**Creative Audio-Visual Coding Code Link and Source Code**

**Link to Code:**

https://editor.p5js.org/lilley49/sketches/ieJdEUbva (Accessed: 13 May 2021).

**Source Code:**

var song, // in this section of code I am declaring variables. This is important so that the code when run recognises these variables and doesn't present a syntax error.

amp,

fft,

peakDetect,

clock,

time,

time2,

bar\_time,

line\_time,

clock2,

Linetranslate,

LineTranslate\_Pre,

clocktimer2,

clocktimer3,

clocktimer4,

barone,

bartwo,

barthree,

barfour,

barfive,

rotateVal,

clockrotate,

bar\_,

loop\_var,

i\_val,

clock\_change,

clock\_change\_time,

freq\_heighs,

spacer,

spacer2,

spacer3,

spacer4,

bar6;

let cam;

function preload() {

song = loadSound("cold.mp3"); //This preloads the audio file 'cold.mp3' and assigns it to the variable song. This is important to ensure that the audio file in already loaded before the rest of the code is run because it can sometimes take some time. If the audio file was fully loaded and the code started to run this would cause major issues.

}

function setup() { //In this section most of what I am doing is intialising variables.

createCanvas(windowWidth, windowHeight, WEBGL); //Here I am creating the canvas. I have set it to match the size of the window. WEBGL introduces 3D rendering.

normalMaterial(); //Loads material for 3D shapes.

cam = createCamera();

song.play(); //Plays the song

frameRate(60); //Setting the frame rate to 60Hz

fft = new p5.FFT(); //Instantiation of p5.FFT object assigns it to 'fft' variable. This gives the code the functionality to analyse audio.

peakDetect = new p5.PeakDetect();

peakDetect2 = new p5.PeakDetect(200, 500, 0.15, 30); //Peak detect between 200-500Hz, with a 0.15 threshold and 30 frames per peak. These values were all experimented with to achieve required characteristic for peak detection.

clock = 0;

time = 0;

time2 = 0;

bar\_time = 1000000; //This needed to be intially large to stop for loop condition applying before calcualtion + assignment occurs

clock2 = 0;

clocktimer = 0;

clocktimer2 = 0;

clocktimer3 = 0;

clocktimer4 = 0;

rotateVal = 0;

clockrotate = 0;

freq\_heighs = 0;

amp = 0;

}

function draw() {

bass = fft.getEnergy(60, 250); // assigns variable 'bass' to the volume of the audio file between 60Hz and 250Hz. This is constantly changing (continuous data))

treb = fft.getEnergy(8000, 20000); // assigns variable 'treble' to the volume of the audio file between 8000Hz and 20000Hz.

bassreal = bass - 150; // simple calcualtion to reduce the 'bass' value closer to the range desired. Assigned this to 'bassreal'

//Some of this 'spectrum' and for loop section below is inspired by 'The Coding Train'. Available at: https://www.youtube.com/watch?v=2O3nm0Nvbi4 (Accessed: 10 May 2021).

var spectrum = fft.analyze(1024); //assigns variable 'spectrum' to an array, the specific values in the array, of which there are for 1024 bins, related to the audio files amplitude at different frequency ranges.

for (var i = 0; i < spectrum.length; i++) { // this initiates a for loop. First it sets 'i' equal zero. Then while 'i' is less then 'spectrum.length'/the length of the array/the number of bins the code below bracketed is run and 'i' is increased by one.

var amp = spectrum[i]; // In each iteration of the for loop variable 'amp' is assigned to the value of bin which the for loop is currently focusing on. This bin represented by 'i' the number of iterations the for loop has completed.

if (amp > freq\_heighs) { //A conditional if 'amp' is larger the 'freq\_heighs', 'freq\_heighs' is intially assigned 0.

freq\_heighs = amp; // Then let 'freq\_heighs' = 'amp'. The effect this has is only the largest value from the total of 1024 bins (the bin (frequency range) with the largest amplitude) is stored in 'freq\_heighs' by the end of each instance of the for loop.

// print(i);

i\_val = i; // 'i\_val' is assigned to 'i' which is the number of iterations through the for loop the largest amplitude bin occured and an identifier for this bin.

}

if (i == spectrum.length - 1) { // This if statement resets the values for the next instance in the timeline of the audio.

freq\_heighs = 0;// if this variable was not reset to zero then the code may not properly identify the correct largest bins (frequency ranges). If in one instance of the 'spectrum' array the largest bin was bin 10 and had a value of '9' and in the next instance the largest bin was bin 76 but it only had a value of '6'. 'freq\_heighs' would not be reassigned and recognise bin 76 as the maximum in that instance.

}

}

function colour\_change(I\_val, r, g, b) { // this is a function which I use to change the colour of the shapes. How it works is I have created specific colours which relate to different bins represented by 'i\_val'. I call these functions but only if the specified paramter 'I\_val' is equal to the 'i\_val' at any specific moment then the colour will be applied.

if (i\_val == I\_val) {

fill(r, g, b);

}

}

colour\_change(1, 199, 21, 133); // The functions recalls are all for specific bins (frequency values).

colour\_change(2, 255, 165, 0);

colour\_change(3, 0, 255, 0); // e.g. bin 3 is blue.

colour\_change(4, 144, 0, 255);

colour\_change(5, 255, 69, 0);

colour\_change(6, 255, 174, 66);

colour\_change(7, 154, 205, 50);

colour\_change(11, 255, 0, 0);

colour\_change(12, 255, 69, 0);

colour\_change(13, 255, 165, 0);

colour\_change(14, 255, 255, 0);

colour\_change(15, 154, 205, 50);

colour\_change(16, 0, 255, 0);

colour\_change(17, 13, 152, 186);

colour\_change(18, 13, 152, 186);

colour\_change(19, 199, 21, 133);

colour\_change(20, 144, 0, 255);

colour\_change(21, 144, 0, 255);

colour\_change(22, 255, 0, 0);

colour\_change(25, 255, 69, 0);

colour\_change(43, 144, 0, 255);

colour\_change(54, 255, 69, 0);

function line\_colour\_change(I\_val, r, g, b, delay) { // This function operates in much the same way as the 'colour\_change' function with a couple differences. The stroke object was necessary instead of the fill object as the elements effected are lines and not shapes or spheres as in 'colour\_change'. Also I wanted to include a delay function so that I could control when a certain line a colour change. This was important as I had tried it without a delay and it seemed to make the visual flatter. I wanted to keep and enhance the visuals 3D depth.

let clock\_change = 0; // To create this delay I used a simple conditional. First assign 'clock\_change' to zero.

if (clock\_change == 0) { // If 'clock\_change' = zero then 'clock\_change\_time' equals run time.

clock\_change\_time = millis();

clock\_change = clock\_change + 1; //Increases 'clock\_change' by one. Therefore 'clock\_change' does not equal 0 and 'clock\_change\_time' stays constant for that instance of the function.

}

if (i\_val == I\_val) { //This section of the code is very similar to the 'colour\_change' function. However it also includes another conditional, which only changes the colour when the run time is larger than the static 'clock\_change\_time' and parameter defined delay combined.

if (clock\_change\_time + delay < millis());

stroke(r, g, b);

}

}

function calling\_colours(delay) { //This section is similar to the'colour\_change' recalls but with the addition of the delay parameter. In addition I've included it in a function itself as I plan to use it repeatedly.

line\_colour\_change(1, 199, 21, 133, delay);

line\_colour\_change(2, 255, 165, 0, delay);

line\_colour\_change(3, 0, 255, 0, delay);

line\_colour\_change(4, 144, 0, 255, delay);

line\_colour\_change(5, 255, 69, 0, delay);

line\_colour\_change(6, 255, 174, 66, delay);

line\_colour\_change(7, 154, 205, 50, delay);

line\_colour\_change(11, 255, 0, 0, delay);

line\_colour\_change(12, 255, 69, 0, delay);

line\_colour\_change(13, 255, 165, 0, delay);

line\_colour\_change(14, 255, 255, 0, delay);

line\_colour\_change(15, 154, 205, 50, delay);

line\_colour\_change(16, 0, 255, 0, delay);

line\_colour\_change(17, 13, 152, 186, delay);

line\_colour\_change(18, 13, 152, 186, delay);

line\_colour\_change(19, 199, 21, 133, delay);

line\_colour\_change(20, 144, 0, 255, delay);

line\_colour\_change(21, 144, 0, 255, delay);

line\_colour\_change(22, 255, 0, 0, delay);

line\_colour\_change(25, 255, 69, 0, delay);

line\_colour\_change(43, 144, 0, 255, delay);

line\_colour\_change(54, 255, 69, 0, delay);

}

push(); // push() and pop() combination allows me to conduct transformations on this section of code independently without effecting other sections.

background(0); //Sets the background to black.

if (bassreal < 0) { // As I had applied a subtraction to 'bass' (-150) and assigned it to 'bassreal' sometimes 'bassreal' was less than zero. I've used this if statement so that if 'bassreal' is less than zero it, it equals zero. Therefore negative numbers are removed.

bassreal = 0;

} else {

bassreal = bassreal;

}

let bass\_real\_mapped = map(bassreal, 69.1, 102.7, 50, 100, 1); //I've conducted some mappings to convert the variable(bassreal) to suitable ranges to use in my project.

let bass\_real\_mapped2 = map(bassreal, 50, 100, 0, 1, 1);

stroke(20); //This sets the colour of the border for the 'sphere' object (near black).

cam.move(0, 0, 0.5); //Sets the camera moving out of the z plane.

strokeWeight(1); // Sets the width of the border of the sphere object.

rotateY(-millis() / 1000); //Sets a rotation for the sphere object. This occurs because '-millis()' is related to the program run time and is constantly decreasing. Therefore the value of rotation is constantly changing.

sphere(bass\_real\_mapped);//Draws a sphere. Its radius is determined by the mapped value (between 50 and 100) of volume of bass in the audio 'bass'/'bassreal'.

stroke(255);//Sets intial line colour as black.

strokeWeight(30);//Sets line width.

rotateY(((bass\_real\_mapped / 60) \* millis()) / 1000);//This sets another rotation value. This one is related to 'bass\_real\_mapped' or the mapped value of the bass amplitude. Therefore when the bass amplitude is lower effected shapes/lines will rotate faster when the bass amplitude is lower they will rotate slower.

calling\_colours(30); //Recalling the 'calling\_colours' function and passing a delay time of '30' millisecond into it.

line(0, windowHeight / 4, windowWidth / 4, 0); //Drawing some line which relate to the window size.

line(0, windowHeight / 4, -windowWidth / 4, 0);

line(0, -windowHeight / 4, windowWidth / 4, 0);

line(0, -windowHeight / 4, -windowWidth / 4, 0);

pop(); //Closes the push() object. The code in between is transformed independently.

push();//Opening a new independent push pop clause.

stroke(255); //Sets intial line colour as black.

strokeWeight(10); //Sets line width.

calling\_colours(60);//recalling the 'calling\_colours' function and passing a delay time of '60' millisecond into it.

rotateY(((bass\_real\_mapped / 60) \* millis()) / 10000); //Setting a new rotation value inside this push pop clause. As with the other rotations it is in the Y plane.

line(windowWidth / 2, 0, 0, windowHeight / 2); //Drawing lines. These are effected by the rotation.

line(-windowWidth / 2, 0, 0, -windowHeight / 2);

line(windowWidth / 2, 0, 0, -windowHeight / 2);

line(-windowWidth / 2, 0, 0, windowHeight / 2);

pop();//Closes the independent push pop clause.

push(); //New rotation independent push pop clause.

stroke(255); //Sets intial line colour as black.

strokeWeight(4); //Sets line width.

calling\_colours(120); //recalling the 'calling\_colours' function and passing a delay time of '120' millisecond into it.

rotateY((-(bass\_real\_mapped / 60) \* millis()) / 1.5); //Sets a new rotation value.

//Below more lines are drawn these differ from similar previous lines of code as not only are they effected by the bass amplitude in there rotation but also there position. Cordinates are related to 'bass\_real\_mapped2', mapped bass amplitude (between 0 and 1).

line(((3 \* windowWidth) / 4) \* bass\_real\_mapped2, 0, 0, ((3 \* windowHeight) / 4) \* bass\_real\_mapped2);

line(((-3 \* windowWidth) / 4) \* bass\_real\_mapped2, 0, 0,

((-3 \* windowHeight) / 4) \* bass\_real\_mapped2);

line(((3 \* windowWidth) / 4) \* bass\_real\_mapped2, 0, 0, ((-3 \* windowHeight) / 4) \* bass\_real\_mapped2);

line(((-3 \* windowWidth) / 4) \* bass\_real\_mapped2, 0, 0, ((3 \* windowHeight) / 4) \* bass\_real\_mapped2);

pop(); //Close of independent rotation ext. clause.

let millisecond = millis(); // Assigns 'millisecond' to 'millis()' or run time.

peakDetect.update(fft); //Updates initialised 'peakDetect' in setup, identifies peaks.

peakDetect2.update(fft);

if (peakDetect.isDetected) { //If peak is detected, increase clock by one. This is a way of keeping track of the number of peaks detected.

clock = clock + 1;

}

if (clock == 1) { //Similar method that I've done previously to ensure only one time/value is assigned to a variable. If 'clock' is 1 and 'time' is 0 then 'time' equals run time and framCount/2 is assigned to 'LineTranslate\_Pre'. However, then 'time' is not equal to zero so 'time' and 'LineTranslate\_Pre' are not reassigned.

if (time == 0) {

time = millisecond;

LineTranslate\_Pre = frameCount/2;

}

}

if (clock == 10) {// This is the same format as the previous If statement but for 'clock' equalling 10, when the second bar starts.

if (time2 == 0) {

time2 = millisecond; //Run time is recorded (assigned to 'time2')

bar\_time = time2 - time; //Calculation to work out the time for one bar 'time2 - time'.

}

}

let elements = {}; //Initialises the associative array 'elements' which I use to store time and space information for specific bar starts.

elements.bar1 = { // Assigns values for space and time for 'bar1' in 'elements' associative array.

time: time + bar\_time, //Time is assigned to 'time' (run time up to start of the song) + 'bar\_time' (time from start of the music to the end of bar1).

space: Linetranslate, //Linetranslate is the frameCount/2 at the start of (elements.bar1.time).

};

elements.bar2 = { //Assigns values for space and time for 'bar2' in 'elements' associative array.

time: time + 2 \* bar\_time, //Same as for elements.bar1.time but with an additional 'bar\_time' added

space: Linetranslate + (Linetranslate - LineTranslate\_Pre), //Same as for elements.bar1.space but with '(Linetranslate - LineTranslate\_Pre)' added. This is space value for one bar.

};

//The code below up until the commented brackets all performs the same function. An if statement creates a condition so that when a certain time is past (the previous element.bar(?).time) e.g. elements.bar2.time a peakDetect triggers a clock. Once this clock hits a certain value e.g.16 (when the next bar or time value desired) run time (millisecond) is assigned to a variable e.g. 'barthree' and 'frameCount/2' (calculation for space translate) is also assigned to another variable e.g. spacer. These variables are then used to create a new addition in the associative array.

if (millisecond >= elements.bar2.time) {

if (peakDetect2.isDetected) {

clocktimer = clocktimer + 1;

if (clocktimer == 16) {

barthree = millisecond;

spacer = frameCount/2

}

}

}

elements.bar3 = {

time: barthree,

space: spacer,

};

if (millisecond >= elements.bar3.time) {

if (peakDetect2.isDetected) {

clocktimer2 = clocktimer2 + 1;

if (clocktimer2 == 17) {

barfour = millisecond;

spacer2 = frameCount/2

}

}

}

elements.bar4 = {

time: barfour,

space: spacer2,

};

if (millisecond >= elements.bar4.time) {

if (peakDetect2.isDetected) {

clocktimer4 = clocktimer4 + 1;

if (clocktimer4 == 33) {

bar\_ = millisecond;

spacer3 = frameCount/2;

}

if (clocktimer4 == 65){

bar6 = millisecond;

spacer4 = frameCount/2

}

}

}

elements.bar5 = {

time: bar\_,

space: spacer3,

};

elements.bar6 = {

time: bar6,

space: spacer4,

};

//}}

push(); //Opens a new independent push pop clause. Transformational settings are saved.

if (elements.bar1.time <= millisecond) { //This if statement provides the logic so that when the run time equals 'elements.bar1.time' 'frameCount/2' is assigned to 'Linetranslate' which is used as the 'elements.bar1.space', the amount shapes are translated by to appear full screen.

if (clock2 < 1) { // This if statement ensures that only one value is assigned to Linetranslate. This is because in setup clock2 is intialised at zero. This is less than one. The if statement runs and clock2 is increased by 1 to 1. 1 is not less than 1 so the if statement does not run again. This is important because otherwise 'Linetranslate' would keep being re-assigned new values and shapes translated with 'Linetranslate' would not stay stationary in the z plane as required.

Linetranslate = frameCount / 2;

clock2 = clock2 + 1;

}

translate(0, 0, Linetranslate); //Translate the shapes by 'Linetranslate' in the z plane.

stroke(255); //Sets intial line colour as black.

calling\_colours(300);//Recalling the 'calling\_colours' function and passing a delay time of '300' millisecond into it.

strokeWeight(6);//Sets line width.

if (treb > 40) { //Creates a translation. If the mapped 'treb'/treble amplitude is larger than 40. Then translate the shapes in this current push() pop() clause.

translate(windowWidth / 8, 0);

}

//Drawing lines.

line(

-windowWidth / 2,

-windowHeight / 2,

-windowWidth / 2,

-windowHeight / 4

);

line(

-windowWidth / 2,

windowHeight / 2,

-windowWidth / 2,

windowHeight / 4

);

if (treb > 40) {

translate((-2 \* windowWidth) / 8, 0); //This is for the opposite translation '-2' was required to counteract the previous '(windowWidth / 8, 0)' translate

}

line(

windowWidth / 2,

-windowHeight / 2,

windowWidth / 2,

-windowHeight / 4

);

line(windowWidth / 2, windowHeight / 2, windowWidth / 2, windowHeight / 4);

}

pop(); //Ends this push pop clause.

function line\_composition(time, space) { //This is the same code above converted into a function with 'time' and 'space' as parameters. The reason I had to include the above version is because it contained assignments for 'elements.bar1.space'/'Linetranslate'. Therefore this value could not be passed through as a parameter as it would not have been assigned yet.

if (time <= millisecond) {

translate(0, 0, space);

stroke(255);

calling\_colours(300);

strokeWeight(6);

if (treb > 40) {

translate(windowWidth / 8, 0);

}

line(

-windowWidth / 2,

-windowHeight / 2,

-windowWidth / 2,

-windowHeight / 4

);

line(

-windowWidth / 2,

windowHeight / 2,

-windowWidth / 2,

windowHeight / 4

);

if (treb > 40) {

translate((-2 \* windowWidth) / 8, 0);

}

line(

windowWidth / 2,

-windowHeight / 2,

windowWidth / 2,

-windowHeight / 4

);

line(

windowWidth / 2,

windowHeight / 2,

windowWidth / 2,

windowHeight / 4

);

}

}

function rotatelines(time, space, rotationcoefficient) { //This function allows the rotation of the outer lines.

if (millisecond > elements.bar4.time) { // This if statement runs once 'millisecond' (run time) is larger than 'elements.bar4.time'.

if (clockrotate < 1) {//If 'clockrotate' is less than 1 then run time 'millis()' is assigned to 'rotateVal'.

rotateVal = millis();

clockrotate = clockrotate + 1; //Clockrotate then increased by one. From 0 to 1 therefore 'clockrotate' is not less than 1. 'rotateVal' stays constant.

}

if ((millis() - rotateVal) / (elements.bar4.time - elements.bar2.time) < PI / 2) { //If statement. (millis() - rotateVal) will be zero when they (millis() - rotateVal) are equal then increase. Used PI/2 as this rotation by PI/2 is a 180 degrees rotation.

rotateZ((rotationcoefficient \* (millis() - rotateVal))/(elements.bar4.time - elements.bar2.time)); //Rotation coefficient is a parameter used to control the direction of rotation. As stated above when (millis() - rotateVal) rotation will be 0. As 'millis' (run time) increase and 'rotateVal' stays constant (millis() - rotateVal) will increase and therefore the lines will rotate. I have included (elements.bar4.time - elements.bar2.time) as the denominator to relate the rotation to the musical time. The rotation will take the time difference between (elements.bar4.time - elements.bar2.time).

line\_composition(time, space); //Calls line\_composition function (which is rotated).

} else { //If the values/variables controlling the rotation becomes equal to PI/2 then the rotation stops.

rotateZ(PI / 2);//The lines just stay rotated by (PI / 2) or 180 degrees.

line\_composition(time, space);

}

} else { // If run time is not larger than 'elements.bar4.time' than no rotation occurs 'line\_composition' function called.

line\_composition(time, space);

}

}

//Below are multiples callings of the 'rotatelines' function with different values from elements in the 'elements' associative array used as parameters. Push and pop couplings are necessary here so that the rotations of each of the 'rotatelines' instances do not effect each other. The third parameter 'rotationcoefficient' e.g. '-1' controls direction of rotation.

push();

rotatelines(elements.bar2.time, elements.bar2.space, -1);

pop();

push();

rotatelines(elements.bar3.time, elements.bar3.space, 1);

pop();

push();

rotatelines(elements.bar4.time, elements.bar4.space, -1);

pop();

//The code below is repeated from earlier in the program and is used to implement some of the more central rotating lines at a later point in the music. I attempted but couldn't do this using a function as it caused issues with the push pop clauses and rotations.

if (millis() >= elements.bar5.time) {

push();

translate(0, 0, elements.bar5.space);

stroke(255);

strokeWeight(18);

rotateY(((bass\_real\_mapped / 60) \* millis()) / 1000);

calling\_colours(30);

line(0, windowHeight / 4, windowWidth / 4, 0);

line(0, windowHeight / 4, -windowWidth / 4, 0);

line(0, -windowHeight / 4, windowWidth / 4, 0);

line(0, -windowHeight / 4, -windowWidth / 4, 0);

pop();

push();

translate(0, 0, elements.bar5.space);

stroke(255);

strokeWeight(10);

calling\_colours(60);

rotateY(((bass\_real\_mapped / 60) \* millis()) / 10000);

line(windowWidth / 2, 0, 0, windowHeight / 2);

line(-windowWidth / 2, 0, 0, -windowHeight / 2);

line(windowWidth / 2, 0, 0, -windowHeight / 2);

line(-windowWidth / 2, 0, 0, windowHeight / 2);

pop();

push();

translate(0, 0, elements.bar5.space);

stroke(255);

strokeWeight(4);

calling\_colours(120);

rotateY((-(bass\_real\_mapped / 60) \* millis()) / 1.5);

line(

((3 \* windowWidth) / 4) \* bass\_real\_mapped2,

0,

0,

((3 \* windowHeight) / 4) \* bass\_real\_mapped2

);

line(

((-3 \* windowWidth) / 4) \* bass\_real\_mapped2,

0,

0,

((-3 \* windowHeight) / 4) \* bass\_real\_mapped2

);

line(

((3 \* windowWidth) / 4) \* bass\_real\_mapped2,

0,

0,

((-3 \* windowHeight) / 4) \* bass\_real\_mapped2

);

line(

((-3 \* windowWidth) / 4) \* bass\_real\_mapped2,

0,

0,

((3 \* windowHeight) / 4) \* bass\_real\_mapped2

);

pop();

}

}