Lamentations I: Remembering Clive Wearing
an opera

michael winter (mexico city, mx; 2017)

Acclaimed musician and musicologist Clive Wearing suffers from one of the worst known cases of amnesia after contracting herpes encephalitis. Wearing's caretakers encouraged him to keep a journal. *Lamentations I: Remembering Clive Wearing* is an opera that sets entries from Clive Wearing's journal. At intervals prescribed by the journal itself, entries are read and accompanied by a sonic and visual flickering / flourish of activity using recordings of Orlando de Lassus' 9 *Lamentationes Hieremiae Prophetae* (used because Wearing was an expert in the music of Lassus), sustained noises and tones, and lights. The piece intends to reverently reflect the importance of memory on our lives and personal identity.

Setting
The piece should be presented in a very dark, preferably pitch black space.

Selecting journal entries and timings of the waking states / flourishes
The central metaphor of the piece is the act of occasionally waking up into a brief state of consciousness from long periods of unconsciousness. The timings of these waking states are derived by the times that precede the entries in Wearing's journal.

In a given performance, at least 5 entries should be selected. It is preferred to select sets of successive entries with timings that occur within a range suitable for the length of the performance, which should last at least 30 minutes but could go on for much longer. In the journal, there are often long stints of times between entries. Therefore, multiple series of entries (even from different days / pages) can be selected. As explained in more detail below, any successive groupings of entries will be accompanied by a continuous drone. In the case that an interval between two entries is too long to be faithfully observed or a switch is made to another series of entries, the drone fades to silence and then fades back in to indicate the start of the next entry or series of entries. Provided below is an example sequence of times and entries.

0'00": (fade in drone) 3'00": (wake) 5:26pm

Visual elements
The visual elements of the piece consist of 18 lights that are controlled programmatically by a modified version of an algorithm called the rich-get-richer which is programmed in the included SuperCollider application. A more detailed explanation of the algorithm is detailed below however in short there are two sets of 9 lights each corresponding to one of the 9 Lamentations. The SuperCollider application generates live video that can be projected onto a wall. However, it is possible, or rather preferred, to use the signals to control real lights (e.g. with an Arduino) instead of, or in addition to, the projection. For real lights, incandescent bulbs with tungsten filaments should be used (or covers that produce a similar color). Further, with real lights, the setting can be designed to utilize the entire space by spreading the lights apart.

Sonic elements
Prelude: The piece may be preceded by—and / or accompanied by a program note with—a brief statement about memory that relates to Wearing's condition. These could be quotation by people who have written about memory or Wearing himself. [such as...] or something written especially for a given performance.

Readings of the journal entries: The entries should be read in a rather undramatic voice (almost solemnly). Wearing often marked out preceding entries and used quotation marks instead of reiterating the same words in successive entries. For each wake, the entire entry should be read starting with the time and including any words that have been marked out so long as they are legible. They should be read approximately 30 seconds into the waking state once the lights have reached a stable state (see below).

***Lamentationes Hieremiae Prophetae* (quinque vocum) of Orlando de Lassus**: Wearing was an expert in the music of Lassus. textitLamentationes Hieremiae Prophetae for five voices consists of 9 pieces each used as sources in the piece. A subsets of these sources are played during each wake up, filtered through a low pass filter, and attenuated according to a modified version of an algorithm called rich-get-richer (explained below).

Toy model of a 60 cycle hums: One of the sonic elements is a toy model of a 60 cycle hum. This generated by an interpolating oscillator reading (60 times per second) a buffer filled with random sample values between -1 and 1. The buffer is then passed through a low pass filter. 9 of these models are instantiated: one for each of the Lamentations.

Sustained tones:

Drone:

version generated: 2017.11.06

remembering_clive_wearing_main.scd

```
(
// MAIN LAUNCH
/*
Loads necessary files and definitions
When starting over, best to reboot server and intepreter

No GUI, just keycodes:
<ctrl + f> = enter full screen, <esc> = exit full screen
<ctrl + r> = run synth, <ctrl + p> = pause synth
<ctrl + s> = trigger start / fade in, <ctrl + e> = trigger end / fade out, <ctrl + t> = trigger wake

When automate == 1 (true), the trigger keys are disabled and the following lists in the SynthDef are used:
startTimes, wakeTimes, and endTimes

This launches the synth in a paused state to give the user time to put the visuals in full screen.
To start the synth, <ctrl + r> must be executed
*/
var automate = 1;
var dir = thisProcess.nowExecutingPath.dirname;
~fadeInTrigBus = Bus.control(s); ~wakeTrigBus = Bus.control(s); ~fadeOutTrigBus = Bus.control(s);
"remembering_clive_wearing_visuals.scd".loadRelative(true, {
        "remembering_clive_wearing_synthdef.scd".loadRelative(true, {
                var fileCount = 0, samples;
                PathName(dir +/+ "../audio").files.postln;
                samples = PathName(dir +/+ "../audio").files.sort({arg a, b;
                        a.fileName.toLower < b.fileName.toLower }).collect({|file|
                        Buffer.read(s, file.fullPath, action: {
                                if(fileCount == 8, {~flicker = \Synth.newPaused(\flicker,
                                        [\automate, automate, \bufs, samples,
                                                \fadeInTrigBusNum, ~fadeInTrigBus,
                                                \fadeOutTrigBusNum, ~fadeOutTrigBus,
                                                \wakeTrigBusNum, ~wakeTrigBus])}, {});
                                fileCount = fileCount + 1;
                        });
                });
        });
});
)
```


remembering_clive_wearing_main.scd

```
(
// SYNTHDEF
SynthDef(\flicker, { |automate = 1, bufs = #[0, 1, 2, 3, 4, 5, 6, 7, 8],
        fadeInTrigBusNum = 0, fadeOutTrigBusNum = 2, wakeTrigBusNum = 1|
        //~~~~~ Vars ~~~~~
        // start times / fade ins
        var fadeInTimes = [5, 100];
        // end times / fade outs (must be same length as fadeInTimes with fadeInTimes[i] < fadeOutTimes[i])
        var fadeOutTimes = [80, 150];
        // These are the times to wake up with a flourish of activity and the reading of an entry in the journal
        var wakeTimes = [15, 105];
        // These are the frequency ratios of the ensemble parts
        var freqRatios = [2, 3/2, 5/4, 7/4, 11/8, 13/8, 17/16, 19/16, 23/16];
        // Triggers
        var fadeInTrigs, wakeUpTrigs, fadeOutTrigs;
        // Rich-get-richer vars
        var pulse, state, runningPulseCount, norms, wealthGainLag, probs, selects;
        // Control signal vars
        var energy, switches, switchesSmoothed;
        // Sonification vars
        var lamentations, hums, ensemble, drone, fadeIn, fadeInOutEnv, wakeEnv, fadeOut;
        // Timed trigger
        var timedTrigger = {|times| Changed.kr(EnvGen.kr(Env.step({|i| i % 2} ! (times.size + 1),
                times.differentiate ++ [0.01]), Impulse.kr(0)))};

        //~~~~~ Triggers ~~~~~
        // Triggers for fadeInTimes
        fadeInTrigs = Select.kr(automate, [InTrig.kr(fadeInTrigBusNum), timedTrigger.value(fadeInTimes)]);
        // Triggers for fadeOutTimes
        fadeOutTrigs = Select.kr(automate, [InTrig.kr(fadeOutTrigBusNum), timedTrigger.value(fadeOutTimes)]);
        // Triggers for wakeTimes
        wakeUpTrigs = Select.kr(automate, [InTrig.kr(wakeTrigBusNum), timedTrigger.value(wakeTimes)]);

        //~~~~~ Rich-get-richer Algorithm ~~~~~
        // Update resolution
        pulse = Impulse.kr(30);
        // State of consciousness: asleep or awake; a wake up lasts 60 seconds plus a bit of a tail
        state =  EnvGen.kr(Env.sine(60), wakeUpTrigs);
        // Binary representation of which element was selected
        selects = LocalIn.kr(9);
        // Running sum of the times  each of the 9 elements has been selected over 120 pulses
        runningPulseCount = RunningSum.kr(pulse * selects * (state > 0), (ControlRate.ir / 30) * 120);
        // Normalize the counts over 121 pulses (adding a 1 in the wake states so probs is always > 1)
        norms = ((0.925 * (state > 0) + 0.075 + runningPulseCount) / 129);
        // Goes from 0 to 1 over several pulses after an element is chosen ending with 1 to 4 lights on favoring 2 or 3
        wealthGainLag = pow((PulseCount.kr(pulse, selects) /
                TWChoose.kr(wakeUpTrigs, [1, 2, 3, 4], [1, 2, 2, 1], 1)).clip, 4);
        // Probabilities such that the rich get richer except directly after an element is selected
        probs = {|i| pow(norms[i] * wealthGainLag[i], state * 4)} ! 9;
        // Select an element
```

```
        selects = TWindex.kr(pulse, probs, 1);
        // Feedback binary representation
        LocalOut.kr({|i| (i <= selects) * (selects <= i) } ! 9);

        //~~~~~ Control signals ~~~~~
        // The norms are basically the amount of energy to each element
        energy = norms;
        // In a toy model manner, this mimics voltage to the system
        switches = {|i| TRand.kr(0, 1, pulse) < energy[i]} ! 9;
        // Smooths the signal such that the lag time is greater as the element gets richer
        switchesSmoothed = {|i| Lag2.kr(switches[i], 0.25 + (0.75 * energy[i]))} ! 9;

        //~~~~~ Sonification ~~~~~
        // Playback of the Lassus Lamentations with a LPF as to not overwhelm to overall sound
        // Each of the Lamentations as a 2 in 3 chance of sounding
        lamentations = {|i| LPF.ar(PlayBuf.ar(2, bufs[i], 1, wakeUpTrigs,
                TIRand.kr(0, BufFrames.ir(bufs[i]), wakeUpTrigs), 1), 2880) *
            switchesSmoothed[i] * TWChoose.kr(wakeUpTrigs, [0, 1], [1, 2], 1)} ! 9;
        // Toy model of an electric hum
        hums = {|i| var buf = Array.fill(256, {1.0.rand2}).as(LocalBuf.clear);
                LPF.ar(Osc.ar(buf, 60, 0), 960) * switchesSmoothed[i]} ! 9;
        // Sustained tones based on energy to that element
        ensemble = {|i| SinOsc.ar(240 * freqRatios[i], 0) * energy[i]} ! 9;
        // Drone in sleep state that gets attenuated in wake state
        drone = {|i| var harm = pow(2, 2 - (i / 3).trunc), amp = (1 - (0.75 * energy.sum)) * (1 / pow(harm, 2));
                SinOsc.ar(60 * harm + TRand.kr(-3, 3, switches[i]), 0, amp)} ! 9;

        //~~~~~ Mix ~~~~~
        // Fade ins (10 secs) / outs (30 secs)
        fadeInOutEnv = EnvGen.kr(Env.asr(10, 1, 30, \sine),
                Latch.kr(Select.kr(fadeOutTrigs, [1, 0]), fadeInTrigs + fadeOutTrigs));
        // Fade wake sounds in and out based on total energy in the system
        wakeEnv = pow(0.8 * energy.sum + 0.2, 4);
        // Final mix currently set to output stereo where the multiplier is the final output level
        // Send to separate channel for more control of the indivuals sounds (e.g. with a mixer)
        // The lamentations of a 50 / 50 chance of sounding at all
        Out.ar([0,1], Mix.new(lamentations) * fadeInOutEnv * wakeEnv * 0.75 * TIRand.kr(0, 1, wakeUpTrigs));
        Out.ar([0,1], Mix.new(hums) * fadeInOutEnv * wakeEnv * 0.075);
        Out.ar([0,1], Mix.new(ensemble) * fadeInOutEnv * wakeEnv * 0.125);
        Out.ar([0,1], Mix.new(drone) * fadeInOutEnv * 0.25);

        //~~~~~ Visualization Control ~~~~~
        // Send signals to visualizer (these could be sent and scaled appropriated to control real lights)
        {|i| SendTrig.kr(pulse, i, switchesSmoothed[i] * fadeInOutEnv)} ! 9;
        {|i| SendTrig.kr(pulse, i + 9, energy[i] * fadeInOutEnv * wakeEnv)} ! 9;

        //~~~~~ Monitor Time ~~~~~
        Poll.kr(PulseDivider.kr(pulse, 30), PulseCount.kr(PulseDivider.kr(pulse, 30)), \time);
}).send(s);
)
```

remembering_clive_wearing_main.scd

```
(
// VISUALS
// Init vars and window
var projectionWin, scoreWin, osc_func, run = true, blend = {0} ! 27;
projectionWin = Window.new("Lamentations I: Remembering Clive Wearing", Rect(500, 500, 750, 500)).front;
projectionWin.background = Color.black;

// Keybinds (these can be change if conflicting with system keybinds)
projectionWin.view.keyDownAction = { |doc, char, mod, unicode, keycode, key|
        case
        // <ctrl + f> = enter full screen
        {mod == 262144 && key == 70} {projectionWin.fullScreen}
        // <esc> = exit full screen
        {mod == 0 && key == 16777216} {projectionWin.endFullScreen}
        // <ctrl + r> = run synth
        {mod == 262144 && key == 82} {~flicker.run}
        // <ctrl + p> = pause synth
        {mod == 262144 && key == 80} {~flicker.run(false)}
        // <ctrl + s> = start / fade in
        {mod == 262144 && key == 83} {~fadeInTrigBus.set(1)}
        // <ctrl + e> = end / fade out
        {mod == 262144 && key == 69} {~fadeOutTrigBus.set(1)}
        // <ctrl + t> = trigger wake
        {mod == 262144 && key == 84} {~wakeTrigBus.set(1)}
};

// Get control signals from SynthDef
osc_func = OSCFunc.new({arg msg, time; blend[msg[2]] = msg[3]},'/tr', s.addr);

// Draw the window
projectionWin.drawFunc = {
        var projectionRect = projectionWin.view.bounds;
        {|i| var outerLen, vPad, outerSquare;
                outerLen = projectionRect.width / 2;
                vPad = projectionRect.height - outerLen;
                outerSquare = projectionRect.insetBy(outerLen * i, vPad / 2).resizeTo(outerLen, outerLen);
                outerSquare = outerSquare.insetBy(outerLen * 0.05,
                        outerLen * 0.05).resizeTo(outerLen * 0.9, outerLen * 0.9);
                {|j| var innerLen, innerSquare;
                        innerLen = outerSquare.width / 3;
```

```
                        innerSquare = outerSquare.insetBy(innerLen * (j % 3),
                                innerLen * (j / 3).trunc).resizeTo(innerLen, innerLen);
                        Pen.addOval(innerSquare);
                        Pen.fillRadialGradient(innerSquare.center, innerSquare.center,
                                (innerSquare.width / 16), (innerSquare.width / 2),
                                Color.black.blend(Color.new255(255, 214, 170, 255), pow(blend[(i * 9) + j], 0.5)),
                                    Color.black);
        } ! 9 } ! 2
};
projectionWin.refresh;

// Refresh function
{ while { run } { projectionWin.refresh; 30.reciprocal.wait; } }.fork(AppClock);
)
```