Linköpings universitet

Ljudfysik

TFYA65

Civilingenjör i medieteknik

MT3B

# Visualisering av ljudintensiteter och frekvenser inuti en spelmiljö

*Författare:*
Samuel Svensson | samsv787
*Författare:*
Christoffer Paulusson | chrpa821
*Författare:*
Adam Morén | adamo472

*Examinator:*
Per Sandström
*Examinator:*
Peter Andersson

14 oktober 2018

**Sammanfattning**

I kursen Ljudfysik TNFYA65 som hålls vid Linköpings Universitet så är ett av delmomenten ett projektarbete. Syftet med arbetet är att fördjupa sig inom något område av kursen och på så vis öka förståelsen av ljudfysik. I den här rapporten beskrivs ett sätt att visualisera ljud från en ljudfil, med hjälp av spelmotorn Unity, för att sedan skapa ett enklare spel där spellogiken baseras på ljudet. Det främsta fokuset ligger på processen bakom skapandet av spelet och problemområdena kring att skapa interaktiva objekt från ljudet.

**Contents**

## 1. Bakgrund

Som Civilingenjör i Medieteknik är det viktigt att kunna visualisera data av olika typer. Ljud är en stor del inom de flesta teknikområden vilket gör det intressant. Det är viktigt att ha förståelse för ljudets egenskaper och hur dessa kan användas till vår fördel vid skapandet av applikationer, spel och liknande. Framförallt är det digitaliserade ljudet intressant, då det är detta som ger den data som vi kan hantera med kod. Hur datan sedan hanteras beror på ett antal parametrar som anpassar resultatet med hänsyn till syftet. Projektets kärna låg i att visualisera ett ljudspektrum på ett så representabelt sätt som möjligt och hitta ett effektivt sätt att extrahera ljudsignaler till data som kan hanteras på olika sätt. Spelmotorn Unity innehåller flera verktyg såsom funktioner och Fouriertransformeringar som kan hjälpa en på vägen men att analysera en ljudfils frekvenser och konvertera dessa till visuell data visade sig vara en utmaning.

### 1.1. Syfte

Projektets syfte var att kunna visualisera ljud på ett representabelt sätt för att skapa ett spel som återkopplar en känsla av ljudets intensitet och frekvens.

## 2. Metod

Till att börja med samplades en ljudfil. I en dator så samplas ljud vid 48000Hz. Men människan hör endast ljud upp till högst hälften av detta så man behöver endast fokusera på frekvenser mellan 0Hz och 24000Hz. En matris skapades med en storlek med 1024 platser som representerade samplingsfrekvensen och med detta kunde frekvensupplösningen beräknas enligt nedan.

$$\frac{fs}{N} = \frac{24000}{1024} = 23,4Hz \tag{1}$$

Varje element i matrisen innehöll då den relativa amplituden (normerade värden mellan 0 till 1, där 0 är 0Hz och 1 är 24000Hz). Första platsen i matrisen (samples[0]) innehåller då alla frekvenser mellan 0-23,4Hz och sedan är resterande platser linjärt uppdelade på samma sätt upp till 24000Hz. Sedan använder man sig utav en funktion inbyggd i Unity som kallas *GetSpectrumData* som hämtar ljudspektrumet från ljudfilen. Funktionen *GetOutputData* hämtar alla sampel från ljudfilen för att kunna ta reda på dB-värdet vid alla tidpunkter. [2] Genom att summera alla sampel från denna data, kvadrera dessa och sedan ta medelvärdet av detta så togs *RMS*-värdet fram. *RMS* står för *root mean square* och betecknas $x_{rms}$ i formeln nedan.

$$x_{rms} = \sqrt{\frac{1}{n}(x_1^2 + x_2^2 + ... + x_n^2)} \tag{2}$$

En beräkning av dB-värdet (medelvärdsintensiteten) gjordes för varje element var 23,4:e ms för att kunna till exempel modifiera bakgrundsfärgen i takt med musiken. Med *RMS*-värdet konverterades datan till dB genom ekvationen nedan.

$$dB = 20 \cdot log_{10}(\frac{r_1^2}{r_2^2}) \tag{3}$$

där $r_1^2$ är vårat RMS-värde och $r_2^2$ är ett referensvärde som i detta fall är 1024.

För att extrahera de specifika frekvenserna och fördela det till ett kubobjekt är processen lite mer komplicerad. Först hittades den dominanta frekvensen som sedan appliceras till kubobjektet. Sedan hittades det element i matrisen som hade högst amplitud och detta värde multiplicerades med frekvensupplösningen. I de fall då frekvensen hamnade mellan två element så krävdes en interpolering mellan värdena för att få ett bättre resultat. Det här fungerar i teorin bra vid mindre komplexa ljud som sinusoidvågor, fyrkantsvågor eller triangelvågor men när riktig musik tillämpas finns det en chans att man får en missrepresentation. Detta är på grund av att i musik så ligger övertonerna högre än grundtonerna och man kan ofta höra flera olika toner samtidigt till skillnad från en vanlig sinusvåg där en ton hörs konstant. Ett problem med att dela upp frekvenserna linjärt på det här sättet är att det är stora skillnader på de lägre frekvenserna medan det är svårt för människor att höra skillnader i de högre registrena. Mellan 50Hz och

100Hz är det en hel oktavs skillnad vilket är väl hörbart, medan en 50Hz skillnad på en 10kHz nivå är väldigt svår att urskilja. Frekvenserna 10000Hz och 10050Hz är i princip samma ton (D#9). Därför behöver frekvensområdet delas upp logaritmiskt, för att få en så realistisk representation som möjligt av ljudets intensitetsnivåer.

### 2.1. Funktioner och variabler

Funktionen *AnalyzeSound* mäter volymen av ljudet som spelas i stunden i RMS och dB, och tar fram tonhöjden i Hertz samt Fouriertransformerar allting. Detta gjordes med hjälp av Unitys egna inbyggda funktion *GetSpectrumData* som utnyttjar funktionen *FFTWindow.BlackmanHarris. Blackman Harris window* är en av flera olika tillgängliga fönsterfunktioner som används för att anpassa mätserien vid en frekvensanalys och därmed minimera bakgrundsartefakterna i visualiseringen. Just Blackman Harris fönsterfunktionen anses vara en av de mest främsta inom signalbehandling. [1]

Datan lagrades uppdelad i matriser och tillämpades sedan på objekt i Unity. Dessa förändrades beroende på musikens signal. Vid spelstart genereras ett antal kubobjekt. Matrisen innehållandes alla nivåer applicerades sedan på dessa kubobjekt inuti Unity där man manipulerade deras skalor längs y-axeln beroende på vilken amplitud som gavs av frekvensområdet.

Det var möjligt att definiera antalet kubobjekt som skulle genereras med variabeln *AmnVisual*, där antalet element från matrisen fördelas jämnt ut överallt till varje kub.

En parameter *KeepPercentage* användes för att sträcka ut frekvensspektrumet för att få ett mer jämnt fördelad visualisering, då det ibland kan förekomma att de högsta frekvenserna ( +18000Hz) inte får någon intensitet. Samtidigt fördelades frekvenserna ut till kubobjekten bredvid den kub som får högst värde, så att en mer jämnt fördelad kulle av staplar bildas istället för att endast ett kubojekt skalas upp.

Vid testande av olika låtar och genrer av musik märktes det snabbt att vissa av objekten fick för höga intensiteter och då blev skalade för mycket. Därmed introducerades en tröskelvariabel *MaxValue* som användes till att bandbreddsbegränsa hela signalen för att inte få för höga intensiteter som därmed kan förändra spelupplevelsen på ett negativt sätt.

Samtidigt implementerades en variabel *VisualModifier* som amplifierade skalningen av kuberna så man fick bättre kontroll över hur visualiseringen presenterades.

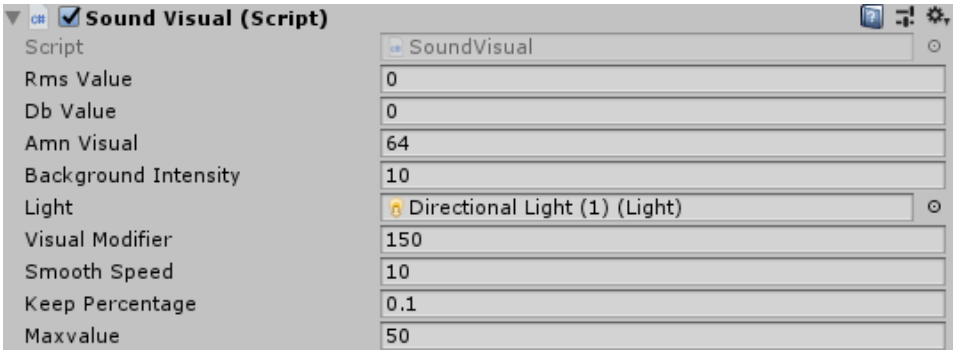
När intensiteterna fördelades till kubobjekten så blev animationerna väldigt hackiga då värdena uppdaterades på millisekunder och det existerade ingen mjuk övergång. För att det skulle bli en mer behaglig upplevelse att titta på så introduceras en variabel *SmoothSpeed* som interpolerade mellan låga och höga hastigheter så att när stapelns höjd förändrades fick kuben en mjuk och långsam rörelse.

När visualiseringen var komplett så började spellogiken implementeras. För att få en bra spelupplevelse bestämdes att spelaren skulle fånga upp mynt samtidigt som den studsade på ljudspektrumet. Det skulle även finnas objekt som gav en kraft uppåt så det var möjligt att hålla sig i luften en längre tid för att kunna nå mynten. Senare ändrades spellogiken så man skulle flyga en rymdraket och undvika ljudspektrumet samt asteroider som föll nedåt. Det skulle finnas batterier som gav raketen en kraft uppåt för att undvika att krascha.

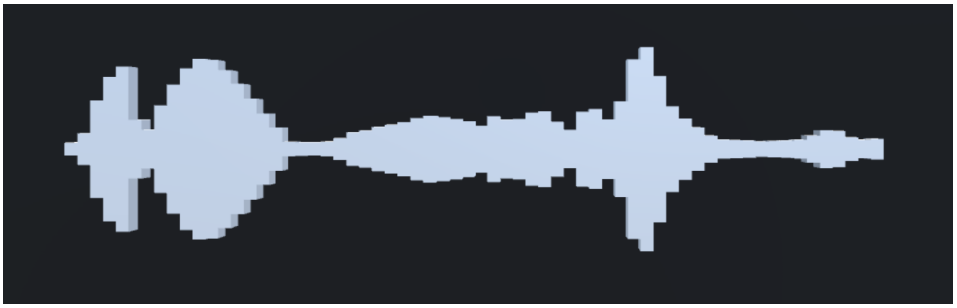
## 3. Resultat

Vid första versionen av det visualiserade ljudet så genererades 64 stycken kuber med hjälp av *AmnVisual*. Med hjälp av experimentering sattes *VisualModifier* till 170, *SmoothSpeed* sattes till 10, *KeepPercentage* sattes till 0.1 och *MaxValue* till 40. Se figur 1.

Genom dessa värden skapades den första versionen av ljusspektrumet. Resultatet visar på ett rörligt spektrum i takt med musiken. I och med att *AmnVisual* sattes till värdet 64, så genererades 64 stycker kubobjekt där varje kubobjekt representerade ett frekvensomfång på 375Hz. Se figur 2.




Figur 1: Huvudskriptet som är kopplat till kubobjekten där alla parametrar kan anpassas.

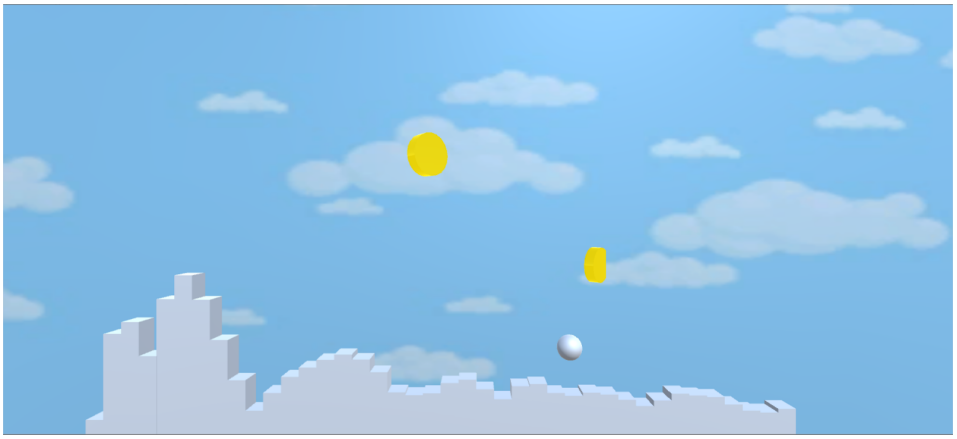


Figur 2: Det första rörliga spektrumet.

För den andra versionen av ljudspektrum användes istället funktionen *SpawnCircle* med samma värden som vid försök 1. Detta resulterade i en liknande visualisering fast med en cirkel istället för en linje. Se figur 3.

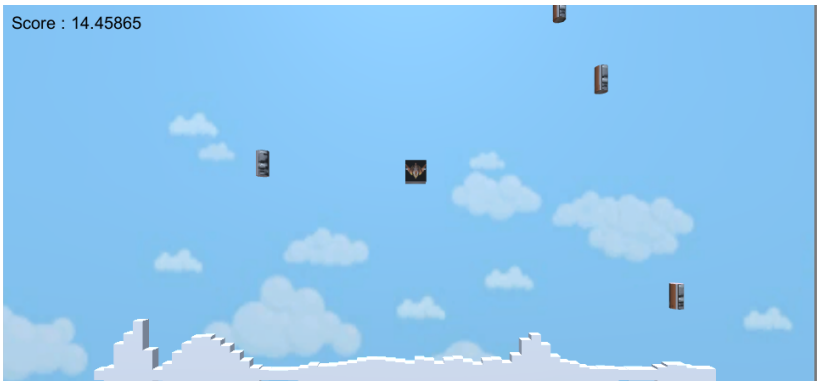
Figur 3: Spektrumet som en cirkel.

Vid första försöket till skapandet av ett spel valdes det att använda det visualiserade spektrumet som olika typer av växande skyskrapor, där spelaren var en boll som skulle samla guldmynt. Se figur 4.



Figur 4: Första utkastet spelet.

Vid andra försöket byttes bollen ut till ett flygande objekt och idén att ha fallande batterier introducerades. Se figur 5.




Figur 5: Andra utkastet av spelet.

Därefter bearbetades spelet ytterligare. En menysida implementerades samt en slutscen där man får möjligheten att spela igen. Se figur 6 och 7.

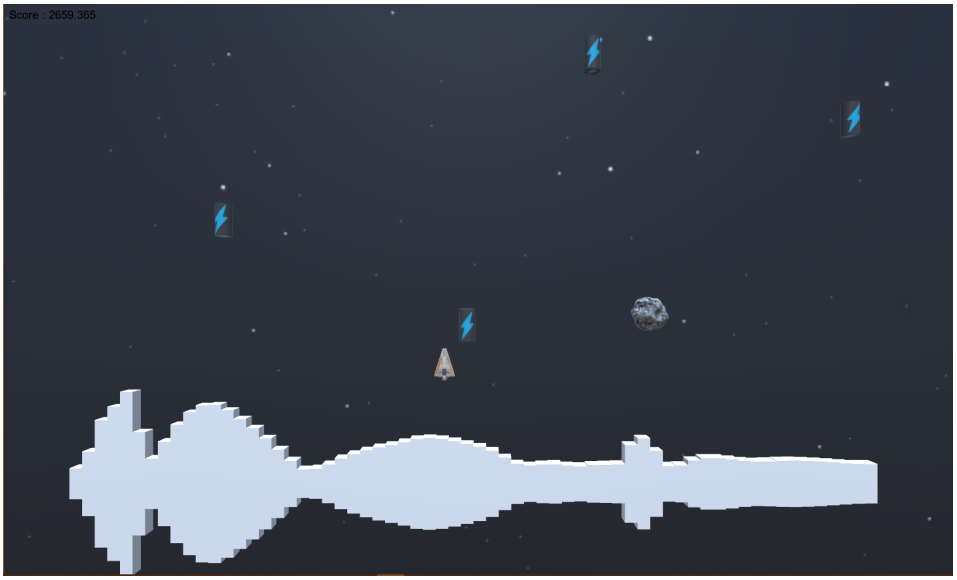

Figur 6: Menysidan som visas när spelet startas.


Figur 7: Menysidan som visas när spelet är över.

Den färdiga produkten innehåller menysidan, spelet samt slutscenen när man kraschat rymdskeppet. Se figur 8.

## 4. Diskussion

Syftet med den här rapporten var att visualisera ljud och att skapa ett spel med koppling till ljudets intensitet och frekvens. Spellogiken skulle baseras på ljudet och det beslutades att den första visualiseringen i figur 2 skulle användas. Detta eftersom gruppen kom fram till att det var enklare att skapa ett spel med ett rakt spektrum, snarare




Figur 8: Det färdiga spelet.

än ett cirkulärt. Genom denna visualisering togs spelets struktur fram i form av en raket som ska ta sig så långt som möjligt utan att krascha. Denna spelstruktur var till följd av den tidigare valda låtens tempo. Det kändes helt enkelt naturligt att ha ett spel som går ut på snabba händelseförlopp, då låten som användes i sig var högintensiv.

Genom den första visualiseringen i figur 2 kunde vi skapa upplevelsen av att raketen blev "jagad" i en hastighet som berodde på ljudet. Detta resulterade i att spelet som slutprodukt till stor del har att göra med ljudfilen som används. En förenkling för oss var att vi redan innan visste vilken låt/ljud som skulle användas. En förbättring av projektet skulle kunna vara att användaren själv fick välja vilken låt denne ville spela med. Då skulle det skulle det dock behöva mer avancerade funktioner som tar hänsyn till egenskaper hos ljudet. Till exempel raketens hastighet och batteriernas effekt kunna vara beroende av ljudets egenskaper. Nu visste vi redan delar av låtens egenskaper från början och kunde på så sätt experimentera fram värden tills det medförde till ett bra resultat.

### 4.1. Slutsats

Med hjälp av Unity har vi visualiserat ett spektrum från en ljudfil och skapat ett spel utav det. Ljudets intensitet speglas genom hastigheten av objektens rörelser i spelet vilket skapar ett mervärde för upplevelsen. Utvecklingsmöjligheter ligger främst inom tillägg av andra typer av objekt och återkopplingar som speglar andra delar av ljudets egenskaper.

---

[1] Stack Exchange , What is a window function in DSP and why do we need it? , *Nov 2016,Link*
[2] Unity Answers, GetOutputData and GetSpectrumData, what represent the values returned? , *Aug 2011,Link*