Laboration 3 – Sonic interaction design

I denna laboration ska ni använda ljudprogrammeringsmiljön SuperCollider (SC). SC är en realtidssyntesmiljö med stora möjligheter att koda ljudsyntes och signalbehandling. SC finns för Mac, Linux och Windows och kan laddas ned här:
https://supercollider.github.io/

Uppgiften

Uppgiften går ut på att göra ett procedurellt ljud till ett nytt och innovativt bilmärke. Bilmärket är nytt och tillverkar högteknologiska elektriska bilar. Deras värdeord/slogan är:

High tech transport solutions for the young adults
with sustainability for the future.

Och, dessa värdeord genomsyrar allt de gör, från marknadskommunikation via tekniska lösningar till design av bilarna.

Er uppgift är att skapa procedurellt ljud för backsensorn på bilen. Ljudet ska reflektera värdeorden samt ge rätt information och rätt känsla för användaren.

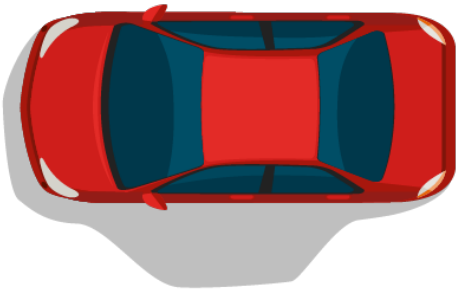
Funktionen i ljudet ska vara:

1. När avståndet mellan bil och hinder bakom bilen är längre än 3 meter ska inget ljud användas.
2. När avståndet är kortare än 3 meter men längre än 30 cm ska ett ljud användas. Avståndet här ska sonifieras på något sätt så att föraren får information om avståndsförändringen.
3. När avståndet är 30 cm eller kortare ska detta tydligt sonifieras och förklaras för föraren.

Inför laborationen

Förberedelse – 1. Tänk på vilket ljud som ska användas

Fundera över värdeorden för företaget. Vad symboliserar *high tech*, vad sänder en signal om *unga vuxna*, på vilket sätt kan *sustainability* återges i ljud?



Och bestäm om ni ska göra ljudet helt procedurellt eller om ni ska spela in ljud som sedan procedurellt förändras och anpassas.

Här följer några länkar för inspiration, först info om ljudlogotyper:
https://soundlogo.wikimedia.org/learn-more/

Sound design: tell a story without using words
https://soeliok.com/en/blog/sound-design-tell-a-story/

Basic Semantics of Product Sounds
http://www.ijdesign.org/index.php/IJDesign/article/view/957/473

Och här finns info om sonic interaction design:
https://en.wikipedia.org/wiki/Sonic_interaction_design

Förberedelse – 2. Bekanta er med koden som ges i zip-filen

Synthdefinitioner

Koden har en kort/liten synthdefinition (`distanceSonification`) som från början innehåller en triangelvåg (`LFTri.ar`) i 220Hz och med en ljudvolymsförändring (`level`) vilken styrs av när bilen är 3 meter eller närmre ett hinder bakom och musknappen är nedtryckt eller inte. Ljudvolymen i sin tur ändras med hjälp av `lag(1)`, vilket innebär att ljudet går från 0 till 1 respektive från 1 till 0 på en sekund.

När ni löser uppgiften så skapar ni fler synthdefinitioner om ni anser att ni vill ha det.

Det finns en till synthdefinition (`sendMouse`) som skickar muskoordinaterna och musklick via OSC till klientsidan. Det som kan vara lite intressant att titta på i denna synthdefinition är `Impulse.kr(60)`. Detta ställer hur ofta muskoordinaterna ska kollas och skickas via OSC, skickas det för sällan blir responsen i interaktionen väldigt seg och är frekvensen för hög skickas antagligen data i onödan. Läs mer här:
https://doc.sccode.org/Classes/Impulse.html
https://www.daskeyboard.com/blog/guide-to-mouse-polling-rate/

Klientsideskript

På klientsidan har jag försökt sortera koden så att de viktigaste delarna för er kommer i början. Först stoppas synthdefinitionen på servern in i `~distanceSynth`.

Därefter följer en funktion `~adjustTheSonification` som tar emot ett inputargument (`distance`), och sedan har en `case`-sats (eftersom SuperCollider inte har else if) med de tre avstånden som specificerats uppe bland funktionaliteten. I slutet av funktionen sätts `~distanceSynth` i relation till det ni kommer att sätta för de olika avstånden.

Längre ner i koden kommer GUI-grejer, definitioner av flaggor om musknappen trycks ned i GUIt, en muslyssnare registreras, följt av `OSCFunc` som är det som lyssnar på musmeddelandena och sedan ropar på funktionen `~adjustTheSonification` och skickar med avståndet, och avslutningsvis finns en `CmdPeriod` vilket är en funktion som körs när ni stoppar koden med Cmd + punkt i MacOS och Ctrl + punkt i Windows. Jag är osäker på om detta funkar i Windows… Det som görs i den funktionen är att stänga ned fönstret och ta bort OSC-lyssnaren.
https://doc.sccode.org/Classes/CmdPeriod.html

Skapa kopplingar mellan data och ljud

Lag

Lag har redan nämts ovan och är ett bra sätt att skapa en ”fade” mellan två värden. Ibland kan man använda det som en väldigt enkel envelop med en attacktid och en releasetid som i exempelkoden. Men, det kan ibland vara användbart att ha en kort `lag` för att förhindra att det uppstår konstiga ljud när ett värde ändras väldigt snabbt. Ta exempelvis följande:

```
var filtered = LPF.ar(soundIN, cutOffFreq);
```

Om `cutOffFreq` ändras från exempelvis 5000Hz till 150Hz direkt så kan det uppstå missljud, som knäpp eller ett kort knaster. Då kan följande vara bättre:

```
var filtered = LPF.ar(soundIN, cutOffFreq.lag(0.1));
```

Denna fördröjning kan också användas som `Lag.ar` eller `Lag.kr`, genom `Lag.kr(in, lagTime, mul, add)`. Läs mer här:
https://doc.sccode.org/Classes/Lag.html

Mappning av värden

Ofta behövs värden transformeras för att vara anpassade till en förändring som ska göras i ljudet. Denna mappning av data kan naturligtvis göras med en matematisk formel, men det går också att göra tämligen enkelt genom färdiga funktioner i SuperCollider. Dessa funktioner kallas linlin respektive linexp, och är en linjär skala till en linjär skala respektive en linjär skala till en exponentiell skala.

På klientsidan skrivs dessa som `var newValue = linlin(inputValue, inputMin, inputMax, outputMin, outputMax);`.

Följande kod:

```
var inputValue = 0.64;
var newValue = linlin(inputValue, 0, 1, 0, 255);
```

ger att `newValue` är 163.2.

Medan:

```
var newValue = linexp(inputValue, 0, 1, 0, 255);
```

ger att `newValue` är `nan`. Detta beror på att `linexp` inte kan ha ett `outputMin` som är 0. Istället:

```
var newValue = linexp(inputValue, 0, 1, 1, 256) - 1;
```

vilket ger att `newValue` är 33.78 (ungefär).

Läs mer här:
https://doc.sccode.org/Classes/LinLin.html
https://doc.sccode.org/Classes/LinExp.html

Variabler

Fler variabler kan med säkerhet behövas läggas till i början av funktionen `~adjustTheSonification`, i grundkoden finns ju bara `level`. Variabler kan deklareras på varsin rad och de behöver inte deklareras med ett värde, som `var level;`. Det kan vara bra att deklarera en variabel och tilldela den ett värde, så att man är säker på att det alltid finns ett värde när variabeln används. Det går också att kolla om en variabel har ett värde innan den används
https://doc.sccode.org/Classes/Nil.html

Variablerna kan skapas på flera rader, eller flera på samma rad, `var frequency, level, tempo;`. Fundera på vad som är mest lättläst och pedagogiskt, och kom ihåg att variabler måste deklareras först i en funktion i SuperCollider.

Att förändra tempo, frekvens, ljudvolym

Det går att skapa ett tempo eller puls exempelvis genom att skapa en fyrkantsvåg (`LFPulse.kr`) med en `range` på 0 och 1 som amplitudmodulerar en annan vågform. Frekvensen på fyrkantsvågen kan sedan förändras, från exempelvis 1Hz och uppåt till 10Hz, för att skapa förändringen i tempo. Med hjälp av `width:`, dvs pulsbredden, kan också skillnaden mellan ljud och tyst i förändras i relation till inputargument. Det går ju också

förändra tonhöjden på en vågform med en fyrkantsvåg, exempelvis för att skapa omväxlande en högre ton och en lägre ton.

I stället för fyrkantsvåg/pulsvåg går det att använda de andra vågformerna, en triangelvåg för att svepa ljudvolymen eller tonhöjden linjärt mellan hög och låg och hög, en sinus för att skapa en förändring som saktar ner lite vid vändningarna, eller en sågtand för att få ett uppåtgående svep som startar om (detta ändras rätt lätt till att få en fallande förändring).

Fyrkantsvågen kan också användas som en gate-signal (dvs den simulerar nedtryckningar av tangenten på en klaviatur) som används i en envelopgenerator så kan man använda olika attack- och releasetid:

```
var amMod = LFPulse.kr(1, width: 0.1);
var envelope = EnvGen.kr(Env.perc(0.01, 1), gate: amMod);
```

Det finns andra sätt att skapa förändringar på, exempelvis med `line`. I `line` sätts ett startvärde och ett slutvärde och hur lång tid den här förändringen ska ta. Fördelen med `line` är att den inte startar om/loopar som exempelvis `LFSaw` gör. Med `XLine` skapas en exponentiell förändring. Läs mer här:
https://doc.sccode.org/Classes/Line.html
https://doc.sccode.org/Classes/XLine.html

Att ändra omfång av värden eller vågformer

Ett ljud i SuperCollider svänger runt 0 mellan 1 och -1. Med hjälp av `range` kan detta omfång ändras. Följande exempel skapar en sinusvåg i 220Hz som svänger runt 0.5 mellan 1 och 0:

```
var sinWave = SinOsc.ar(220).range(0, 1);
```

Detta hade kunna göras genom att ändra amplituden av sinusen och sedan DC-offseten, med:

```
var sin2 = SinOsc.ar(220, mul: 0.5, add: 0.5);
```

Men, `range` är bekvämt om man exempelvis vill förändra/svepa brytfrekvensen i ett filter mellan 150 och 2500Hz:

```
var lfo = SinOsc.kr(1).range(150, 2500);
var filteredSound = LPF.ar(input, lfo);
```

Om man vill ändra tonhöjd/frekvens på ett ljud så är det bra att använda MIDInotnummer. Dessa är nummer på tangenter på ett klaviatur och är kopplade till korrekt frekvens för den tonen. Exempelvis tonen E3 (MIDInotnummer 52) har frekvensen 164.81Hz och vill man sedan byta ton till F3 (MIDInotnummer 53) så har den frekvensen 174.61. Man ändrar sedan från MIDInotnummer med hjälp av `midicps` som står för MIDI till cycles per second (vilket är frekvensen i Hz). Det går också att ändra från frekvens i Hz till MIDInotnummber med `cpsmidi`.

Det kan vara bra att tänka på här att en oktav är en fördubbling av frekvensen, dvs tonen A3 har frekvensen 220Hz och A4 har dubbelt så hög frekvens på 440Hz, men att det bara är 12 toner (med halvtonerna) mellan dessa två tangenter på en klaviatur. Så tonen A3, dvs MIDInotnummer `57 + 12` är en oktav upp, medan `57.midicps * 2` är en oktav upp i frekvens.

Läs mer här:
https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies
https://doc.sccode.org/Classes/AbstractFunction.html

Skapa många syntar och doneAction: 2

Ofta skapar vi en environmentalvariable (med `~`) eller en vanlig variabel (med `var`) för en synt på klientsidan. Exempelvis:

```
~distanceSynth = Synth.new(\distanceSonification).register;
```

i grundkoden i den här laborationen. Vi kan redan när vi registrerar synten skicka med argument, typ:

```
~distanceSynth = Synth.new(\distanceSonification, [\frequency, 220, \level, 1]).register;
```

Men, vi behöver inte lägga synten i en variabel. Vi kan skapa en ny synt varje gång vi behöver den. Typ så här i en OSC-lyssnare:

```
OSCdef(\sensorlistener, { arg inputmsg;
   if (inputmsg.size > 1) {
      Synth.new(\distanceSonification, [\frequency, 220, \level, 1]).register;
   };
}, 'sensordata');
```

Det innebär att varje gång som `inputmsg` är större än `1` så skapas en ny instans av synten på servern. Om den synthdefinitionen innehåller ett ”plingljud” så spelas det upp och vi behöver inte skicka `gate`-information eller så till synthdefinitionen. Det finns också en fördel i att om två meddelanden kommer väldigt tätt kommer båda att spelas upp i varsin synt i stället för att det andra meddelandet kommer att störa uppspelningen av det första.

Men, det finns också en nackdel med det här. Om vi skapar nya instanser av synten hela tiden kommer vi sakta men säkert köra fler och fler processer, och datorn kommer så småningom att klaga för att SuperCollider kräver för mycket resurser. Därför behöver vi använda en så kallad `doneAction` i synthdefinitionen.

En `doneAction` är kopplat till något som har en tidsaspekt i synthdefinitionen, som en `PlayBuf` eller en `Line` eller en `EnvGen`. Enklast skriver man en `doneAction` med just `doneAction:` följt av vilken action som ska gälla. Då slipper man komma ihåg exakt hur många inputargument man använder och var man ska stoppa in `doneAction`. Default är action 0 som betyder att inget ska göras när tidsaspekten är klar, men i det här fallet vill vi använda 2 som betyder att synten tas bort från servern.

Läs mer här:
https://doc.sccode.org/Classes/Done.html

Ta följande exempel:

```
SynthDef(\distanceSonification, { arg freq = 220;
   var triWave = LFTri.ar(freq);
   var envelope = EnvGen.kr(Env.perc(0.01, 1), doneAction: 2);
   var output = triWave * envelope;
   Out.ar(0, {output}!2);
}).add;
```

Den här synthdefinitionen spelar upp ett kort ljud, men en snabb attack (0.01 sekund) och klingar ut i 1 sekund (releasetiden) och syntinstansen tas sedan bort från servern med `doneAction: 2` i `EnvGen` som står för tidsaspekten. Grundfrekvensen är 220Hz, men denna kan ändras genom att ta emot ett inputargument.