AuraSync: Plan de Développement Technique et Feuille de Route pour un Visualiseur Web 3D Audio-Réactif

Introduction

Ce document présente un plan de développement technique exhaustif et une feuille de route pour le projet "AuraSync", un visualiseur musical 3D audio-réactif de pointe. Conçu pour un étudiant en quatrième année à Epitech, ce rapport a pour objectif de structurer un projet dont l'envergure et la complexité technique justifient l'attribution de 4 crédits ECTS, correspondant à environ 104 heures de travail.

Le projet s'appuie sur une pile technologique moderne et performante, comprenant TypeScript, React (avec l'outillage Vite), la bibliothèque React Three Fiber (R3F) pour le rendu 3D déclaratif, l'API Web Audio pour l'analyse sonore en temps réel, et le langage GLSL (OpenGL Shading Language) pour la création de shaders personnalisés.

Ce rapport est structuré pour servir un double objectif : il agit comme un guide d'implémentation pratique et détaillé pour le développeur, tout en constituant un dossier technique formel, apte à être présenté pour une évaluation académique. Il est décomposé en sections qui correspondent aux fonctionnalités requises pour les différents jalons du projet, culminant avec une feuille de route journalière qui aligne les tâches de développement sur la structure de rendu par "Releases" exigée par Epitech.

Partie I : Architecture Fondamentale et Visuels de Base (Releases 1.0 & 2.0 - Le Socle de 2 Crédits)

Section 1: Échafaudage du Projet et Initialisation de la Scène 3D (Cible : Release 1.0 - 25%)

L'objectif de cette phase initiale est d'établir un environnement de développement robuste et de rendre une première scène 3D interactive. Cette fondation est critique, car elle conditionne la stabilité et l'évolutivité de l'ensemble de l'application.

Environnement de Développement avec Vite

Le choix de Vite comme outil de build est motivé par sa rapidité exceptionnelle et son écosystème moderne, qui accélèrent considérablement le cycle de développement.[2] L'initialisation du projet se fait via la commande :

```
npm create vite@latest nom-du-projet -- --template react-ts
```

Une fois le projet créé, les dépendances fondamentales doivent être installées :

- three et @types/three : La bibliothèque de rendu WebGL de base.
- @react-three/fiber : Le moteur de rendu React pour Three.js, qui permet une approche déclarative.[3]
- @react-three/drei : Une collection indispensable d'assistants, de helpers et d'abstractions prêtes à l'emploi qui augmentent drastiquement la productivité.[4]

Une structure de projet bien définie est essentielle pour la maintenabilité. La structure suivante est recommandée :

```
src/
├── assets/
├── components/      # Composants React génériques (UI, etc.)
├── glsl/            # Fichiers de shaders (.glsl)
├── hooks/           # Hooks React personnalisés (ex: useAudioAnalyzer)
├── scenes/          # Composants représentant les "Auras"
└── styles/          # Fichiers CSS
```

Composant Canvas Principal

Le point d'entrée de tout rendu R3F est le composant <Canvas>, qui doit être intégré dans le fichier principal App.tsx. Pour une expérience immersive, ce canevas doit occuper toute la fenêtre du navigateur. Ceci est réalisé en enveloppant le composant <Canvas> dans une div et en appliquant des styles CSS pour qu'elle remplisse le viewport.

CSS (index.css):

```
#root,.canvas-container {
  width: 100vw;
  height: 100vh;
  margin: 0;
  padding: 0;
}
```

React (App.tsx):

```
import { Canvas } from '@react-three/fiber';

function App() {
  return (
    <div className="canvas-container">
      <Canvas>
        {/* Les éléments de la scène 3D seront placés ici */}
      </Canvas>
    </div>
  );
}
```

Configuration de la Scène de Base

À l'intérieur du <Canvas>, les éléments fondamentaux de la scène sont ajoutés de manière déclarative. Pour commencer, une configuration minimale inclut des lumières pour éclairer les objets et des contrôles pour la navigation.

```
import { Canvas } from '@react-three/fiber';
import { OrbitControls } from '@react-three/drei';

function Scene() {
  return (
    <>
      <ambientLight intensity={0.5} />
      <pointLight position={} />
      <OrbitControls />
      <mesh>
        <boxGeometry />
        <meshStandardMaterial color="orange" />
      </mesh>
    </>
  );
}

// Dans App.tsx
// <Canvas><Scene /></Canvas>
```

L'ajout de <OrbitControls> de la bibliothèque drei est une étape cruciale pour le développement, car elle rend la scène immédiatement navigable à la souris (rotation, zoom, panoramique), ce qui facilite grandement l'inspection et le débogage.

Décision Architecturale : Le Paradigme Déclaratif

L'adoption de React Three Fiber n'est pas une simple question de syntaxe ; c'est un choix architectural fondamental. Contrairement à l'approche impérative de Three.js vanilla (où l'on exécute des commandes comme scene.add(mesh)), R3F utilise une approche déclarative. La scène 3D est décrite en JSX, devenant une fonction directe de l'état et des props de React.

Ce paradigme a une implication majeure : il crée un flux de données unidirectionnel et transparent entre la logique de l'application (gérée par l'état React) et le rendu visuel. Lorsqu'une variable d'état change (par exemple, la couleur d'un objet ou l'intensité d'un effet), React déclenche un nouveau rendu du composant, et R3F met à jour l'objet Three.js sous-jacent de manière optimisée. L'ensemble de l'application "AuraSync" sera architecturé autour de ce principe, où l'état React gouvernera tous les aspects visuels, de l'aura active à l'intensité de l'effet de lueur. Cette approche est non seulement plus élégante, mais aussi plus robuste et évolutive qu'une manipulation manuelle de la scène.

Section 2: Intégration et Analyse Audio en Temps Réel (Cible : Release 1.0 - 25%)

Cette section se concentre sur la construction d'un module audio réutilisable et performant, capable de capturer et d'analyser le son provenant de fichiers locaux ou du microphone.

Fondamentaux de la Web Audio API

L'API Web Audio est un système de traitement audio basé sur un graphe de nœuds (AudioNode). Le signal audio circule d'un nœud source (un fichier audio, un microphone) à travers une série de nœuds de traitement (analyse, gain, filtres) jusqu'à une destination (généralement les haut-parleurs).[6]

Pour ce projet, le nœud central est l'AnalyserNode. Sa fonction est d'exposer les données de fréquence et de domaine temporel de l'audio en temps réel, sans altérer le signal qui le traverse. Il agit comme une sonde non destructive sur le flux audio.[7]

Création d'un Hook useAudioAnalyzer Réutilisable

Pour garantir la modularité et la réutilisabilité, toute la logique audio sera encapsulée dans un hook React personnalisé, useAudioAnalyzer.

Ce hook gérera :

1. **L'état interne** : via useState, il suivra l'élément <audio>, l'instance de l'AnalyserNode, et l'état de la connexion.
2. **La source audio** : Il exposera des fonctions pour changer la source.
   - **Fichier local** : Une fonction gérera le changement d'un <input type="file">. Lorsqu'un fichier est sélectionné, URL.createObjectURL sera utilisé pour créer une URL locale pour le fichier, qui sera ensuite assignée à l'attribut src d'un élément <audio> caché.[8]
   - **Microphone** : Une autre fonction utilisera navigator.mediaDevices.getUserMedia({ audio: true }) pour demander l'accès au microphone. Si l'utilisateur l'accorde, le MediaStream résultant sera utilisé pour créer un MediaStreamAudioSourceNode.[9]
3. **La connexion du graphe audio** : Le hook initialisera un AudioContext et un AnalyserNode. Il connectera la source active (fichier ou micro) à l'analyseur, puis l'analyseur à la destination du contexte (context.destination). Ce chaînage est essentiel pour que l'audio soit à la fois analysé et entendu.[10]

Extraction des Données FFT

L'AnalyserNode effectue une Transformation de Fourier Rapide (FFT) pour convertir le signal temporel en un spectre de fréquences. Le résultat est accessible via la méthode getByteFrequencyData(). Cette méthode remplit un tableau (Uint8Array) avec des valeurs entières de 0 à 255, où chaque valeur représente l'amplitude (le "volume") d'une bande de fréquence spécifique, des basses aux aigus.[11]

Pour obtenir des données en temps réel, cet appel sera effectué à chaque image dans la boucle de rendu de R3F, le hook useFrame.[13]

TypeScript

```
// À l'intérieur d'un composant utilisant le hook useFrame
useFrame(() => {
  if (analyser) {
    const dataArray = new Uint8Array(analyser.frequencyBinCount);
    analyser.getByteFrequencyData(dataArray);
    // dataArray contient maintenant le spectre de fréquences actuel
  }
});
```

Raffinement des Données : Au-delà du Spectre Brut

Une visualisation directement basée sur les centaines de valeurs brutes du tableau FFT est souvent visuellement chaotique et "nerveuse". Elle ne reflète pas nécessairement la perception musicale humaine du rythme ou de la mélodie. Une étape de traitement supplémentaire est donc nécessaire pour extraire des informations plus pertinentes.

Au lieu d'utiliser le tableau brut, le hook useAudioAnalyzer effectuera une agrégation des données à chaque image. Il calculera des moyennes sur des plages de fréquences prédéfinies pour produire des indicateurs de plus haut niveau :

- bass: Moyenne des premières bandes de fréquences (ex: 0-100 Hz).
- mids: Moyenne des bandes de fréquences moyennes (ex: 400-2000 Hz).
- treble: Moyenne des bandes de fréquences hautes (ex: 5000-10000 Hz).
- volume: Moyenne de l'ensemble du spectre, représentant l'amplitude globale.

Les composants visuels de l'application réagiront à ces valeurs agrégées et lissées, et non aux données brutes. Par exemple, une pulsation visuelle sera liée à la valeur bass, tandis qu'un scintillement pourrait être lié à treble. Cette abstraction est une caractéristique clé d'un visualiseur bien conçu, car elle crée une connexion plus forte

et plus esthétique entre le son et l'image.

Partie II : Fonctionnalités Avancées et Complexité Visuelle (Releases 3.0 & 4.0 - L'Extension à 4 Crédits)

Section 3: Implémentation du Premier Visualiseur "Pulsar Grid" & Post-Processing (Cible : Release 2.0 - 50%)

Cette phase vise à construire la première scène audio-réactive complète et à l'améliorer avec des effets de post-traitement pour atteindre le "Wow Effect" mentionné dans la description du projet.

Création de la Grille avec InstancedMesh

Pour afficher une grande grille d'objets (par exemple, 400 cubes) de manière performante, il est inefficace de rendre 400 composants <mesh> distincts. La solution est le *rendu instancié*. Cette technique consiste à envoyer la géométrie de l'objet à la carte graphique (GPU) une seule fois, puis à demander au GPU de la dessiner plusieurs fois à des positions et avec des propriétés différentes. C'est beaucoup plus performant.

React Three Fiber facilite cela grâce au composant InstancedMesh de Three.js. On définit un seul InstancedMesh avec un count de 400 et on manipule ensuite les transformations de chaque instance via une matrice.

Animation Audio-Réactive

Dans la boucle useFrame, les données audio agrégées (bass, mids) du hook useAudioAnalyzer sont utilisées pour animer la grille.

1. Une boucle itère sur les 400 instances.
2. Pour chaque instance, une matrice de transformation est calculée.
3. La valeur bass est mappée à l'échelle (scale) de l'instance. Un Math.sin peut être ajouté pour créer un mouvement ondulatoire.
4. La valeur mids est mappée à la couleur émissive de la matière de l'instance.
5. La matrice de l'instance est mise à jour avec instancedMeshRef.current.setMatrixAt(i, matrix).
6. Enfin, la propriété instancedMeshRef.current.instanceMatrix.needsUpdate = true; est activée pour que Three.js envoie les nouvelles matrices au GPU.

Implémentation du Post-Processing (Effet Néon)

Le post-processing permet d'appliquer des filtres et des effets à l'image rendue finale. La bibliothèque @react-three/postprocessing s'intègre parfaitement à R3F.
Pour obtenir l'effet de lueur "néon", l'effet Bloom est utilisé.

```
import { EffectComposer, Bloom } from '@react-three/postprocessing';

function SceneWithEffects() {
  return (
    <EffectComposer>
      <PulsarGrid /> {/* Le composant de la grille */}
      <Bloom
        intensity={1.0}
        luminanceThreshold={0.1}
        luminanceSmoothing={0.2}
        mipmapBlur
      />
    </EffectComposer>
  );
```

```
}
```

L'effet Bloom fonctionne en identifiant les pixels de l'image qui dépassent un certain seuil de luminosité (luminanceThreshold), puis en les floutant et en les superposant à l'image originale, créant ainsi un halo lumineux.

Conception Holistique : Matières et Post-Processing

Le post-processing n'est pas un simple filtre appliqué à la fin ; son efficacité est intrinsèquement liée à la conception des lumières et des matières de la scène. Pour que l'effet Bloom soit saisissant, il faut que les objets de la scène émettent réellement de la lumière.

L'utilisation de la propriété emissive sur le meshStandardMaterial est donc cruciale. Une couleur émissive n'est pas affectée par les lumières de la scène ; elle représente la lumière émise par la surface de l'objet elle-même. Dans l'animation de la Pulsar Grid, ce n'est pas la propriété color qui sera principalement modifiée par l'audio, mais bien emissive et emissiveIntensity. Ainsi, lorsque la musique devient plus intense, les objets deviennent plus "brillants" d'une manière que le filtre Bloom peut détecter et amplifier, créant la pulsation néon désirée. Cette synergie entre les propriétés des matériaux et les effets de post-traitement est un concept fondamental du graphisme en temps réel.

Section 4: Expansion de l'Expérience avec Plusieurs "Auras" (Cible : Release 3.0 - 75%)

Pour atteindre les 4 crédits, le projet doit offrir une plus grande complexité visuelle. Cette section détaille la création de deux scènes supplémentaires ("Auras") et du système permettant de basculer entre elles.

Gestion d'État pour le Changement de Scène

Un gestionnaire d'état simple comme Zustand (recommandé dans l'écosystème R3F) ou l'API Context de React sera utilisé pour stocker l'état de l'activeAura. L'interface utilisateur mettra à jour cet état, et le composant de scène principal rendra conditionnellement l'Aura correspondante.

Aura 1: "Océan de Particules"

Cette scène démontrera une maîtrise plus profonde de Three.js en créant un système de particules performant à partir de zéro, sans dépendre d'une bibliothèque tierce.

- **Technique** : L'utilisation de BufferGeometry est la méthode la plus performante pour gérer des milliers de particules. Elle permet de définir manuellement tous les attributs de chaque sommet (particule), comme sa position, sa couleur, sa taille, etc..
- **Implémentation** :
  1. Créer un objet <points> contenant un <bufferGeometry>.
  2. Dans la géométrie, définir un <bufferAttribute> attaché à attributes-position. Cet attribut contiendra les coordonnées de toutes les particules.
  3. Générer les positions initiales des particules à l'aide de useMemo et d'un Float32Array. Ce tableau aura une taille de nombreDeParticules * 3. Une boucle générera des coordonnées x, y, z aléatoires pour chaque particule.Dans useFrame, animer les particules en accédant directement au tableau de l'attribut (pointsRef.current.geometry.attributes.position.array) et en modifiant les valeurs en fonction des données audio. Par exemple, la valeur bass peut créer une ondulation sur l'axe Y.
  4. Après chaque modification du tableau, il est impératif de définir pointsRef.current.geometry.attributes.position.needsUpdate = true;. Cela notifie à Three.js que les données ont changé et doivent être renvoyées au GPU.[15]

Aura 2: "Tunnel Hyperspatial Infini"

L'illusion d'un voyage infini dans un tunnel est une technique classique du graphisme en temps réel.

- **Technique** : Plutôt que de déplacer la caméra ou le tunnel, ce qui serait coûteux et complexe à rendre infini, l'illusion est créée en faisant défiler une texture le long de la surface intérieure d'un tube fixe.
- **Implémentation** :
    1. Définir la trajectoire du tunnel à l'aide d'une THREE.CatmullRomCurve3, créée à partir d'une série de THREE.Vector3.
    2. Générer la géométrie du tunnel en utilisant <tubeGeometry>, en passant la courbe en argument
    3. Créer une matière pour le tube avec une texture conçue pour se répéter sans raccord visible. Les propriétés wrapS et wrapT de la texture doivent être définies sur THREE.RepeatWrapping.
    4. Dans useFrame, animer le décalage (offset) de la texture : materialRef.current.map.offset.x += 0.01;. Cette simple ligne crée l'illusion de mouvement vers l'avant.
    5. Rendre le tunnel audio-réactif en liant le volume audio à la vitesse de défilement de la texture, et la valeur bass au rayon (radius) du tube, le faisant ainsi "respirer" au rythme de la musique.

Considérations sur la Performance : Animation CPU vs. GPU

Le choix de l'implémentation pour le système de particules a des conséquences directes sur la performance. Animer des milliers de particules sur le CPU en mettant à jour un tableau JavaScript dans useFrame (comme décrit pour l'Aura 1) peut devenir un goulot d'étranglement. Le CPU est occupé à boucler sur un grand tableau, et ces données doivent être transférées au GPU à chaque image.

Une approche plus avancée, qui sera explorée dans la section suivante, consiste à déléguer toute la logique d'animation au GPU. Cela se fait en écrivant un *vertex shader* personnalisé qui calcule la nouvelle position de chaque particule directement sur la carte graphique, en parallèle. L'implémentation de l'Aura 1 avec BufferGeometry est une étape préparatoire essentielle pour cette optimisation, car elle structure les données d'une manière que le GPU peut consommer et manipuler directement.

Démontrer cette progression d'une solution CPU vers une solution GPU est une preuve de compréhension approfondie des pipelines graphiques modernes.

Section 5: Interactivité en Direct et Shaders Personnalisés (GLSL) (Cible : Release 4.0 - 100%)

Cette dernière phase de développement intègre les fonctionnalités les plus avancées techniquement, qui sont au cœur de la justification des 4 crédits : l'entrée microphone en direct et un shader GLSL personnalisé et audio-réactif.

Entrée Microphone

Le hook useAudioAnalyzer sera étendu pour prendre en charge l'entrée du microphone. Une fonction utilisera navigator.mediaDevices.getUserMedia({ audio: true }) pour demander l'autorisation à l'utilisateur. Si elle est accordée, l'API renvoie un

MediaStream. Un MediaStreamAudioSourceNode est alors créé à partir de ce flux et connecté à l'AnalyserNode de la même manière que la source de fichier, rendant le système entièrement réactif à la voix ou à l'environnement de l'utilisateur.

Introduction à GLSL dans React Three Fiber

GLSL est un langage de bas niveau qui s'exécute directement sur le GPU, offrant un contrôle total sur le rendu. Il existe deux types principaux de shaders :

- **Vertex Shader** : S'exécute une fois pour chaque sommet (vertex) de la géométrie. Sa tâche principale est de calculer la position finale du sommet dans l'espace de l'écran (gl_Position).
- **Fragment (ou Pixel) Shader** : S'exécute une fois pour chaque pixel qui compose la géométrie. Sa tâche est de déterminer la couleur finale de ce pixel (gl_FragColor).

Pour simplifier l'intégration, le helper shaderMaterial de la bibliothèque drei sera utilisé. Il réduit considérablement le code répétitif nécessaire à la création d'un THREE.ShaderMaterial standard.

Écriture d'un Shader Personnalisé

1. **Fichiers et Uniforms** : Des fichiers vertex.glsl et fragment.glsl seront créés. Dans le composant React, le shaderMaterial sera défini avec des uniforms. Les uniforms sont des variables qui agissent comme un pont entre le code JavaScript et le code GLSL. Des uniforms comme uTime (pour le temps), uVolume et uBass (pour les données audio) seront définis.
2. **Vertex Shader** : Un shader de sommet sera écrit pour déformer la géométrie de manière audio-réactive. Par exemple, pour créer un effet d'ondulation sur une surface plane :

```
uniform float uTime;
uniform float uBass;
varying vec2 vUv; // Transmet les coordonnées UV au fragment shader

void main() {
 vUv = uv;
 vec3 newPosition = position;
 float warp = sin(position.x * 10.0 + uTime) * uBass * 0.5;
 newPosition.z += warp;
 gl_Position = projectionMatrix * modelViewMatrix * vec4(newPosition, 1.0);
}
```

   Cet exemple crée des vagues sur la surface dont l'amplitude est contrôlée par la valeur uBass.
3. **Fragment Shader** : Un shader de fragment sera écrit pour manipuler les couleurs. Par exemple, la valeur uVolume peut être utilisée pour faire varier la teinte de la couleur de base :

```
OpenGL Shading Language
uniform float uVolume;
uniform vec3 uColor;
varying vec2 vUv;

void main() {
```

```
  vec3 finalColor = uColor;
  finalColor.r += uVolume * 0.5; // Fait pulser le canal rouge avec le volume
  gl_FragColor = vec4(finalColor, 1.0);
 }
```

Passage des Données Audio au Shader

Dans la boucle useFrame, les uniforms du shader sont mis à jour à chaque image avec les dernières données. Le helper shaderMaterial de drei simplifie cette syntaxe.

```
// À l'intérieur du composant qui utilise le shader
useFrame(({ clock }) => {
  if (shaderRef.current) {
    shaderRef.current.uTime = clock.elapsedTime;
    shaderRef.current.uVolume = audioData.volume; // audioData vient de useAudioAnalyzer
    shaderRef.current.uBass = audioData.bass;
  }
});
```

Cette mise à jour constante des uniforms permet au GPU d'utiliser les dernières données audio pour calculer les déformations et les couleurs, créant un effet visuel dynamique et parfaitement synchronisé.

Section 6: Interface Utilisateur et Finitions (Cible : Release 4.0 - 100%)

La dernière étape consiste à construire une interface utilisateur (UI) complète et intuitive pour contrôler tous les paramètres du visualiseur, transformant le projet en une expérience interactive et polie.

Introduction à Leva

Leva est la bibliothèque de choix pour les panneaux de contrôle et de débogage dans l'écosystème R3F. Elle est légère, esthétique et extrêmement simple à intégrer. Elle s'installe via

npm install leva.

Création du Panneau de Contrôle

Le hook useControls de Leva permet de créer des contrôles de manière déclarative. L'UI sera structurée en dossiers pour une meilleure organisation :

- **Contrôles Globaux** : Un dossier pour les réglages principaux, avec des boutons pour sélectionner la source audio ("Fichier" / "Microphone") et pour choisir l'Aura active ("Pulsar Grid", "Océan de Particules", "Tunnel Hyperspatial").
- **Contrôles d'Effets** : Un dossier "Post-Processing" avec des sliders pour ajuster l'intensity et le luminanceThreshold de l'effet Bloom.
- **Contrôles de Shader** : Un dossier "Paramètres Shader" avec des sliders et des sélecteurs de couleur qui modifient directement les uniforms du shader personnalisé (par exemple, un slider pour l'amplitude de la déformation ou une couleur de base).

Connexion de Leva à la Scène

La beauté de Leva réside dans son intégration transparente avec l'architecture de R3F. Le hook useControls renvoie un objet contenant les valeurs actuelles des contrôles. Cet objet est réactif. Il suffit de passer ces valeurs comme props aux composants de la scène.

Exemple :

```
// Dans le composant UI
const { bloomIntensity, visible } = useControls('Bloom Effect', {
  intensity: { value: 1.0, min: 0, max: 5 },
  visible: true
});

// Dans le composant de la scène
<EffectComposer>
  <Bloom intensity={bloomIntensity} />
  <PulsarGrid visible={visible} />
</EffectComposer>
```

Lorsque l'utilisateur manipule le slider dans l'UI de Leva, la valeur de bloomIntensity est mise à jour, ce qui provoque un nouveau rendu du composant de la scène avec la nouvelle prop, et R3F met à jour l'effet. Le flux de données est direct et efficace.

L'utilisation de Leva n'est pas seulement une question d'UI. C'est un outil qui encourage une bonne architecture de composants. En définissant les contrôles, on définit en fait l'API publique d'un composant visuel, ce qui incite à créer des composants plus modulaires, réutilisables et pilotés par les props. C'est une pratique d'ingénierie logicielle mature et une compétence précieuse à démontrer.

Partie III : Gestion de Projet et Livrables

Section 7: Feuille de Route Détaillée Jour par Jour

Pour garantir une progression constante et structurée, une feuille de route sur 16 jours est proposée. Ce calendrier décompose le projet en tâches quotidiennes gérables et les aligne sur les quatre "Releases" requises par Epitech. Il sert à la fois de guide pour le développeur et de justification de la charge de travail (16 jours * ~6.5h/jour ≈ 104 heures) pour les évaluateurs.

Tableau 1: Feuille de Route du Projet AuraSync (Plan sur 16 jours / 4 Crédits)

Jour	Objectif Clé	Tâches Détaillées	Technologies / Concepts Clés	Cible Release
Semaine 1 : Les Fondations				
1	Environnement & Scène	• Initialiser projet Vite + React + TS. • Installer Three.js, R3F, Drei. • Créer <Canvas> principal, lumières, OrbitControls.	Vite, R3F (<Canvas>), Drei (<OrbitControls>)	Release 1.0
2	Module Audio (Fichier)	• Concevoir le hook useAudioAnalyzer. • Implémenter l'upload de fichier avec <input> et URL.createObje	Web Audio API, React Hooks	Release 1.0

		ctURL. • Créer AudioContext, AnalyserNode.		
3	Analyse Audio & Données	• Extraire données FFT avec getByteFrequencyData. • Agréger les données (bass, mids, etc.). • Valider les données via console.log.	AnalyserNode, useFrame	Release 1.0
4	Pulsar Grid (Instanciation)	• Créer le composant de scène pour la grille. • Utiliser InstancedMesh pour le rendu. • Lier l'échelle des instances à la donnée bass.	InstancedMesh, useFrame	Release 2.0
Semaine 2 : Premier "Wow" & Expansion Visuelle				
5	Pulsar Grid (Couleur & Poli)	• Lier la couleur émissive des instances à la donnée mids. • Affiner les fonctions de mappage pour une animation fluide.	meshStandardMaterial (emissive)	Release 2.0
6	Post-Processing (Bloom)	• Installer @react-three/postprocessing. • Intégrer	Post-processing, <Bloom />	Release 2.0

		<EffectComposer> et <Bloom />. • Configurer les paramètres de l'effet Bloom.		
7	État des Auras & Particules	• Mettre en place la gestion d'état pour le changement d'Aura. • Commencer l'Aura "Océan de Particules". • Créer <points> avec <bufferGeometry>.	Zustand/Context, BufferGeometry	Release 3.0
8	Animation Particules (CPU)	• Remplir l'attribut de position avec des données aléatoires. • Animer les positions dans useFrame via l'audio. • Appliquer needsUpdate = true.	BufferAttribute, useFrame	Release 3.0
Semaine 3 : Visuels Avancés & Shaders				
9	Tunnel Infini (Géométrie)	• Commencer l'Aura "Tunnel Hyperspatial". • Créer CatmullRomCurve3 pour la trajectoire. • Générer TubeGeometry à partir de la	CatmullRomCurve3, TubeGeometry	Release 3.0

		courbe.		
10	Tunnel Infini (Animation)	• Appliquer une texture répétable (RepeatWrapping). • Animer texture.offset dans useFrame. • Lier le volume audio à la vitesse d'animation.	Animation de texture.offset	Release 3.0
11	Entrée Microphone	• Étendre useAudioAnalyzer pour le microphone. • Utiliser getUserMedia et MediaStreamAudioSourceNode.	navigator.mediaDevices.getUserMedia	Release 4.0
12	Configuration Shader GLSL	• Créer les fichiers .glsl. • Utiliser drei/shaderMaterial pour la matière custom. • Définir les uniforms (uTime, uBass, etc.).	shaderMaterial, GLSL, Uniforms	Release 4.0
Semaine 4 : UI & Finalisation				
13	Implémentation Shader GLSL	• Écrire le vertex shader (déplacement). • Écrire le fragment shader (couleur). • Passer les données audio aux uniforms	GLSL (sin, mix), Uniforms	Release 4.0

		dans useFrame.		
14	Implémentation UI (Leva)	• Installer leva. • Créer des useControls pour tous les paramètres. • Structurer l'UI en dossiers.	leva, useControls	Release 4.0
15	Intégration & Débogage	• Connecter tous les contrôles Leva aux props de la scène. • Tester exhaustivement toutes les fonctionnalités. • Corriger les bugs visuels et optimiser la performance.	-	Release 4.0
16	Documentation & Livraison	• Rédiger le README.md (installation, utilisation). • Enregistrer une vidéo de démonstration. • Préparer les documents de soumission Epitech.1	-	Release 4.0

V2:

Functionality 1 : Initialisation Technique & Pipeline Audio de Base (Est. 4 jours)

Details : Cette phase initiale est centrée sur la mise en place d'un socle technique robuste pour l'application : création du projet, structure modulaire, base 3D interactive via React Three Fiber, et configuration du pipeline d'analyse audio (fichier local). Elle vise à dérisquer les points critiques : compatibilité des API audio avec le moteur de rendu, gestion asynchrone des ressources, et intégration propre du flux audio en React.

Tasks :

1. Initialiser un projet Vite + React + TypeScript avec les dépendances : `three`, `@react-three/fiber`, `@react-three/drei`, `leva`.

2. Créer une structure de répertoires claire : `/components`, `/hooks`, `/scenes`, `/glsl`, `/assets`, `/styles`.

3. Intégrer un `<Canvas>` R3F plein écran avec `OrbitControls` pour rendre la scène navigable dès le départ.

4. Créer un hook `useAudioAnalyzer` encapsulant la Web Audio API, capable de :

    - Lire un fichier audio via un `<input type="file" />`

    - Créer un `AudioContext` et un `AnalyserNode` correctement chaînés au graphe audio

    - Afficher les données FFT dans la console (`getByteFrequencyData`) à chaque frame via `useFrame`.

5. Vérification du système : affichage d'un cube dont l'échelle est reliée au volume global → preuve que le pipeline audio réagit correctement.

- **Functionality 2 : Visualisation 3D Instanciée & Mappage Fréquentiel (Est. 3 jours)**

Details : Cette étape établit la première visualisation audio-réactive réaliste. L'enjeu est de relier les données du spectre à une animation performante dans un environnement 3D complexe (grille instanciée). La logique audio est raffinée pour produire des indicateurs de haut niveau (bass, mids, treble, volume).

Tasks :

1. Étendre `useAudioAnalyzer` pour calculer dynamiquement : `bass`, `mids`, `treble`, `volume` via agrégation sur des plages de fréquences FFT.
2. Créer un composant `PulsarGrid` basé sur `InstancedMesh` (400 cubes).
3. Mapper les bandes fréquentielles à :
    - `bass` → scaleY des cubes (respiration rythmée)
    - `mids` → couleur `emissive` pour créer un effet d'éclat
    - `treble` → légère rotation individuelle des instances (effet de finesse)
4. Implémenter une fonction de mapping non linéaire (`easeOutQuad`, clamp) pour lisser l'effet.
5. Ajouter `instanceMatrix.needsUpdate = true` pour forcer le redraw GPU.

● Functionality 3 : Multiples Auras, Gestion d'État & Scènes Dynamiques (Est. 3 jours)

Details : Cette phase introduit la notion d'"Auras", scènes visuelles distinctes manipulables dynamiquement. Elle démontre la capacité à structurer l'application autour d'un état global, avec un pipeline de visualisation modulaire. Cela permet d'explorer différentes techniques de rendu (particules, textures animées, etc.).

Tasks :

1. Intégrer Zustand ou Context API pour gérer `activeAura`.
2. Créer l'Aura 1 : **Océan de Particules**
    - Utilise `<Points>` avec `BufferGeometry` et `Float32Array`

- Animation par mise à jour directe du buffer `.array[]` dans `useFrame`
- Basée sur `bass` (Y position) et `volume` (color jitter)

3. Créer l'Aura 2 : **Tunnel Hyperspatial**
   - Construction d'un `TubeGeometry` autour d'une `CatmullRomCurve3`
   - Application d'une texture animée via `map.offset.x += delta`
   - Le `volume` contrôle la vitesse du déplacement, `bass` modifie le rayon du tube
4. Permettre le switch d'une Aura à une autre via GUI ou touches clavier.

● Functionality 4 : GLSL Shaders Audio-Réactifs & Microphone (Est. 3 jours)

Details : Cette phase introduit les shaders personnalisés (GLSL) et le support de l'entrée audio en direct. Elle valide la maîtrise du pipeline GPU moderne et de la synchronisation des données audio → GPU via `uniforms`. L'utilisateur peut choisir entre micro et fichier comme source.

Tasks :

1. Étendre `useAudioAnalyzer` avec `navigator.mediaDevices.getUserMedia` pour gérer les flux micro (via `MediaStreamAudioSourceNode`).
2. Créer un shaderMaterial (`@react-three/drei`) utilisant `vertexShader` et `fragmentShader` GLSL.
3. Écrire un **vertex shader** qui déforme une surface plane selon :
   - `uTime` : animation temporelle sinusoïdale
   - `uBass` : amplitude des vagues
4. Écrire un **fragment shader** qui module la teinte de la couleur de base avec `uVolume`.
5. Mettre à jour les `uniforms` à chaque frame dans `useFrame` : `shaderRef.current.uniforms.uBass.value = data.bass`.

● Functionality 5 : Interface Utilisateur Réglable, Bloom & Finalisation (Est. 3 jours)

Details : L'objectif ici est de transformer le prototype en une application finie : intégration de `leva` pour piloter tous les paramètres en temps réel, ajout d'effets de post-processing (notamment Bloom) et stabilisation de l'expérience utilisateur.

Tasks :

1. Intégrer `leva` et créer une UI complète :
    - Contrôle de la source audio (file / microphone)
    - Choix de l'Aura active
    - Réglage des shaders : `amplitude`, `color`, etc.
    - Réglage du Bloom : `intensity`, `threshold`, `smoothing`
2. Ajouter `@react-three/postprocessing` avec `EffectComposer` et `Bloom`.
3. Connecter tous les contrôles `leva` aux composants et shaders via props ou `uniforms`.
4. Nettoyage du code, documentation, tests, et production d'une vidéo démo.

Delivery

- **Screenshots (obligatoire)** :
    - Scène PulsarGrid (Release 2.0)
    - Scène Particules et Tunnel (Release 3.0)
    - Interface Leva avec réglages (Release 4.0)
    - Shaders + effet Bloom actif (Release 4.0)
- **README complet (obligatoire)** :
    - Instructions de build (`npm install && npm run dev`)
    - Présentation technique du projet
    - Liste des fonctionnalités et stack utilisée
- **Vidéo de démonstration (optionnelle)** :
    - 1 à 2 minutes avec switching d'Auras, visualisation audio, interaction UI
- **Dépôt Git (optionnel)** :
    - Code open-source en ligne, prêt à être cloné
    - Branches pour chaque Release disponible

○

Conclusion

Le plan de développement pour le projet "AuraSync" décrit une application web 3D riche, complexe et techniquement ambitieuse. En suivant la feuille de route proposée, le projet couvrira non seulement les fonctionnalités de base d'un visualiseur audio, mais explorera également des concepts avancés qui sont au cœur du développement graphique moderne en temps réel.

La portée du projet justifie amplement les 4 crédits ECTS visés. L'architecture modulaire basée sur les hooks React, l'utilisation performante de l'instanciation et des BufferGeometries, la création de multiples scènes visuellement distinctes, et surtout, l'implémentation d'un shader GLSL personnalisé piloté par des données audio en direct, constituent un ensemble de réalisations techniques significatives.

Au-delà de la validation académique, la réalisation de ce projet permettra d'acquérir une maîtrise approfondie de la pile technologique de React Three Fiber, ainsi qu'une compréhension pratique des principes d'optimisation GPU et de la synergie entre l'analyse audio et le rendu visuel. "AuraSync" représente une excellente opportunité de créer une pièce de portfolio impressionnante et de développer des compétences très recherchées dans le domaine du développement web interactif.