Dokumentation - Entwicklung eines Audio-Plugins mit JUCE

C++/JUCE 3

Idee und Einleitung

Im Rahmen dieses Projekts wird eine Audioanwendung entwickelt, die in einem Host oder einer *Digital Audio Workstation* (DAW) verwendet werden kann, um Audiosignale zu bearbeiten. Die Idee für das in diesem Projekt entstandene Audio-Plugin ist das Hervorbringen von *Glitches* oder gewollten Fehlern im Audiosignal.

Das Plugin orientiert sich grob an einem bereits existierenden Plugin von dem Hersteller *Glitchmachines* mit dem Namen *Fracture.*[1] In der Dokumentation des Plugins wird *Fracture* wie folgt beschrieben: “Fracture is a buffer effect plugin geared toward creating robotic artifacts and abstract musical malfunctions.[...] This plugin is especially useful when it comes to adding glitchy articulations and abstract textures to your projects”.[2] *Fracture* zielt also darauf ab, den Eindruck von Artefakten und musikalischen Fehlfunktionen im Audiosignal zu vermitteln. Dies wird beispielsweise mit einer Gruppierung von Funktionen umgesetzt, die in der Dokumentation von *Fracture* unter der Bezeichnung *Buffer Module* zusammengefasst werden. Die Funktion des *Buffer Modules* besteht darin, einen kleinen Teil des Signals aufzunehmen und diesen daraufhin so oft zu wiederholen, wie vom Benutzer angegeben. Die Geschwindigkeit der wiedergegebenen Signale kann ebenfalls angepasst werden. Die Parameter dieses Moduls können über einen LFO, dessen Frequenz und Wellenform ebenfalls angepasst werden kann, automatisch moduliert werden. Im weiteren Verlauf wird die Gruppe dieser Funktionen *Stutter Engine* genannt.

Das in diesem Projekt entstandene *GlitchPlugin* hat ähnliche Funktionen, wie das *Buffer Module* aus dem Plugin *Fracture.* Ziel ist die Entwicklung eines Bausteins zur Umsetzung einer “Glitch-Ästhetik”. In dem folgenden Kapitel wird auf die Umsetzung der einzelnen Funktionen des Plugins eingegangen. Im letzten Kapitel werden die umgesetzten Möglichkeiten des entwickelten Plugins zusammengefasst und Erweiterungsmöglichkeiten vorgestellt.

Umsetzung

Das Projekt baut auf die von JUCE bereitgestellten Klassen *PluginProcessor* und *PluginEditor* auf. Über die Klasse *PluginProcessor* können Operationen auf dem *Audio*

[1] Fracture – Glitchmachines
[2] FRACTURE_User_Guide

Thread auf Sample-Ebene durchgeführt werden. Die Klasse *PluginEditor* ermöglicht das Zusammenstellen einer Benutzeroberfläche und das Einlesen von Benutzereingaben über Elemente wie *Slider* oder *Buttons* und die Verwendung dieser Eingaben als Parameter in der Signalverarbeitung. Diese Klassen verwenden weitere Klassen, die beispielsweise die Logik der *StutterEngine* umsetzen. Dazu gehört die Klasse *StutterBuffer*. Sie realisiert das Aufnehmen eines kleinen Teils des Audiosignals, sowie die Logik zum wiederholten Wiedergeben und enthält die Funktion, linear zwischen Samples zu interpolieren, um die Abspielgeschwindigkeit des Signals zu bestimmen. Neben dem *StutterBuffer* verwendet der *PluginProcessor* die Klasse *LFO*. Diese dient dazu, eine periodische Wellenform mit niedriger Frequenz zu erzeugen. Dessen Werte können vom *PluginProcessor* abgefragt und eingesetzt werden, um Parameter im *StutterBuffer* zu steuern.

Die Parameter des *StutterBuffers* und des *LFOs* können ebenfalls über die Benutzeroberfläche und entsprechenden Reglern gesteuert werden, die über den *PluginEditor* und den *LFOVisualizer* zu Verfügung gestellt werden.

In Abbildung 1 sind die groben Zusammenhänge zwischen den einzelnen Klassen dargestellt. In den folgenden Kapiteln werden die Hauptfunktionen der einzelnen Klassen, begleitet von Code-Beispielen, vorgestellt.

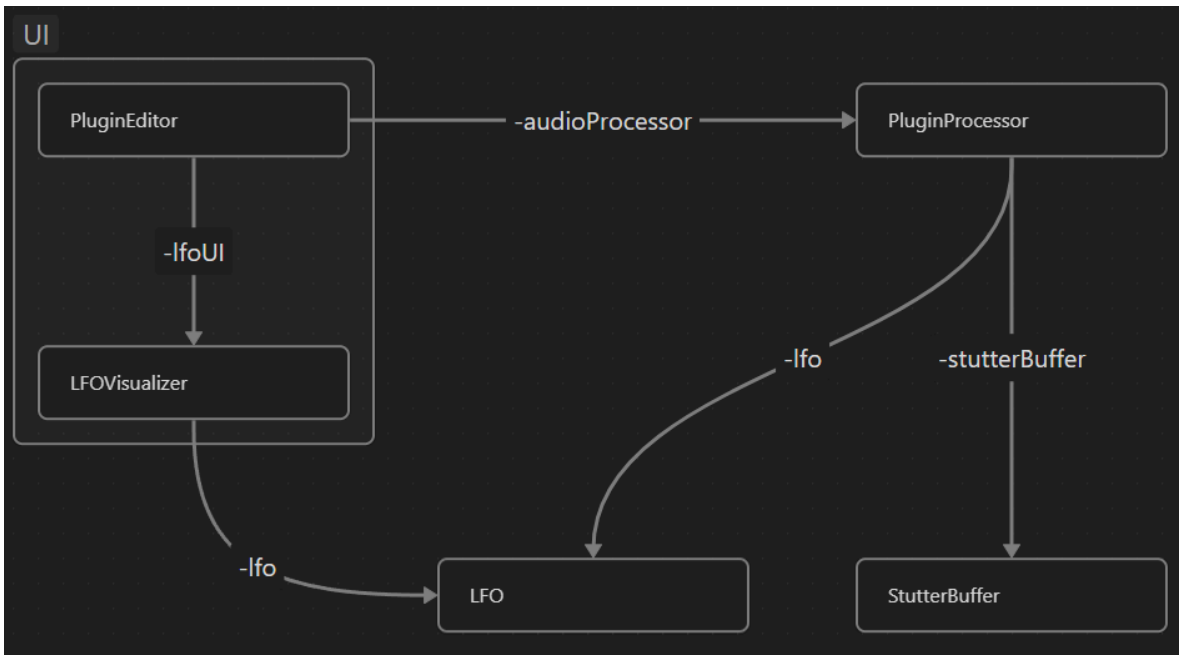

Abbildung 1: Zusammenhänge der Klassen

StutterBuffer

Der *StutterBuffer* implementiert die Logik eines FIFO-Buffers, der kontinuierlich die letzten vier Sekunden des aktuellen Audiosignals speichert und aus dem sich bedient werden kann, wenn ein Signal mit einer bestimmten Länge mehrmals wiederholt werden soll. Zur Umsetzung dieser Logik hält der *StutterBuffer* die Member-Variablen *ringBuffer* und *stutterBuffer* vom Typ *AudioBuffer<float>*. Die Methode *prepareToPlay()* wird dafür genutzt die beiden *AudioBuffers* die vorgesehene Größe zu geben. Als Parameter *maximumStutterSize* wird beispielsweise die im *PluginProcessor* übergebene Samplerate gesetzt, sodass der *stutterBuffer* einer zeitlichen Größe von einer Sekunde entspricht und der *ringBuffer* das vierfache davon. Der Variable *maxStutterIndex* bestimmt die maximale Länge des Audiosignals, das wiederholt wiedergegeben werden soll (Siehe Abbildung 2). Dieser Teil des Audiosignals wird im Folgenden *Stutter* genannt.

```
20  void StutterBuffer::prepareToPlay(int channels, int maximumStutterSize, int maxIndex)
21  {
22      int ringBufferSize = maximumStutterSize * 4;
23      maxStutterIndex = maxIndex;
24
25      ringBuffer.setSize(channels, ringBufferSize);
26      stutterBuffer.setSize(channels, maximumStutterSize);
27
28      ringBuffer.clear();
29      stutterBuffer.clear();
30  }
```

Abbildung 2: Methode prepareToPlay() in der Klasse StutterBuffer

Auf dem Audio-Thread wird die Methode *processBlock* aufgerufen, die wiederum die Methode *process* im *StutterBuffer* aufruft. Mit jedem Aufruf von *process* wird zunächst die Methode *pushBuffer* aufgerufen, die den aktuellen Audioblock in den *ringBuffer* schreibt. Ist der *ringBuffer* voll beschrieben werden die neuen Werte des Audioblocks an den Anfang des Buffers geschrieben. So werden Werte, die aus zeitlicher Sicht länger her sind, als vier Sekunden von neuen Werten überschrieben. Die Member-Variable *ringWriteIndex* wird als Zähler mitgeführt, um beim Auslesen der letzten Werte den Index des neuesten Werts zu kennen (Siehe Abbildung 3).
Sobald ein *Stutter* abgespielt werden soll, wird die Methode *copyStutter* genutzt. Diese schreibt ab dem aktuellen Sample im *ringBuffer* bis zu dem Sample, das eine Sekunde (also die Größe des *stutterBuffers*) in der Vergangenheit liegt, alle Samples in den *stutterBuffer* (siehe Abbildung 4).

```
90  void StutterBuffer::pushBuffer(juce::AudioBuffer<float>& buf)
91  {
92      {
93          for (int chan = ringBuffer.getNumChannels(); --chan >= 0;)
94          {
95              int wIndex = ringWriteIndex;
96              for (int i = 0; i < buf.getNumSamples(); i++)
97              {
98                  ringBuffer.setSample(chan, wIndex, buf.getSample(chan, i));
99                  wIndex = (wIndex + 1) % ringBuffer.getNumSamples();
100             }
101         }
102
103         ringWriteIndex += buf.getNumSamples();
104         ringWriteIndex %= ringBuffer.getNumSamples();
105     }
106 }
```

Abbildung 3: Methode pushBuffer() in der Klasse StutterBuffer

```
108 void StutterBuffer::copyStutter()
109 {
110     for (int chan = ringBuffer.getNumChannels(); --chan >= 0;)
111     {
112         int readSampleIndex = ringWriteIndex;
113         for (int i = stutterBuffer.getNumSamples(); --i >= 0;)
114         {
115             int destSampleIndex = i;
116             readSampleIndex = (readSampleIndex - 1 + ringBuffer.getNumSamples()) % ringBuffer.getNumSamples();
117             stutterBuffer.setSample(chan, destSampleIndex, ringBuffer.getSample(chan, readSampleIndex));
118         }
119     }
120
121     applyCrossfade();
122     stutterReadIndex = 0;
123     copyStutterToggle.set(false);
124 }
```

Abbildung 4: Methode copyStutter() in der Klasse StutterBuffer

Sobald ein *Stutter* kopiert wurde, kann dieser in der Methode *process* des *StutterBuffers* in die Blöcke des *AudioBuffers* geschrieben werden, der als Audio-Output in der DAW ausgegeben wird. Dabei wird *stutterReadIndex* als Zählvariable mitgeführt, um die aktuelle Position im *stutterBuffer* nach jedem verarbeiteten Sample-Block festzuhalten. Überschreitet *stutterReadIndex* *maxStutterIndex*, wird entweder *currentRepeats* inkrementiert, also festgehalten, dass ein *Stutter* einmal mehr wiederholt wurde oder ein neuer Samplebereich wird aus dem *ringBuffer* in den *stutterBuffer* kopiert. Dies ist der Fall, wenn *currentRepeats* größer wird, als die über die Benutzeroberfläche angegebene Wiederholungszahl, die in der Variable *stutterRepeats* festgehalten wird.

Ein *Stutter* hat eine festgelegte Minimaldauer von 20 ms, was bei einer Samplerate von 44100 Hz einer ungefähren Sampleanzahl von 882 entspricht. Das bedeutet, dass ein *Stutter* immer länger ist als ein Audioblock und bei einem Block der Fall eintreten wird, dass die übriggebliebenen Samples eines *Stutters* nicht den kompletten Output-Buffer füllen. In diesem Fall muss entweder der Beginn desselben *Stutters* in den Output-Buffer geschrieben

werden oder der Beginn eines neuen *Stutters*, welcher zuvor aus dem *ringBuffer* über *copyStutter* kopiert werden muss. Dieser Fall wird im Code aus Abbildung 5 in Zeile 61-76 behandelt.

```
32  void StutterBuffer::process(juce::AudioBuffer<float>& buffer)
33  {
34      pushBuffer(buffer);
35
36      if (stutterState.get())
37      {
38          if (copyStutterToggle.get() || stutterBuffer.hasBeenCleared())
39          {
40              copyStutter();
41          }
42          else
43          {
44              int numSamples = buffer.getNumSamples();
45              int maxSample = juce::jmin(numSamples, int(maxStutterIndex - stutterReadIndex));
46              bool paramsUpdated = anyParameterUpdated(numSamples);
47
48              for (int chan = buffer.getNumChannels(); --chan >= 0;)
49              {
50                  int readIndex = stutterReadIndex;
51                  int repeats = currentRepeat;
52
53                  for (int samp = 0; samp < maxSample; samp++)
54                  {
55                      if (paramsUpdated) rampParameters();
56                      buffer.setSample(chan, samp, getInterpolatedSample(stutterBuffer, chan, readIndex));
57                      readIndex = readIndex + 1;
58                  }
59              }
60
61              if (maxSample < numSamples)
62              {
63                  if (currentRepeat >= int(stutterRepeats - 1))
64                  {
65                      copyStutter();
66                      currentRepeat = 0;
67                  }
68                  for (int chan = buffer.getNumChannels(); --chan >= 0;)
69                  {
70                      for (int samp = 0; samp < numSamples - maxSample; samp++)
71                      {
72                          if (paramsUpdated) rampParameters();
73                          buffer.setSample(chan, samp, getInterpolatedSample(stutterBuffer, chan, samp));
74                      }
75                  }
76              }
77              stutterReadIndex += numSamples;
78              if (stutterReadIndex >= maxStutterIndex)
79              {
80                  currentRepeat = (currentRepeat + 1) % int(stutterRepeats);
81              }
82              stutterReadIndex %= int(maxStutterIndex);
83          }
84      }
85  }
```

Abbildung 5: Methode process() in der Klasse StutterBuffer

Der Parameter *ratio* im *StutterBuffer* setzt die Abspielrate oder Wiedergabegeschwindigkeit der ausgegebenen *Stutter.* Dabei entspricht eine *ratio* von 1 der Originalgeschwindigkeit, eine *ratio* von 2 der doppelten Geschwindigkeit und 0.5 der halben. Des Weiteren kann *ratio* auch negative Werte annehmen. Diese spiegeln den Effekt der positiven Werte, wobei der

Stutter mit der jeweiligen Geschwindigkeit rückwärts aus dem *StutterBuffer* gelesen und damit auch rückwärts in den Output-Buffer geschrieben wird. Das Anpassen der Abspielrate wird über lineare Interpolation in der Methode *getInterpolatedSample* realisiert (siehe Abbildung 6).

```
123  float StutterBuffer::getInterpolatedSample(juce::AudioBuffer<float>& buf, int channel, int currentIndex)
124  {
125      double absRatio = std::abs(ratio);
126      float inputIdx = currentIndex / absRatio;
127
128      int x1 = static_cast<int>(std::floor(inputIdx));
129      int x2 = x1 + 1;
130      if (x1 >= maxStutterIndex - 1) x1 = maxStutterIndex - 1;
131      if (x2 >= maxStutterIndex) x2 = maxStutterIndex - 1;
132      float fraction = inputIdx - x1;
133
134      if (ratio < 0.0)
135      {
136          x1 = maxStutterIndex - 1 - x1;
137          x2 = maxStutterIndex - 1 - x2;
138          if (x1 <= 0) x1 = 0;
139          if (x2 <= 0) x2 = 0;
140      }
141      float y1 = buf.getSample(channel, x1);
142      float y2 = buf.getSample(channel, x2);
143
144      return y1 + fraction * (y2 - y1);
145  }
```

Abbildung 6: Methode getInterpolatedSample in der Klasse StutterBuffer

LFO

Die Parameter des *StutterBuffers* lassen sich neben Benutzereingaben auf der Benutzeroberfläche auch über ein LFO steuern. Zur Berechnung der LFO-Werte existiert die Klasse *LFO.* Zentral dabei steht die Funktion *updateLFOState.* Falls die Member-Variable *isEnabled* auf *true* gesetzt ist, wird *updateLFOState* mit jedem Aufruf von *processBlock* im *PluginProcessor* aufgerufen. Mit jedem Aufruf wird eine Variable inkrementiert, die die Phase einer periodischen Funktion repräsentiert. In Abbildung 7 ist in dem Code-Snippet in Zeile 31 und 32 zu sehen, dass die bisher verfügbaren Wellenformen eine Sinuswelle und eine Squarewave sind. Eine mögliche zukünftige Erweiterung könnte das Hinzufügen von weiteren Wellenformen sein.

Die Frequenz des LFOs lässt sich entweder über einen Regler bestimmen, der Werte in der Einheit Hertz zurückgibt oder über die BPM die während der Laufzeit aus der DAW gelesen werden kann. Die an die BPM angepasste Frequenz erschließt sich dann aus der Formel in Abbildung 8.

```cpp
void LFO::updateLFOState(int bufferSize)
{

    if (syncEnabled) {
        freq = syncedFreq();
    }

    float lfoPhaseIncrement = float(juce::MathConstants<float>::twoPi * freq) / (float(this->sampleRate / bufferSize));

    switch (waveType) {
        case Sine:     currentValue = (std::sin(phase) / 2.f + 0.5f); break;
        case Square:   currentValue = phase < juce::MathConstants<float>::pi ? -1.0f : 1.0f; break;
    }

    phase += lfoPhaseIncrement;

    if (phase >= 2 * juce::MathConstants<float>::pi)
        phase -= 2 * juce::MathConstants<float>::pi;

}
```

Abbildung 7: Methode updateLFOState in der Klasse LFO

```cpp
double LFO::syncedFreq()
{
    return bpm / 60.0 * syncFactor; // bei bpm 120 & syncFaktor 0.5 = 1
}
```

Abbildung 8: Funktion syncedFreq in der Klasse LFO

PluginProcessor

Der *PluginProcessor* hält Instanzen des *LFOs* und des *StutterBuffers*. In *processBlock* werden zunächst DAW-Daten ausgelesen, die beispielsweise Auskunft darüber geben, welche BPM in der DAW eingestellt ist, was dann an die *LFO*-Instanz weitergegeben werden kann. Da die *LFO*-Instanz Parameter des *StutterBuffers* ändern kann, geschieht dies bevor *stutterBuffer.process* aufgerufen wird (Siehe Abbildung 9).

```cpp
void GlitchPluginAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::AudioPlayHead* playhead = getPlayHead();
    if (playhead) {
        isPlaying = playhead->getPosition()->getIsPlaying();
    }

    if (isPlaying) {
        if (lfo.isEnabled) {
            updatePositionInfoForLFO(playhead);
            lfo.updateLFOState(buffer.getNumSamples());
            modulateStutterParameters();
        }
        stutterBuffer.process(buffer);
    }
}
```

Abbildung 9: Methode processBlock in PluginProcessor

Falls der *LFO* aktiv ist, werden die Parameter des *StutterBuffers* in der Methode *modulateStutterParameters* moduliert. Dabei lässt sich für jeden Parameter eine jeweilige Modulationstiefe benutzerdefiniert festlegen, sodass es möglich ist, den LFO unterschiedlich stark auf die einzelnen Parameter anzuwenden (siehe Abbildungen 10).

```
215  void GlitchPluginAudioProcessor::modulateStutterParameters()
216  {
217      if (durationModDepth > 0.f)
218      {
219          float minValue = stutterBuffer.getOrigDuration() * (1 - durationModDepth);
220          int modulatedDurationInSamples = std::max(float(lfo.getCurrentValue() * durationModDepth * stutterBuffer.getOrigDuration() + min\
221          stutterBuffer.setStutterDurationInSamples(modulatedDurationInSamples);
222      }
223      if (repeatModDepth > 0.f) {
224          float minValue = stutterBuffer.getOrigRepeats() * (1 - repeatModDepth);
225          stutterBuffer.setStutterRepeats(std::max(float(lfo.getCurrentValue() * repeatModDepth * stutterBuffer.getOrigRepeats()), 1.f));
226      }
227      if (ratioModDepth > 0.f) {
228          float minValue = -2.f;
229          float lfoValue = (lfo.getCurrentValue() * 2 - 1) * ratioModDepth;
230          stutterBuffer.setRatio(std::max(float(stutterBuffer.getOrigRatio()) + lfoValue, minValue));
231      }
232  }
```

Abbildung 10: Methode modulateStutterParameters in PluginProcessor

PluginEditor und LFOVisualizer

Die Benutzeroberfläche, die in Abbildung 11 zu sehen ist, erschließt sich aus den beiden Komponenten des *PluginEditors* und des *LFOVisualizers*. Das Kopieren und Wiederholen eines *Stutters* wird über den Button mit der Aufschrift “Stutter On” angestoßen. Rechts daneben befinden sich die Regler der drei Parameter des *StutterBuffers.* Unter ihnen sind Regler, die die jeweiligen Modulationstiefen der einzelnen Parameter bestimmen.

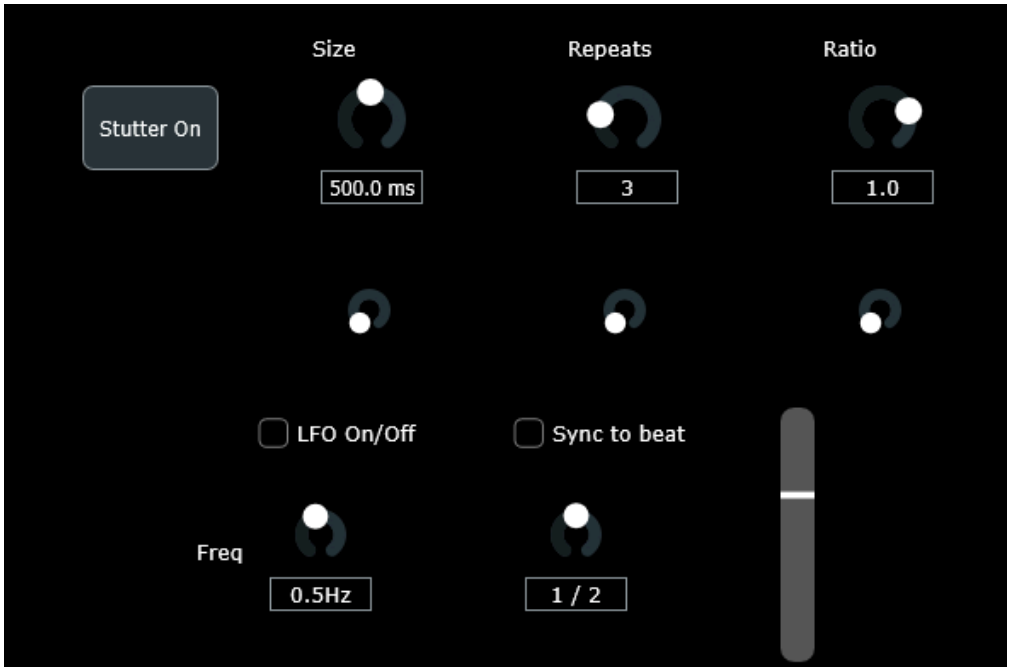

Abbildung 11: Benutzeroberfläche des GlitchPlugins

In der unteren Hälfte der Benutzeroberfläche befinden sich Eingabemöglichkeiten zum Steuern des LFOs. Rechts davon ist eine Visualisierung des aktuellen LFO-Zustandes zu sehen an der sich der Benutzer orientieren kann.

Fazit

Zusammengefasst wurde ein Großteil der Funktionen im *GlitchPlugin* umgesetzt, die auch in *Fracture's Buffer Module* zu finden sind. Dazu gehören die Realisierung eines *StutterBuffers* und die dazugehörigen drei Hauptparameter *stutterDuration, stutterRepeats* und *ratio*. Ein LFO wurde ebenfalls umgesetzt, der noch erweiterbar ist, durch beispielsweise mehr Wellenformen oder die Option, in bestimmten Zeitabständen zufällige Werte für die Hauptparameter zu generieren. Eine weitere Erweiterungsmöglichkeit für dieses Plugin könnte neben einer Umsetzung der "Glitch-Ästhetik" im zeitlichen Bereich auch zufällige oder kontrollierte Verschiebungen im spektralen Bereich bereitzustellen.