixed-rate audio buffer updates

Fixed or variable graphics frame rates.

Challenges:

- Keeping audio buffer filled and in sync with graphics.
- Simulating motion ahead of time, or delaying sound as needed.
- What happens if buffer is read while it is being written?
  - Need to make code thread-safe
    - Identify safe read/write regions.
    - May need to lock/unlock resource for producer/consumer model
      - `std::mutex` or `ofMutex` : "Mutual exclusion lock" or *mutex*
        - https://openframeworks.cc/ofBook/chapters/threads.html#ofmutex

HW0 Pseudocode (Raj Setaluri, 2015)

```
1  // ##############################################################################
2  //
3  // CS 448z, Stanford, Autumn 2015
4  // Homework 0
5  // Author(s): Raj Setaluri
6  //
7  // NOTE: This code is a mix of real syntax and pseudo code.
8  //       It will not compile as is!
9  //
10 // ##############################################################################
11
12 //------------------------------------------------------------
13
14 struct Particle  // particle structure
15 {
16     ofVec3f x,v; // position and velocity
17     float e;     // coefficient of restitution
18
19     Particle()
20         :x(),v(),e((float).6)
21     {}
22 };
23
24 Particle particles[NUM_PARTICLES];    // particles
25 std::vector<float> click_sound;       // *internal* sound buffer
```

setup()

```
28
29  void ofApp::setup(){
30      ofEnableSmoothing();
31      ofSetFrameRate(FRAME_RATE);
32
33      // randomize initial state of particles
34      for(int i = 0;i < NUM_PARTICLES; i++){
35          particles.x = random;
36          particles.v = random;
37      }
38
39      // setup OF sound stream
40      buffer_size = 512;
41      sample_rate = 44100;
42      sound_stream.setup(this, 1, 0, sample_rate, buffer_size, 2);
43
44      // setup *internal* sound buffer
45      audio_size = sample_rate;
46      audio.assign(audio_size, 0.);
47      audio_marker = 0;
48
49      // purely for synchronization between ofApp::update and ofApp::audioOut
50      frame = 0;
51      samples_per_frame = sample_rate / FRAME_RATE;
52      assert( (sample_rate % FRAME_RATE) == 0 );
53      base_audio_index = 0;
54
55      // store our representation of a click sound (e.g. sine wave) so we dont always compute it
56      click_sound.assign( click_length + 1, 0. ); // click_length is some small number here ( << audio_size )
57      for(int i = 0; i < click_length; i++)
58          click_sound[i] = ...;               // maybe click is a small sine wave
59  }
60
```

Only for HW0!

Since "clicks" all different in HW1

update()

```
63  void ofApp::update(){
64
65      const float w = (float)ofGetWindowWidth();
66      const float h = (float)ofGetWindowHeight();
67
68      // purely for synchronization between ofApp::update and ofApp::audioOut
69      if( (frame % FRAME_RATE) == 0 ){
70          base_audio_index = 0;
71          std::fill(audio.begin(), audio.end(), 0.);
72      }
73
74      // iterate over all particles
75      //    (a) Add gravity force to velocity
76      //    (b) Filter *velocities* due to collisions (do not touch positions)
77      //    (c) Update positions based on new velocities
78      for(int i = 0; i < NUM_PARTICLES; i++){
79          particles[i].v.y += (float)9.8 * dt;            // add gravity force
80
81          bool collision = false;                         // did a collision happen for this particle? if so:
82          int collision_time;                             //    (a) when did it happen (within this timestep)
83          float collision_magnitude;                      //    (b) what impulse was applied to this particle
84
85          const ofVec3f& x_old = particles[i].x;          // old position
86          ofVec3f& v = particles[i].v;                    // velocity
87          const ofVec3f x_hat = x_old+v*dt;               // 'candidate' position
88          const float& e = particles[i].e;                // restitution coefficient
89
90          // if goes outside the right wall
91          if( (x_hat.x + PARTICLE_RADIUS > w) && (v.x > 0.) ){
92              collision = true;
93              collision_time = ...;                               // estimate time of collision
94              collision_magnitude=fabs((-e-(float)1.)*v.x);       // collision impulse magnitude
95              v.x *= -e;                                          // bounce back with restit. coeff.
96          }
```

update()

```
97          // if goes outside the left wall
98          else if((x_hat.x-PARTICLE_RADIUS < (float)0.) && (v.x < (float)0.)){
99              collision=true;
100             collision_time = ...;                           // estimate time of collision
101             collision_magnitude=fabs((-e-(float)1.)*v.x);;  // collision impulse magnitude
102             v.x *= -e;                                      // bounce back with restit. coeff.
103         }
104         // if goes outside the top wall
105         else if((x_hat.y+PARTICLE_RADIUS > h) && (v.y > (float)0.)){
106             collision=true;
107             collision_time = ...;                           // estimate time of collision
108             collision_magnitude=fabs((-e-(float)1.)*v.y);;  // collision impulse magnitude
109             v.y *= -e;                                      // bounce back with restit. coeff.
110         }
111         // if goes outside the bottom wall
112         else if((x_hat.y-PARTICLE_RADIUS < (float)0.) && (v.y < (float)0.)){
113             collision=true;
114             collision_time = ...;                           // estimate time of collision
115             collision_magnitude=fabs((-e-(float)1.)*v.y);;  // collision impulse magnitude
116             v.y *= -e;                                      // bounce back with restit. coeff.
117         }
118
119         // if a collision happens...
120         if(collision){
121             audio[ collision_time : click_length ] += collision_magnitude * click; // matlab-esque pseudo-code!
122         }
123
124         particles[i].x += particles[i].v * dt; // update x based on new velocity
125     }
126
127     frame++;                                // purely for synchronization between ofApp::update and ofApp::audioOut
128     base_audio_index += samples_per_frame;  // purely for synchronization between ofApp::update and ofApp::audioOut
129 }
130
```

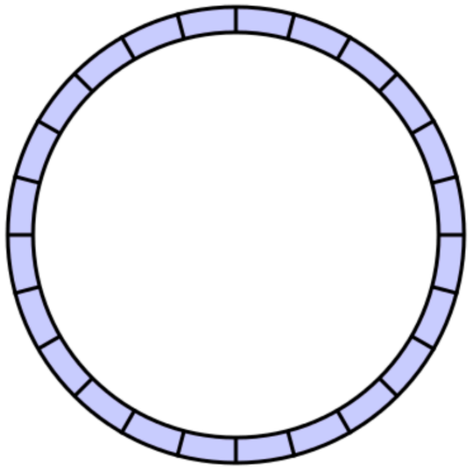
HW0 Pseudocode

```
132
133  void ofApp::draw(){
134      ofBackground(ofColor::black);
135      ofSetColor(ofColor::white);
136      ofFill();
137      for(int i=0;i<NUM_PARTICLES;i++){
138          draw_particle; // (sphere, circle, etc.)
139      }
140  }
141
142  //--------------------------------------------------------------
143
144  void ofApp::audioOut(float * output, int bufferSize, int nChannels){
145      for(int i = 0; i < bufferSize; i++){                // copy internal sound buffer to 'output'
146          output[i] = audio[audio_marker];
147          audio_marker = (audio_marker + 1) % audio_size; // our sound buffer is circular
148      }
149  }
150
151  //--------------------------------------------------------------
152
```

audioOut()

A useful data structure: Audio Ring Buffer

https://en.wikipedia.org/wiki/Circular_buffer



A ring showing, conceptually, a circular buffer. This visually shows that the buffer has no real end and it can loop around the buffer. However, since memory is never physically created as a ring, a linear representation is generally used as is done below.

Simulation/graphics thread writes audio to a ring buffer.

Questions:

- How large should the buffer be?
- How long can the audio clips written to the buffer be?

Potential Issues:

- What happens if graphics thread can't keep up writing?
- What happens if audio thread can't keep up reading?
- What happens if audio is written when audio is also being read?

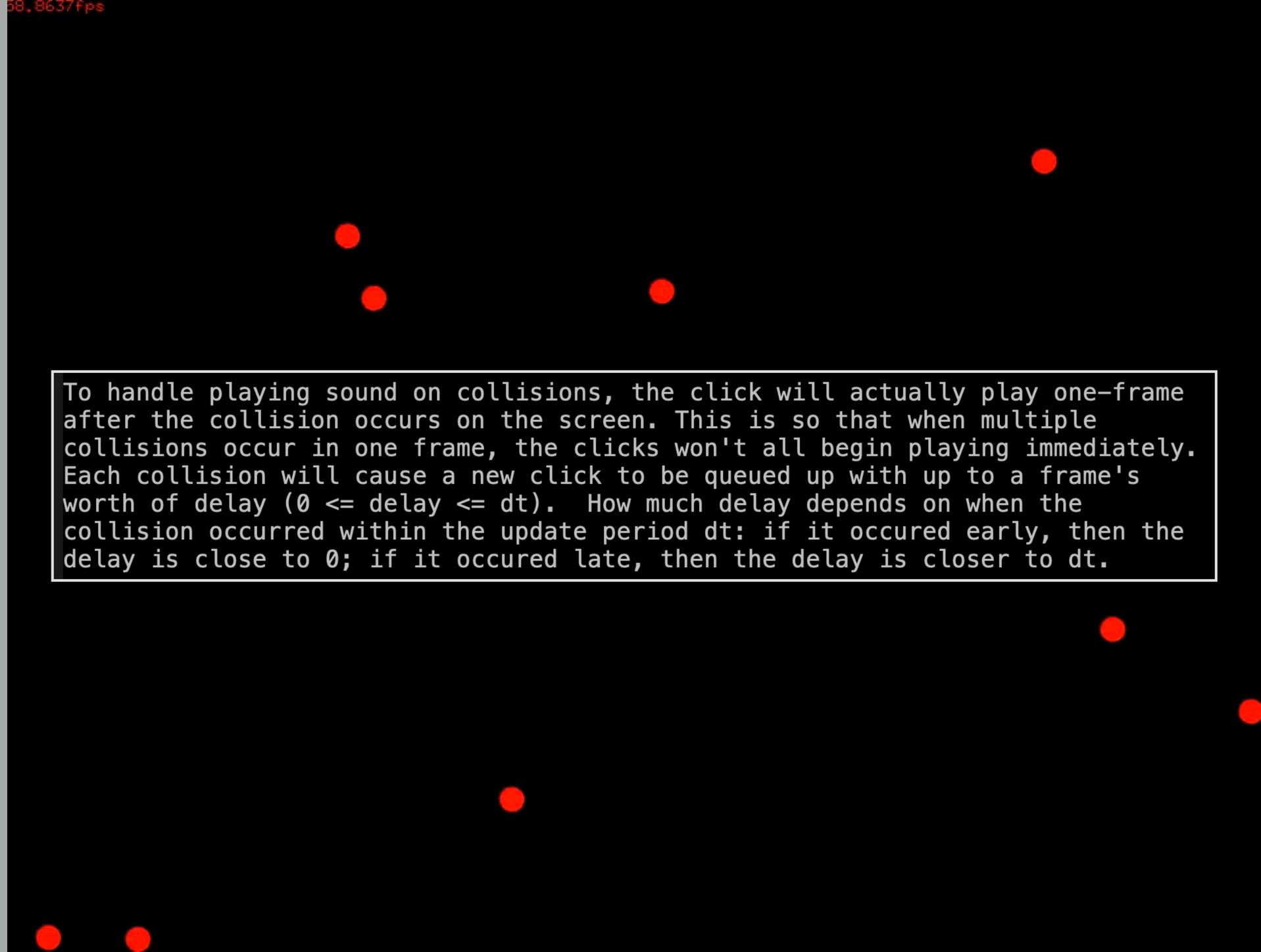
See code example: RingBuffer.h (not threadsafe!)

RingBuffer

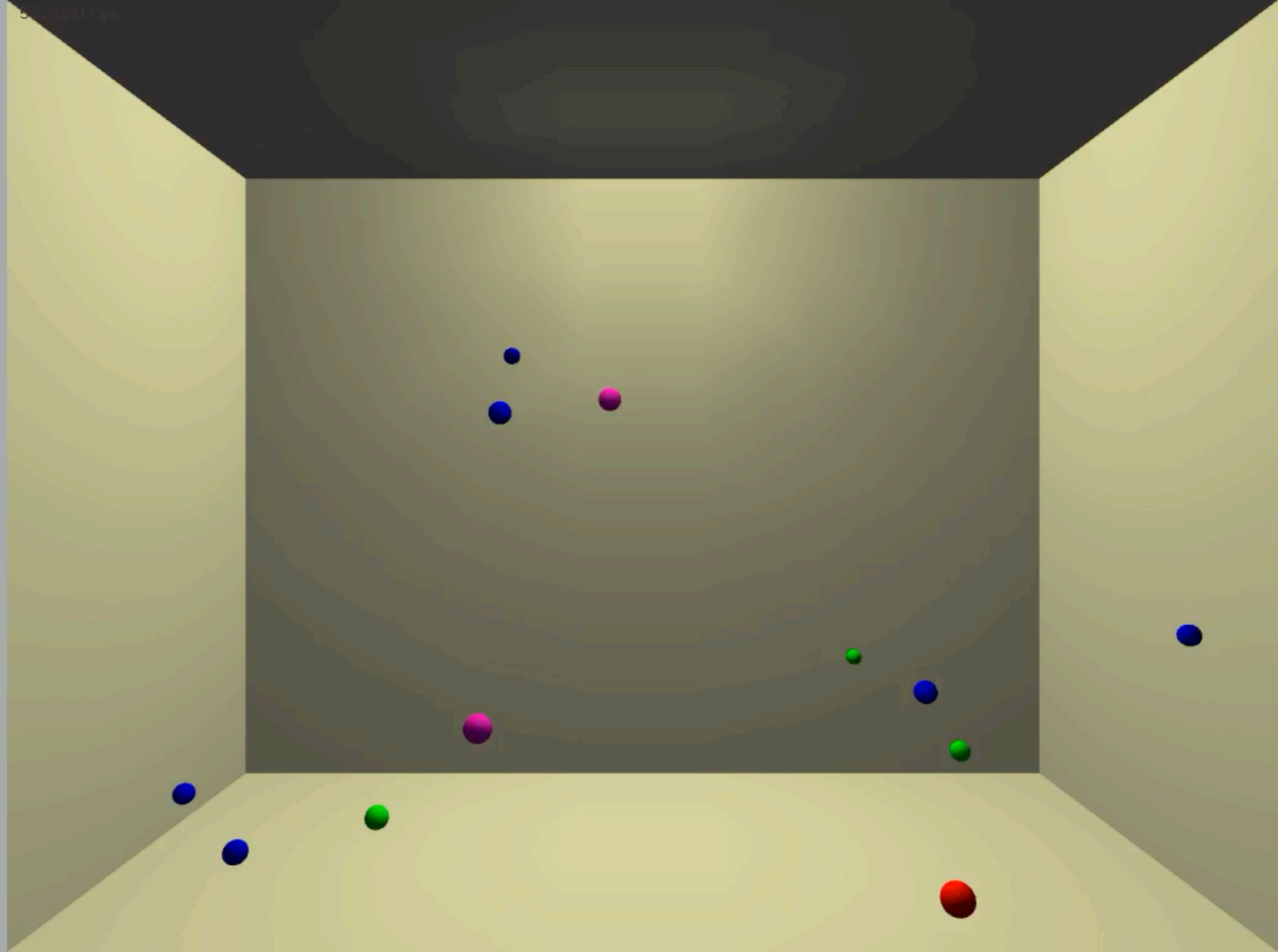
```cpp
#ifndef RINGBUFFER_H
#define RINGBUFFER_H

#include <stdlib.h>

template <class T, size_t N>
class RingBuffer {
public:
    class Iterator {…

public:
    RingBuffer() : rIndex(0), wIndex(0) {}

    size_t push(const T* buffer, size_t n) {
        n = std::min(n, capacityRemaining());
        size_t toEnd = N + 1 - wIndex;
        size_t n1 = std::min(toEnd, n);
        memcpy(&values[wIndex], buffer, n1 * sizeof(T));
        memcpy(&values[0], buffer + n1, (n - n1) * sizeof(T));
        wIndex = (wIndex + n) % (N + 1);
        return n;
    }
    size_t pushZeros(size_t n) {
        n = std::min(n, capacityRemaining());
        size_t toEnd = N + 1 - wIndex;
        size_t n1 = std::min(toEnd, n);
        memset(&values[wIndex], 0, n1 * sizeof(T));
        memset(&values[0], 0, (n - n1) * sizeof(T));
        wIndex = (wIndex + n) % (N + 1);
        return n;
    }
    size_t peek(T* buffer, size_t n) const {
        n = std::min(n, size());
        size_t toEnd = N + 1 - rIndex;
        size_t n1 = std::min(toEnd, n);
        memcpy(buffer, &values[rIndex], n1 * sizeof(T));
    size_t pop(size_t n) {
        n = std::min(n, size());
        rIndex = (rIndex + n) % (N + 1);
        return n;
    }
    size_t pop(T* buffer, size_t n) {
        n = std::min(n, size());
        size_t toEnd = N + 1 - rIndex;
        size_t n1 = std::min(toEnd, n);
        memcpy(buffer, &values[rIndex], n1 * sizeof(T));
        memcpy(buffer + n1, &values[0], (n - n1) * sizeof(T));
        rIndex = (rIndex + n) % (N + 1);
        return n;
    }

    size_t size() const {
        return (N + 1 + wIndex - rIndex) % (N + 1);
    }
    size_t capacityRemaining() const {
        return (N + rIndex - wIndex) % (N + 1);
    }

    Iterator begin() {
        return Iterator(&values[0], N, rIndex);
    }
    Iterator end() {
        return Iterator(&values[0], N, wIndex);
    }
    Iterator at(int i) {
        return Iterator(&values[0], N, (rIndex + i) % (N + 1));
    }
private:
    T values[N + 1];
    size_t rIndex;
    size_t wIndex;
};
```

Prior Student Results & Tips: Yixin Wang (F2015)

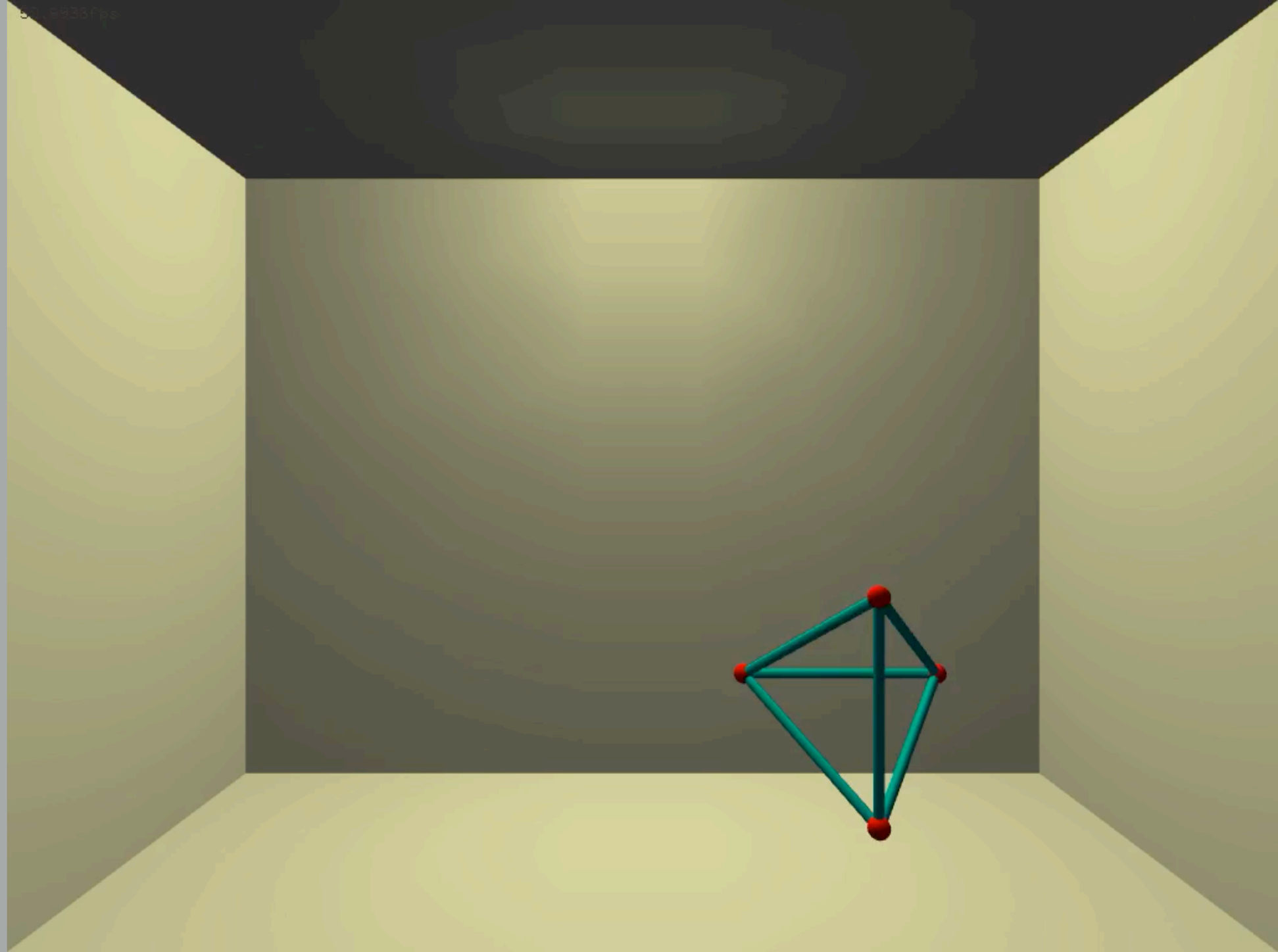
Prior Student Result: Yixin Wang (F2015) HW0



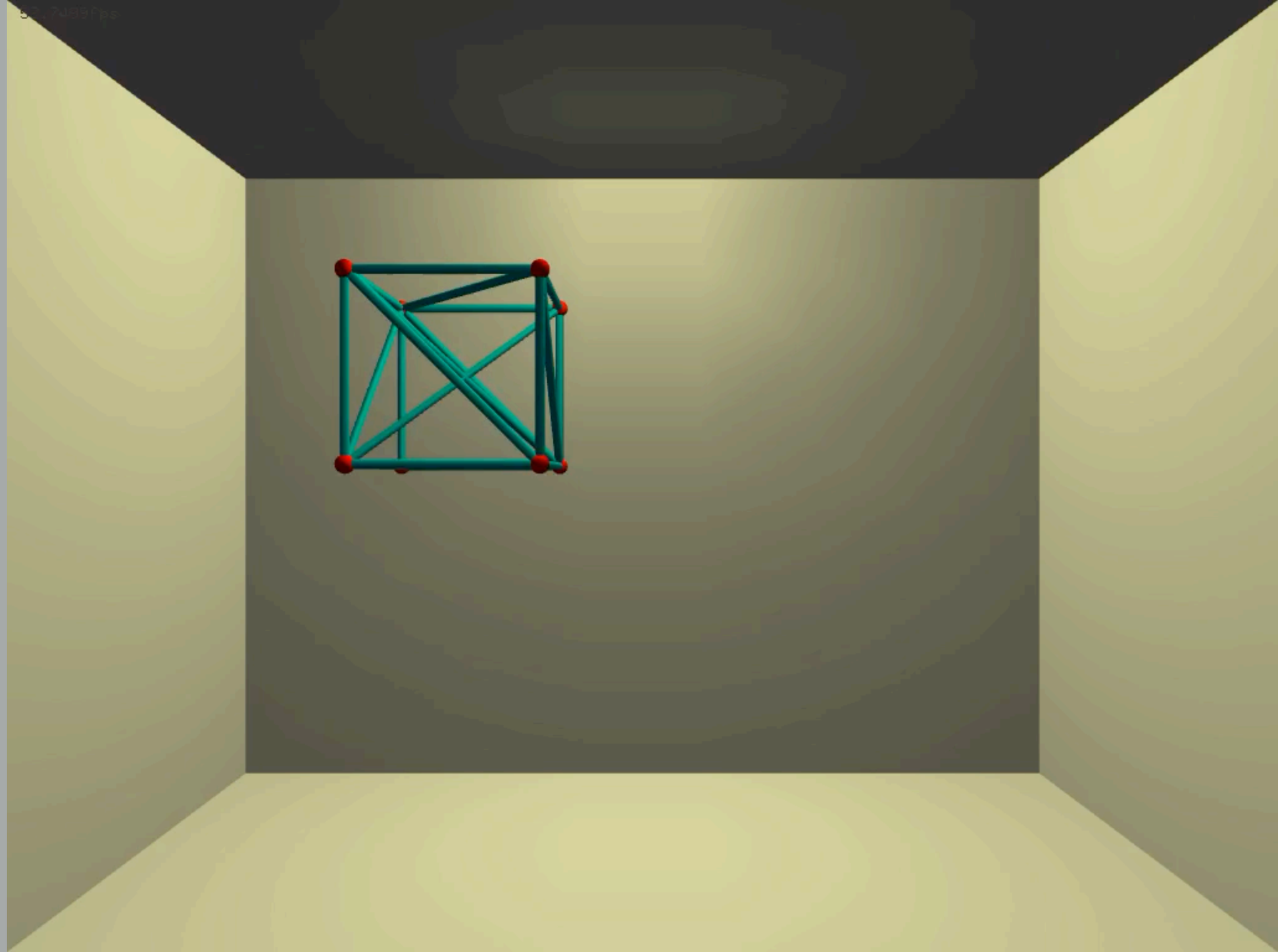
Prior Student Result:
Yixin Wang (F2015)
HW1



Prior Student Result:
Yixin Wang (F2015)
HW1



Prior Student Result:
Yixin Wang (F2015)
HW1



Prior Student Result:
Yixin Wang (F2015)
HW1

Findings

In my implementation for the previous assignment, my audio ring buffer stored info about the currently-playing audio instances instead of the audio samples themselves. Then, in the audioOut() function, the samples requested would be computed from the instances info. This did not require much computation since for each instance currently being played, I just had to find the right sample in the sound effect's WAV, scale it by its loudness factor, and accumulate that into the buffer. But for this assignment, each audio sample of each collision needs to be computed from a complicated formula, which means it was no longer feasible for the audio sample computation to occur in audioOut(). So the first modification I had to make was to change my audio ring buffer to store the computed audio samples instead of the metadata that can be used to compute them.

HW1 Code Snippets

Yixin Wang (2015)

ofApp.h

```
    RingBuffer<float, CHANNELS * AUDIO_SAMPLE_RATE> audioBuffer;
    std::mutex audioBufferLock;
```

ofApp.cpp

```
//--------------------------------------------------------------
void ofApp::audioOut(float* output, int bufferSize, int nChannels) {
    audioBufferLock.lock();
    int samplesPopped = audioBuffer.pop(output, bufferSize * CHANNELS);
    audioBufferLock.unlock();
    int samplesRemaining = CHANNELS * bufferSize - samplesPopped;
    if (samplesRemaining > 0) {
        memset(&output[samplesPopped], 0, samplesRemaining * sizeof(float));
    }
}
```

std::mutex

https://en.cppreference.com/w/cpp/thread/mutex

HW1 Code Snippets

Yixin Wang (2015)

ofApp.cpp: in update()

```
// process and add audio samples if any were produced this frame
if (audioEnd - audioStart > 0) {
    // add samples from the audio buffer to the audio ring buffer
    audioBufferLock.lock();
    Write tmp sound result from this update() to the buffer
    audioBufferLock.unlock();
}
```