CellularAutomataMusic.java

```
 1 /*
 2  * Algorithmic Music Composition Software
 3  * @author Tom Donald Richmond
 4  * @version 2.0
 5  * @since 02/12/17
 6  */
 7
 8 import java.awt.BorderLayout;
 9 import java.awt.Color;
10 import java.awt.Dimension;
11 import java.awt.Graphics;
12 import java.awt.event.ActionEvent;
13 import java.awt.event.ActionListener;
14 import java.util.ConcurrentModificationException;
15
16 import javax.swing.JButton;
17 import javax.swing.JFrame;
18 import javax.swing.JPanel;
19 import javax.swing.Timer;
20 import javax.swing.JOptionPane;
21
22 import javax.sound.midi.*;
23
24 public class CellularAutomataMusic  extends JFrame{
25
26     private static final Color white = Color.WHITE, black = Color.BLACK;
27
28     private Board board;
29     private JButton start_pause, medieval, renaissance, baroque,
   classical, romantic, modern;
30     // variables to track total number of interval occurrences
31     int t;
32     // variables to track the occurrences of each interval for testing
33     int[] totals = new int[8];
34     // variable to hold string value representing era
35     String era;
36     // Boolean variable representing
37     Boolean analysis = false;
38
39     /*
40     * Creates blank board to feature automata, with start button to
41     * commence composition, as well as buttons to select epoch
```

```
42     * */
43     public CellularAutomataMusic(){
44
45         board = new Board();
46         board.setBackground(white);
47
48         /*
49         * Create buttons for start/stop
50         * */
51         start_pause = new JButton("Compose");
52         start_pause.addActionListener(board);
53
54         /*
55          * Create buttons for epoch selection
56          * */
57         medieval = new JButton("Medieval");
58         medieval.addActionListener(board);
59         renaissance = new JButton("Renaissance");
60         renaissance.addActionListener(board);
61         baroque = new JButton("Baroque");
62         baroque.addActionListener(board);
63         classical = new JButton("Classical");
64         classical.addActionListener(board);
65         romantic = new JButton("Romantic");
66         romantic.addActionListener(board);
67         modern = new JButton("Modern");
68         modern.addActionListener(board);
69
70         /*
71          * Subpanel for epoch selection
72          * */
73         JPanel subPanel = new JPanel();
74         subPanel.setLayout(new java.awt.GridLayout(6, 1));
75         subPanel.add(medieval);
76         subPanel.add(renaissance);
77         subPanel.add(baroque);
78         subPanel.add(classical);
79         subPanel.add(romantic);
80         subPanel.add(modern);
81
82         /*
83          * Add buttons to layout
```

```
84          * */
85         this.add(board, BorderLayout.CENTER);
86         this.add(start_pause, BorderLayout.SOUTH);
87         this.add(subPanel, BorderLayout.WEST);
88         //this.setLocationRelativeTo(null);
89 
90         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
91         this.pack();
92         this.setVisible(true);
93 
94     }
95 
96     public static void main(String args[]){
97         new CellularAutomataMusic();
98     }
99 
100    /*
101     * Board object featuring 4x15 Automata model, black and white values
102     * */
103    private class Board extends JPanel implements ActionListener{
104 
105        // Variables for board dimensions
106        private final Dimension DEFAULT_SIZE = new Dimension(15, 4);
107        private final int DEFAULT_CELL = 40, DEFAULT_INTERVAL = 100,
   DEFAULT_RATIO = 50;
108        private Dimension board_size;
109        private int cell_size, interval, fill_ratio;
110 
111        //boolean whether the composer is active
112        private boolean run;
113        // Timer for playing notes evenly
114        private Timer timer;
115        // variables to ensure the composer runs linearly
116        public int myOctave = 5, currentDiff = 0, range;
117        // variable to store the probability of each interval
118        double uni, step, third, fourth, fifth, sixth, seventh, octave;
119        // boolean to see if an epoch has been selected
120        boolean selected = false;
121        //grid to display automata-model
122        private Color[][] grid;
123 
124
```

```
125         /*
126          * Default constructor for Board object
127          */
128         public Board(){
129             board_size = DEFAULT_SIZE;
130             cell_size = DEFAULT_CELL;
131             interval = DEFAULT_INTERVAL;
132             fill_ratio = DEFAULT_RATIO;
133             run = false;
134 
135 
136             grid = new Color[board_size.height + 1][board_size.width + 1];
137             for (int h = 0; h < board_size.height; h++)
138                 for (int w = 0; w < board_size.width; w++){
139                     //int r = (int)(Math.random() * 100);
140                     //if (r >= fill_ratio)
141                     //grid[h][w] = black;
142                     //else grid[h][w] = white;
143                     grid[h][w] = white;
144                 }
145             timer = new Timer(interval, this);
146         }
147 
148         @Override
149         public Dimension getPreferredSize(){
150             return new Dimension(board_size.height * cell_size,
    board_size.width * cell_size);
151         }
152 
153         @Override
154         public void paintComponent(Graphics g){
155             super.paintComponent(g);
156             for (int h = 0; h < board_size.height; h++){
157                 for (int w = 0; w < board_size.width; w++){
158                     try{
159                         if (grid[h][w] == black)
160                             g.setColor(black);
161                         else if (grid[h][w] == white)
162                             g.setColor(white);
163                         g.fillRect(h * cell_size, w * cell_size,
    cell_size, cell_size);
164                     }
```

```
                    catch (ConcurrentModificationException cme){}
                }
            }
        }

        /*
         * Method to re-adjust the probability values when new epoch is selected
         * @param String representing epoch
         */
        public void changeEpoch(String epoch) {
            if(epoch=="medieval") {
                playNote(60);
                uni = 0.1484;
                step = 0.4998;
                third = 0.1178;
                fourth = 0.0371;
                fifth = 0.0234;
                sixth = 0.004;
                seventh = 0.0014;
                octave = 0.0057;
                range = 14;
                era = "Medieval";
            }
            else if(epoch=="renaissance") {
                playNote(62);
                uni = 0.2571;
                step = 0.4305;
                third = 0.1061;
                fourth = 0.0728;
                fifth = 0.048;
                sixth = 0.0048;
                seventh = 0.0006;
                octave = 0.0094;
                range = 22;
                era = "Renaissance";
            }
            else if(epoch=="baroque") {
                playNote(64);
                uni = 0.2623;
                step = 0.3558;
                third = 0.1114;
```

```
206                 fourth = 0.0728;
207                 fifth = 0.0442;
208                 sixth = 0.0292;
209                 seventh = 0.0108;
210                 octave = 0.0379;
211                 range = 23;
212                 era = "Baroque";
213             }
214             else if(epoch=="classical") {
215                 playNote(66);
216                 uni = 0.148;
217                 step = 0.3964;
218                 third = 0.1713;
219                 fourth = 0.0818;
220                 fifth = 0.0574;
221                 sixth = 0.0435;
222                 seventh = 0.0195;
223                 octave = 0.0353;
224                 range = 25;
225                 era = "Classical";
226             }
227             else if(epoch=="romantic") {
228                 playNote(68);
229                 uni = 0.207;
230                 step = 0.2791;
231                 third = 0.1112;
232                 fourth = 0.0649;
233                 fifth = 0.0416;
234                 sixth = 0.0282;
235                 seventh = 0.0123;
236                 octave = 0.0217;
237                 range = 30;
238                 era = "Romantic";
239             }
240             else if(epoch=="modern") {
241                 playNote(70);
242                 uni = 0.3086;
243                 step = 0.2153;
244                 third = 0.1011;
245                 fourth = 0.1053;
246                 fifth = 0.0723;
247                 sixth = 0.0591;
```

```
248                 seventh = 0.0364;
249                 octave = 0.0571;
250                 range = 37;
251                 era = "Modern";
252             }
253             else {
254                 System.out.println("Woah, how'd you manage that bud?");
255             }
256         }
257 
258         /*
259          * Method designed to generate a new musical note value based on
    given previous note value
260          * @param int prevVal
261          * @returns int newVal
262          * */
263         public int ruleGenerator(int prevVal){
264             if (prevVal == 0){
265                 return 1;
266             }
267 
268             /* Sets ascLim and descLim to half of the average range of the
269              * given epoch. DescLim gets the ceiling arbitrarily*/
270             int ascLim = range/2;
271             int descLim= (range/2) + (range%2);
272 
273             double running = 0.0;
274             double value = Math.random();
275 
276             int newVal;
277             int diff = 0;
278             int direction = (int)(Math.random()*2);
279 
280             /* determines before each note whether it was generated to be
    ascending
281              * or descending. This process is regulated with ascLim and
    descLim */
282             boolean ascending = false;
283             if(direction == 1)
284                 ascending = true;
285 
286             /* Resets the valFound var to false for next note generation
```

```
   */
287             boolean valFound = false;
288
289             /* checks which range the generated number falls in and
   produces a
290              * note based on this value. Once note is found, valFound is
   set to
291              * true, and no other if statements are reached. It will
   access each
292              * if statement until the correct is found, increasing running
   total
293              * as it goes. */
294             if (value <= uni){
295                 totals[0]+=1;
296                 t+=1;
297                 diff = 0;
298                 valFound = true;
299                 System.out.println("Unison");
300             }
301             running += uni;
302             if ((value <= step + running) && valFound == false){
303                 totals[1]+=1;
304                 t+=1;
305                 diff =  1;
306                 valFound = true;
307                 System.out.println("Step");
308             }
309             running += step;
310             if (value <= third + running && valFound == false){
311                 totals[2]+=1;
312                 t+=1;
313                 diff =  2;
314                 valFound = true;
315                 System.out.println("Third");
316             }
317             running += third;
318             if (value <= fourth + running && valFound == false){
319                 totals[3]+=1;
320                 t+=1;
321                 diff =  3;
322                 valFound = true;
323                 System.out.println("Forth");
```

```
            }
            running += fourth;
            if (value <= fifth + running && valFound == false){
                totals[4]+=1;
                t+=1;
                diff =  4;
                valFound = true;
                System.out.println("Fifth");
            }
            running += fifth;
            if (value <= sixth + running && valFound == false){
                totals[5]+=1;
                t+=1;
                diff =  5;
                valFound = true;
                System.out.println("Sixth");
            }
            running += sixth;
            if (value <= seventh + running && valFound == false){
                totals[6]+=1;
                t+=1;
                diff =  6;
                valFound = true;
                System.out.println("Seventh");
            }
            running += seventh;
            if (value <= octave + running && valFound == false){
                totals[7]+=1;
                t+=1;
                diff =  7;
                valFound = true;
                System.out.println("Octave");
            }

            //System.out.println((currentDiff+diff) +": total diff");
            if (ascending && currentDiff + diff >= ascLim) {
                System.out.println("Switched, too high");
                ascending = false;
            }
            if (!ascending && -1*(currentDiff - diff) >= descLim) {
                System.out.println("Switched, too low");
                ascending = true;
```

```
366             }
367             System.out.println("Ascending = "+ascending);
368             if(ascending){
369                 currentDiff += diff;
370                 System.out.println(currentDiff);
371                 newVal = prevVal;
372                 for (int i = 0; i < diff; i++){
373                     if (newVal == 5 || newVal == 12)
374                         newVal += 1;
375                     else
376                         newVal += 2;
377                     if (newVal > 12) {
378                         myOctave++;
379                         newVal -= 12;
380                     }
381                 }
382             }
383             else{
384                 currentDiff -= diff;
385                 System.out.println(currentDiff);
386                 newVal = prevVal;
387                 for (int i = 0; i < diff; i++){
388                     if (newVal == 6 || newVal == 13 || newVal == 1)
389                         newVal -= 1;
390                     else
391                         newVal -= 2;
392                     if (newVal < 1) {
393                         newVal += 12;
394                         myOctave--;
395                     }
396                 }
397             }
398             System.out.println(newVal + " " + ascending);
399             int noteVal = toNote(newVal, ascending);
400 
401             //System.out.println(prevVal);
402             //newVal = 1+((int)(Math.random()*12));
403             return noteVal;
404         }
405 
406         /*
407          * Method designed to generate a new musical note value based on
```

```
   given previous note value
408          * @param int prevVal
409          * @returns int newVal
410          * */
411         public void ruleGeneratorAnalysis(){
412 
413             double running = 0.0;
414             double value = Math.random();
415 
416             /* Resets the valFound var to false for next note generation
   */
417             boolean valFound = false;
418 
419             /* checks which range the generated number falls in and
   produces a
420              * note based on this value. Once note is found, valFound is
   set to
421              * true, and no other if statements are reached. It will
   access each
422              * if statement until the correct is found, increasing running
   total
423              * as it goes. */
424             if (value <= uni){
425                 totals[0]+=1;
426                 t+=1;
427                 valFound = true;
428             }
429             running += uni;
430             if ((value <= step + running) && valFound == false){
431                 totals[1]+=1;
432                 t+=1;
433                 valFound = true;
434             }
435             running += step;
436             if (value <= third + running && valFound == false){
437                 totals[2]+=1;
438                 t+=1;
439                 valFound = true;
440             }
441             running += third;
442             if (value <= fourth + running && valFound == false){
443                 totals[3]+=1;
```

```
444                 t+=1;
445                 valFound = true;
446             }
447             running += fourth;
448             if (value <= fifth + running && valFound == false){
449                 totals[4]+=1;
450                 t+=1;
451                 valFound = true;
452             }
453             running += fifth;
454             if (value <= sixth + running && valFound == false){
455                 totals[5]+=1;
456                 t+=1;
457                 valFound = true;
458             }
459             running += sixth;
460             if (value <= seventh + running && valFound == false){
461                 totals[6]+=1;
462                 t+=1;
463                 valFound = true;
464             }
465             running += seventh;
466             if (value <= octave + running && valFound == false){
467                 totals[7]+=1;
468                 t+=1;
469                 valFound = true;
470             }
471 
472             /* When the composer has generated 100 notes,
473              * it automatically calculates the results and prints
474              * for analysis process */
475             if(t==100) {
476                 System.out.println(kernResults());
477                 //JOptionPane.showMessageDialog(null,kernResults());
478                 clearStats();
479             }
480         }
481 
482         /*
483          * Method that takes note value representation from binary as
    integer, prints corresponding
484          * value and plays note using MIDI output
```

```
485          * @param int val - Value of note (1-13) generated by the rule system
486          * @returns String letter value equivelant to corresponding int value
487          * */
488         public int toNote(int val, Boolean asc){
489             int noteVal;
490             int C = myOctave * 12;
491 
492             if(val == 1 || val == 13){
493                 noteVal = C+0;
494                 System.out.println("C");
495             }
496             else if(val == 2){
497                 noteVal = C+1;
498                 System.out.println("C#/D-");
499             }
500             else if(val == 3){
501                 noteVal = C+2;
502                 System.out.println("D");
503             }
504             else if(val == 4){
505                 noteVal = C+3;
506                 System.out.println("D#/E-");
507             }
508             else if(val == 5){
509                 noteVal = C+4;
510                 System.out.println("E");
511             }
512             else if(val == 6){
513                 noteVal = C+5;
514                 System.out.println("F");
515             }
516             else if(val == 7){
517                 noteVal = C+6;
518                 System.out.println("F#/G-");
519             }
520             else if(val == 8){
521                 noteVal = C+7;
522                 System.out.println("G");
523             }
524             else if(val == 9){
```

```
525                 noteVal = C+8;
526                 System.out.println("G#/A-");
527             }
528             else if(val == 10){
529                 noteVal = C+9;
530                 System.out.println("A");
531             }
532             else if(val == 11){
533                 noteVal = C+10;
534                 System.out.println("A#/B-");
535             }
536             else if(val == 12){
537                 noteVal = C+11;
538                 System.out.println("B");
539             }
540             else {
541                 return 0;
542             }
543             //System.out.println(noteVal);
544             playNote(noteVal);
545             return val;
546         }
547 
548         /*
549          * (non-Javadoc)
550          * Action Listener for all buttons, compose, terminate, medieval,
551          * renaissance, baroque, classical, romantic and modern.
552          * @see
    java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
553          */
554         public void actionPerformed(ActionEvent e) {
555 
556             //reads binary value of last sequence
557             int a = 0, b = 0, c = 0, d = 0, val = 0;
558 
559             //counts binary from board for conversion to decimal
560             if (grid[0][board_size.width-1]  == black)
561                 a = 1;
562             if (grid[1][board_size.width-1]  == black)
563                 b = 1;
564             if (grid[2][board_size.width-1]  == black)
565                 c = 1;
```

```
566             if (grid[3][board_size.width-1]  == black)
567                 d = 1;
568 
569             //converts binary sequence into decimal with variable val
570             if(a==1)
571                 val+=8;
572             if(b==1)
573                 val+=4;
574             if(c==1)
575                 val+=2;
576             if(d==1)
577                 val+=1;
578 
579             //shifts bottom n-1 sequences up to make room for next
    sequence
580             for (int h = 0; h < board_size.height; h++){
581                 for (int w = 0; w < board_size.width-1; w++){
582                     grid[h][w] = grid[h][w+1];
583                 }
584             }
585 
586             //repaints the bottom line sequence based on rule
587             if (e.getSource().equals(timer) && analysis == false){
588                 int newNote = ruleGenerator(val);
589 
590                 if (newNote >= 8){
591                     grid[0][board_size.width-1] = black;
592                     newNote = newNote-8;
593                 }
594                 else
595                     grid[0][board_size.width-1] = white;
596                 if (newNote >= 4){
597                     grid[1][board_size.width-1] = black;
598                     newNote = newNote-4;
599                 }
600                 else
601                     grid[1][board_size.width-1] = white;
602                 if (newNote >= 2){
603                     grid[2][board_size.width-1] = black;
604                     newNote = newNote-2;
605                 }
606                 else
```

```
607                     grid[2][board_size.width-1] = white;
608                 if (newNote >= 1){
609                     grid[3][board_size.width-1] = black;
610                     newNote = newNote-1;
611                 }
612                 else
613                     grid[3][board_size.width-1] = white;
614                 repaint();
615                 Color[][] newGrid = new Color[board_size.height]
    [board_size.width];
616             }
617 
618             //repaints the bottom line sequence based on rule
619             if (e.getSource().equals(timer) && analysis == true){
620                 ruleGeneratorAnalysis();
621             }
622 
623             //Start-Pause button processing
624             else if(e.getSource().equals(start_pause)){
625                 if(run){
626                     timer.stop();
627                     //JOptionPane.showMessageDialog(null,printResults());
628                     JOptionPane.showMessageDialog(null,printResults());
629                     start_pause.setText("Compose");
630                 }
631                 else {
632                     if (selected) {
633                         timer.restart();
634                         start_pause.setText("Terminate");
635                 }
636                 else {
637                         JOptionPane.showMessageDialog(null, "Must first
    select an epoch from which to compose");
638                         run = !run;
639                 }
640             }
641             run = !run;
642             }
643 
644             //Medieval button processing
645             else if(e.getSource().equals(medieval)){
646                 medieval.setEnabled(false);
```

```
647                 renaissance.setEnabled(true);
648                 baroque.setEnabled(true);
649                 classical.setEnabled(true);
650                 romantic.setEnabled(true);
651                 modern.setEnabled(true);
652                 changeEpoch("medieval");
653                 selected = true;
654             }
655             //Renaissance button processing
656             else if(e.getSource().equals(renaissance)){
657                 medieval.setEnabled(true);
658                 renaissance.setEnabled(false);
659                 baroque.setEnabled(true);
660                 classical.setEnabled(true);
661                 romantic.setEnabled(true);
662                 modern.setEnabled(true);
663                 changeEpoch("renaissance");
664                 selected = true;
665             }
666             //Baroque button processing
667             else if(e.getSource().equals(baroque)){
668                 medieval.setEnabled(true);
669                 renaissance.setEnabled(true);
670                 baroque.setEnabled(false);
671                 classical.setEnabled(true);
672                 romantic.setEnabled(true);
673                 modern.setEnabled(true);
674                 changeEpoch("baroque");
675                 selected = true;
676             }
677             //Classical button processing
678             else if(e.getSource().equals(classical)){
679                 medieval.setEnabled(true);
680                 renaissance.setEnabled(true);
681                 baroque.setEnabled(true);
682                 classical.setEnabled(false);
683                 romantic.setEnabled(true);
684                 modern.setEnabled(true);
685                 changeEpoch("classical");
686                 selected = true;
687             }
688             //Romantic button processing
```

```
689             else if(e.getSource().equals(romantic)){
690                 medieval.setEnabled(true);
691                 renaissance.setEnabled(true);
692                 baroque.setEnabled(true);
693                 classical.setEnabled(true);
694                 romantic.setEnabled(false);
695                 modern.setEnabled(true);
696                 changeEpoch("romantic");
697                 selected = true;
698             }
699             //Modern button processing
700             else if(e.getSource().equals(modern)){
701                 medieval.setEnabled(true);
702                 renaissance.setEnabled(true);
703                 baroque.setEnabled(true);
704                 classical.setEnabled(true);
705                 romantic.setEnabled(true);
706                 modern.setEnabled(false);
707                 changeEpoch("modern");
708                 selected = true;
709             }
710         }
711     }
712 
713     /*
714      * Method to play note value using MIDI synthesizer based upon input
    note
715      * @param int representing the MIDI value of desired note.
716      */
717     public void playNote(int i) {
718         try{
719         /* Create a new Synthesizer and open it.
720          */
721             Synthesizer midiSynth = MidiSystem.getSynthesizer();
722             midiSynth.open();
723 
724             //get and load default instrument and channel lists
725             Instrument[] instr =
    midiSynth.getDefaultSoundbank().getInstruments();
726             MidiChannel[] mChannels = midiSynth.getChannels();
727 
728             midiSynth.loadInstrument(instr[0]);//load an instrument
```

```
729             mChannels[0].noteOff(i);//turn off the previous note
730             mChannels[0].noteOn(i, 120);//On channel 0, play note number i
    with velocity 120
731             try {
732                 //Following line controls duration of notes played. 1000
    used for samples of 30 seconds. 750 used for samples of 15 seconds
733                 Thread.sleep(750); // wait time in milliseconds to control
    duration
734             }
735             catch( InterruptedException e ) {}
736         }
737         catch (MidiUnavailableException e) {}
738     }
739 
740     /*
741      * method that returns string that prints composition statistics for
    visual analysis
742      * @returns String statistics
743      */
744     public String printResults() {
745         return "Total length of composition: "+t+"\n"
746                 +"\tStatistics:\n"
747                 +"\nUnison:\t "+((double)totals[0]/t)
748                 +"\nStep:\t "+((double)totals[1]/t)
749                 +"\nThird:\t "+((double)totals[2]/t)
750                 +"\nForth:\t "+((double)totals[3]/t)
751                 +"\nFifth:\t "+((double)totals[4]/t)
752                 +"\nSixth:\t "+((double)totals[5]/t)
753                 +"\nSeventh:\t "+((double)totals[6]/t)
754                 +"\nOctave:\t "+((double)totals[7]/t);
755     }
756 
757     /*
758      * method that returns string that prints composition statistics for
    analysis
759      * @returns String statistics
760      */
761     public String kernResults() {
762         //variable to store percentage of most common interval
763         int max = 0;
764 
765         // computes the most common interval
```

```
766         for(int i = 0; i<8;i++) {
767             if(totals[i] > max){
768                 max = totals[i];
769             }
770         }
771 
772         //returns expected String output based on totals array and above
    computation
773         return ""+((double)totals[0]/t)
774                 +","+((double)totals[1]/t)
775                 +","+((double)totals[2]/t)
776                 +","+((double)totals[3]/t)
777                 +","+((double)totals[4]/t)
778                 +","+((double)totals[5]/t)
779                 +","+((double)totals[6]/t)
780                 +","+((double)totals[7]/t)
781                 +","+((double)max/t)
782                 +","+era;
783     }
784 
785     /*
786      * Method to clear the statistics after terminations for next
    composition
787      */
788     public void clearStats() {
789         //loops through all saved data and resets to 0 for future
    processing
790         for (int i = 0; i < 8; i++) {
791             totals[i] = 0;
792         }
793         t = 0;
794     }
795 }
```