Remembering Clive Wearing
an installation-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. *Remembering Clive Wearing* is an installation-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' nine *Lamentationes Hieremiae Prophetae* (as Wearing was an expert in the music of Lassus), sustained noises and tones, as well as 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. Perhaps as an installation.

Selecting journal entries and timings of the waking states / flourishes

The central metaphor of the piece is the act of occasionally waking up into brief states of consciousness from long periods of unconsciousness. The timings of these waking states are derived by the times in Wearing's journal.

In a given performance, at least five 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 thirty minutes but could go on for much longer (or indefinitely in the case of an installation). There are often long stints 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 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)		
0:03	(wake)	10:08 am	Now I am superlatively awake. First time awake for years. Patience Begins.
0:08	(wake)	10:13 am	Now I am overwhelmingly awake. First time for years.
0:16	(wake)	10:21 am	A gente liedown.
0:23	(wake)	10:28 am	Actually I am now first time awake for years. Patience observedly needed.
0:33	(wake)	10:38 am	Actually I am now awake for the first time for years. Patience begins so I may see everything. First thought: I adore Deborah forever.
0:43	(wake)	10:48 am	I am now totally, perfectly awake. First time for years. Patience begins fully observed.
0:46	(fade out drone)		
0:49	(fade in drone)		
0:52	(wake)	7:47 am	This illness has been like death. Till now. All senses work. First thought: I love darling Deborah forever. Our father.
0:56	(wake)	7:51 am	First conscious stroll
1:12	(wake)	8:07 am	I am totally, perfectly awake. First time. Breakfast
1:36	(wake)	8:31 am	Now I am really completely awake. First time. Patience.
1:40	(wake)	8:35 am	Time to see relaxing TV.
1:45	(fade out drone)		

Rich-get-richer algorithm

All parameters of the piece are controlled by an algorithm that models the sociological effect of the rich-get-richer. In short: the more an element is chosen, the more likely it is to be chosen. The exception is that immediately after an element is chosen, the probability that it is chosen over the next few trials is diminished. This allows, metaphorically speaking, multiple elements to accumulate wealth as opposed to only one garnering all the wealth. This algorithm works by making selections thirty times a second and keeping track of the number of times a given element (of nine in this particular case) is selected over a window of time. The count is then normalized and smoothed to control volume of the sonic elements and intensity of the visual elements. The algorithm was originally programmed in SuperCollider. The computer program, particularly the few lines of code that run the rich-get-richer algorithm, is thoroughly commented throughout with the intention that the code, in and of itself, documents and expresses the algorithm in complete detail.

Visual elements

The visual elements of the piece consist of eighteen lights that are controlled programmatically by the rich-get-richer algorithm. There are two sets of nine lights. In each set, each light corresponds to one of the nine Lamentations. One set is governed by impulses (smoothed over time) with frequencies based on the counts of the rich-get-richer algorithm. This creates a stochastic flickering effect. The other set is governed more directly by the counts themselves creating longer fades. While the SuperCollider application generates live video that can be projected onto a wall, 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. In the generated video (a screenshot of which is shown above), each set of nine lights is presented in a square matrix, the first set described above is on the left while the second is on the right. 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 a brief statement about memory that relates to Wearing's condition. This could also be in the form of an accompanying program note. For example, the statement / program note could include quotations by people who have written about memory or Wearing himself (such as Colin Blakemore, Oliver Sacks, or Wearing's wife at the time of the incident Deborah Wearing). A text may also be written especially for a given performance.

Drone: For a majority of the performance there is a drown which is only attenuated during the waking states and silences between sets of journal entries. The drone consists of nine sine tones corresponding to each of the nine Lamentations. The Lamentations are actually grouped in three sets of three: prima, segunda, and tertia. The three sine tones corresponding to the prima Lamentations are centered around 60 hertz, the three sine tones corresponding to the segunda Lamentations are centered around 240 hertz (two octaves above), and the three sine tones corresponding to the tertia Lamentations are centered around 960 hertz. During the non-waking states, impulses occasionally trigger a change in frequency (plus / minus 3 hertz from the center frequency) of the corresponding sine tone thus changing the overall beating pattern.

Readings of the journal entries: The entries should be read in a rather undramatic (almost solemn) voice. The text should be clearly audible with amplification. 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 relative stability. In the case of an installation, the readings may be prerecorded.

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

Toy model of a 60 cycle hum: One of the sonic elements is a toy model of a 60 cycle hum. This is 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. Nine of these models are instantiated: one corresponding to each of the Lamentations.

Sustained tones / ensemble: The lights also function as an animated score for the performers. During the drone, the performers follow the first set of nine lights (one for each performer) described in the section on the visual elements above. Occasionally, when their corresponding light flashes, they may play a long tone fading in and out from nothing or a very low volume (however, the general texture should remain sparse). For the waking states, the players follow the second set of lights (presented to the right in the generated video version) interpreting the brightness of the light as loudness of the tone. Note that this part is also synthesized / doubled with sine tones by the SuperCollider program. The pitches of the tones should correspond to octave equivalences of the frequencies programmed in the application which are based on prime harmonics of 60 hertz given as follows in terms of frequency ratio with respect to 60 hertz: 2, 3/2, 5/4, 7/4, 11/8, 13/8, 17/16, 19/16, and 23/16. Note that the tones, in prime order, correspond directly to the order of the Lamentations. That is, 2 corresponds to Lamentation 1.1, 3/2 corresponds to Lamentation 1.2, 5/4 corresponds to Lamentation 1.3, 7/4 corresponds to Lamentation 2.1, etc.

SuperCollider program

As mentioned in the description of the rich-get-richer algorithm, the code of the SuperCollider program intends to be a complete description of the processes that control all the elements of the piece. To date, the program does not have a graphical user interface and is controlled by keycodes given in the code. The wake and fade times can be automated or triggered manually. The signals that control the virtual lights may also be piped to an Arduino for use with real lights. Currently, the code would still need to be updated to playback recordings of the readings of the journal for an installation setting.

To launch the application, execute `remembering_clive_wearing_main.scd` in SuperCollider after booting the server (on Linux, this is achieved by pressing cmd+enter with the cursor anywhere within the code block). To run the synthesis processes, press cmd+r (note that this will start the piece if automation is on).

The source code for the application is appended at the end of this score and can be can be downloaded from a git repository at `https://www.github.com/mwinter80/remembering_clive_wearing`. Audio files of recordings of the Lassus Lamentations need to be placed in the `audio/` folder and named alphabetically / numerically such that the Lamentations get loaded in their proper order.

The generation of this document (using LaTex) contain version dates in order to help track changes and the git repository will also detail commit changes. The piece was written using SuperCollider version 3.8.0.

Arduino with AC dimmer circuit

The current version of the SuperCollider code includes code that will send messages to an Arduino controlling an AC dimmer circuit. This particular circuit will only work with incandescent bulbs, not LEDs. A schematic by Robert Twomey is provided below as well as the Arduino code. The circuit consists of one zero cross detector and up to 18 optoisolated light dimmers (the piece could be done with projection and 9 real lights, but, again, using 18 real lights is preferred). The schematic is built for 120 volts, but works with 240 volts if another 33k resistor is added between pin 2 of the H11AA1 optocoupler and AC neutral. There is also a small change that must be made in the Arduino code that is documented accordingly. Similarly, for 240 volts, bulbs with a higher voltage rating would need to be used. The circuit will also work with dimmable leds because the circuit converting the signal from an AC dimmer and control the led accordingly is contained within the bulb. Generally leds are not ideal because of the color of the light and the dimming mechanism is different, however there are dimmable leds available that look reasonably close to classic incandescent lights. Finally, there are minBrightness and maxBrightness variables in the SuperCollider code to limit and the range of values sent to the Arduino which can be adjusted for different types of lights.

version generated: 2018.05.21

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 ~arduino == 1 (true), the app assumes that an arduino is reachable, otherwise errors will be thrown

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 dir = thisProcess.nowExecutingPath.dirname;
var automate = 1;
~arduino = 1;
~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_synthdef.scd

```
(
// SYNTHDEF
SynthDef(\flicker, { |automate = 1, bufs = #[0, 1, 2, 3, 4, 5, 6, 7, 8],
    fadeInTrigBusNum = 0, fadeOutTrigBusNum = 2, wakeTrigBusNum = 1|
    //~~~~~ Vars ~~~~~
    // fadeInTimes, fadeOutTimes, wakeTimes according to example in score with 1 minute delay
    // start times / fade ins
    var fadeInTimes = [1, 50] * 60;
    // end times / fade outs (must be same length as fadeInTimes with fadeInTimes[i] < fadeOutTimes[i])
    var fadeOutTimes = [47, 106] * 60;
    // These are the times to wake up with a flourish of activity and the reading of an entry in the journal
    var wakeTimes = [4, 9, 17, 24, 34, 44, 53, 57, 73, 97, 101] * 60;
    // 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)))};
    // Monitor time
    var sTrig, sCount, secs, mins;

    //~~~~~ 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)
```

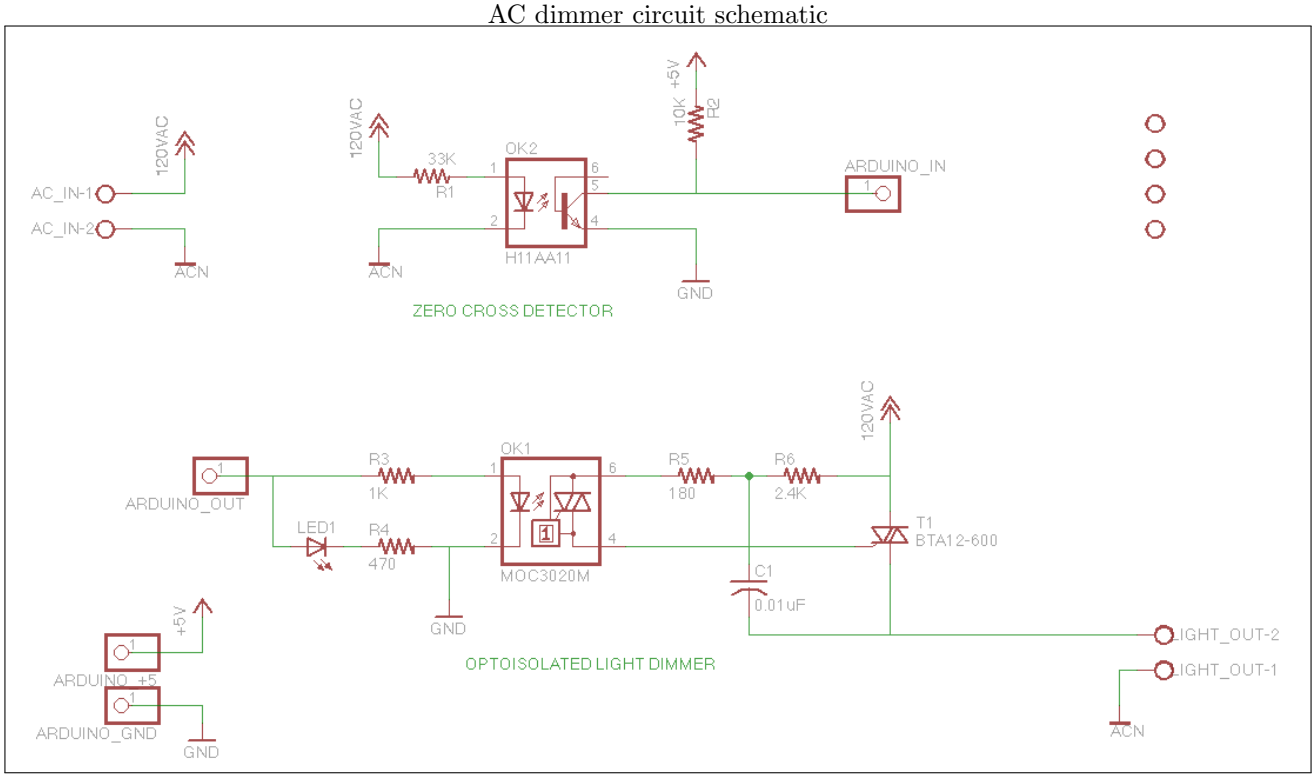
```
    norms = ((0.925 * (state > 0) + 0.075 + runningPulseCount) / 129);
    // Goes from 0 to 1 over several pulses after an element is chosen ending with 1 to 4 lights on favoring 2 or 3
    wealthGainLag = pow((PulseCount.kr(pulse, selects) /
        TWChoose.kr(wakeUpTrigs, [1, 2, 3, 4], [1, 2, 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 * 0.8 * 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 * 0.12);
    Out.ar([0,1], Mix.new(drone) * fadeInOutEnv * 0.2);

    //~~~~~ 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 ~~~~~
    sTrig = PulseDivider.kr(pulse, 30); sCount = PulseCount.kr(sTrig);
    secs = sCount % 60; mins = (sCount / 60).trunc;
    Poll.kr(sTrig, mins + (secs / 100), "time (min.secs)");
}).send(s);
)
```

remembering_clive_wearing_visuals.scd

```
(
// VISUALS
// Init vars and window
var projectionWin, arduino_port, osc_func, brightness = {0} ! 27, refresh_func;
projectionWin = Window.new("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} {refresh_func.fork(AppClock); ~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)}
};

// Connect arduino; edit first arg / port to match index or name of port: SerialPort.listDevices
if(~arduino == 1, { arduino_port = SerialPort( 0, 115200) }, {});

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

```
32
33  // Draw the window
34  projectionWin.drawFunc = {
35      var projectionRect = projectionWin.view.bounds;
36      {|i| var outerLen, vPad, outerSquare;
37          outerLen = projectionRect.width / 2;
38          vPad = projectionRect.height - outerLen;
39          outerSquare = projectionRect.insetBy(outerLen * i, vPad / 2).resizeTo(outerLen, outerLen);
40          outerSquare = outerSquare.insetBy(outerLen * 0.05,
41              outerLen * 0.05).resizeTo(outerLen * 0.9, outerLen * 0.9);
42          {|j| var innerLen, innerSquare;
43              innerLen = outerSquare.width / 3;
44              innerSquare = outerSquare.insetBy(innerLen * (j % 3),
45                  innerLen * (j / 3).trunc).resizeTo(innerLen, innerLen);
46              Pen.addOval(innerSquare);
47              Pen.fillRadialGradient(innerSquare.center, innerSquare.center,
48                  (innerSquare.width / 16), (innerSquare.width / 2),
49                  Color.black.blend(Color.new255(255, 214, 170, 255), pow(brightness[(i * 9) + j], 0.5)),
50                  Color.black);
51      } ! 9 } ! 2
52  };
53  projectionWin.refresh;
54
55  // Refresh function
56  refresh_func = { while { true } {
57      // updateProjection
58      projectionWin.refresh;
59
60      // updateArduino
61      if(~arduino == 1, {
62          // set min and max Brightness between 0 and 1 depending on wattage of light
63          // it is best to keep the min slightly higher than 0 to keep the light from turning completely off
64          {|i| var minBrightness = 0.15, maxBrightness = 0.85;
65              arduino_port.putAll(i.asString);
66              arduino_port.put(Char.space);
67              arduino_port.putAll((((brightness[i] * (maxBrightness - minBrightness) + minBrightness) - 1).abs * 256)
68                  .trunc.asString);
69              arduino_port.put(Char.space);
70              arduino_port.put(13);
71      } ! 18 }, {});
72
73      // delay
74      30.reciprocal.wait; } };
75  )
```

AC dimmer circuit schematic



remembering_clive_wearing_arduino.scd

```
1
2  #include <TimerOne.h> // Include Timer1 library
3  #include <SimpleMessageSystem.h> // Include SimpleMessageSystem library
4
5  // This is programmed for an Arduino Mega controlling lights from outs 22+
6  int AC_pin_offset = 22; // lowest out for lights
7  int dims[18]; // dimmer values
8  unsigned char clock_tick; // variable for Timer1
9
10 void setup() {
11   Serial.begin(115200);
12   for(int i = 0; i < 18; i++){
13     dims[i] = 256;
14     // Set the pins to the triacs as outputs (using outs 0 - 17)
15     pinMode(AC_pin_offset + i, OUTPUT);
16   };
17   attachInterrupt(0, zero_crosss_int, RISING);
18   // for resoution of 128 steps, set timer so 33 microseconds for 60 Hz or 39 for 50 Hz
19   Timer1.initialize(39);
20   Timer1.attachInterrupt(timerIsr); // attach the service routine here
21 }
22
23 // this is the dimmer
24 void timerIsr() {
25   clock_tick++;
26   for(int i = 0; i < 18; i++) {
27     if (dims[i] == clock_tick) {
28       digitalWrite(AC_pin_offset + i, HIGH); // triac on
29       }
30   };
31 }
32
33 // function to be fired at the zero crossing to dim the light
34 void zero_crosss_int() {
35   clock_tick=0;
36   for(int i = 0; i < 18; i++){
37     digitalWrite(AC_pin_offset + i, LOW); // triac Off
38   }
39 }
40
41 void loop(){
42   int light = 0;
43   int val = 0;
44   // Checks to see if the message is complete and erases any previous messages
45   if (messageBuild() > 0) {
46     light = messageGetInt();
47     val = messageGetInt();
48     dims[light] = val;
49   }
50 }
```