

Precision Parking — Mini-jeu en p5.js

1. Présentation générale

Ce projet consiste à développer un **mini-jeu de stationnement de voiture** en utilisant la bibliothèque **p5.js**.

Le joueur doit contrôler une voiture et la positionner correctement dans une place de parking, tout en évitant des obstacles et en respectant des contraintes de temps et de précision.

L'objectif principal du projet est de mettre en pratique les notions de **programmation interactive**, de **gestion des collisions**, de **logique de jeu** et de **structuration du code** en JavaScript.

2. Technologies utilisées

- JavaScript
 - p5.js (dessin, animation, gestion des entrées clavier)
 - p5.sound.js (sons et musique)
 - Ressources graphiques simples (image de voiture)
 - Fichiers audio pour les effets sonores
-

3. Principe de fonctionnement du jeu

Le joueur contrôle la voiture à l'aide des flèches du clavier :

- flèche haut : accélération
- flèche bas : frein / marche arrière
- flèche gauche et droite : rotation

Le but est d'atteindre la place de parking :

- sans entrer en collision avec les obstacles,
- avec une vitesse faible,
- dans un temps limité.

Un système de score et d'étoiles permet d'évaluer la performance du joueur.

4. Organisation du projet

La structure du projet est organisée de la manière suivante :

```
/project
├── index.html
├── sketch.js
├── vehicle.js
├── obstacle.js
├── /assets
│   ├── car.png
│   ├── music.mp3
│   ├── success.wav
│   └── crash.wav
```

Chaque fichier JavaScript a un rôle précis afin de rendre le code plus lisible et plus facile à maintenir.

5. Gestion de la voiture

La logique de la voiture est implémentée dans le fichier `vehicle.js`.

Fonctionnalités principales :

- gestion de la position, de la vitesse et de l'accélération,
 - application d'une friction pour ralentir naturellement la voiture,
 - rotation progressive pour un contrôle plus réaliste,
 - affichage de la voiture à l'aide d'une image,
 - ajout d'une ombre pour améliorer la lisibilité de la position et de la direction.
-

6. Gestion des obstacles

Les obstacles sont gérés dans le fichier `obstacle.js`.

Deux types d'obstacles sont utilisés :

- obstacles rectangulaires (murs),
- obstacles circulaires (plots / cônes).

Chaque obstacle :

- est affiché graphiquement,
 - possède une détection de collision avec la voiture,
 - permet de calculer la distance entre la voiture et l'obstacle le plus proche.
-

7. Détection de proximité

Un système d'alerte de proximité a été mis en place :

- lorsque la voiture s'approche trop près d'un obstacle, un message d'avertissement est affiché,
- la distance exacte entre la voiture et l'obstacle le plus proche est indiquée,
- cela permet au joueur d'anticiper les collisions et d'améliorer sa trajectoire.

Ce système améliore l'aspect pédagogique du jeu en aidant le joueur à comprendre ses erreurs.

8. Gestion du temps

Chaque niveau dispose d'un temps limite :

- un chronomètre est affiché en haut de l'écran,
- si le temps est écoulé avant le stationnement réussi, le niveau est considéré comme échoué.

Cette contrainte oblige le joueur à trouver un compromis entre vitesse et précision.

9. Système de score et d'étoiles

À la fin de chaque niveau réussi :

- le joueur reçoit entre une et trois étoiles,
- le nombre d'étoiles dépend du temps restant,
- le score total est calculé à partir des étoiles obtenues.

Ce système permet de motiver le joueur à rejouer les niveaux pour améliorer son résultat.

10. Sons et musique

Le jeu utilise plusieurs éléments sonores :

- une musique de fond,
- un son de réussite lorsque le parking est réussi,
- un son d'échec en cas de collision ou de dépassement du temps.

Pour respecter les règles des navigateurs, les sons sont lancés uniquement après une interaction utilisateur (clic ou touche clavier depuis le menu).

11. Description des niveaux

Niveau 1 : prise en main

Niveau simple avec peu d'obstacles, permettant au joueur de comprendre les contrôles et le comportement de la voiture.

Niveau 2 : slalom vertical

Présence d'une ligne verticale de plots remplissant l'écran du haut vers le bas.
Ce niveau demande plus de précision et un meilleur contrôle de la trajectoire.

Niveau 3 : slalom multiple avancé

Plusieurs lignes verticales de plots sont générées avec des décalages alternés vers le haut et le bas.

Une légère part d'aléatoire est introduite afin de rendre chaque partie différente et plus difficile.

12. Méthodologie de réalisation

Le projet a été développé progressivement :

1. déplacement basique de la voiture,
2. ajout des collisions,
3. intégration de la place de parking,
4. gestion du temps,
5. ajout des obstacles,
6. ajustement des distances de détection,
7. ajout du système de proximité,
8. intégration des sons,
9. équilibrage des niveaux.

Chaque étape a été testée avant de passer à la suivante.

13. Objectifs pédagogiques

Ce projet permet de consolider :

- la compréhension de la boucle de rendu de p5.js,
- l'utilisation des vecteurs pour le mouvement,
- la gestion des collisions simples,
- l'organisation du code en classes,
- les bases du game design (difficulté, feedback, progression).

14. Lancement du projet

Pour exécuter le projet :

1. ouvrir le dossier du projet,
2. lancer le fichier `index.html` dans un navigateur,
3. cliquer sur le bouton de démarrage,
4. utiliser les flèches du clavier pour jouer.

15. Conclusion

Ce mini-jeu montre qu'il est possible de réaliser un projet interactif complet avec p5.js tout en restant dans un cadre académique.

Il combine logique, interaction utilisateur et organisation du code, et constitue une base solide pour des projets plus avancés.

Formules et logiques utilisées dans le jeu Precision Parking

Cette section décrit les principales **formules mathématiques** et **logiques algorithmiques** utilisées pour la gestion du mouvement, des obstacles, des collisions, de la proximité et du stationnement.

1. Modélisation du mouvement de la voiture

La voiture est modélisée à l'aide de **vecteurs** :

- Position : **pos**
- Vitesse : **vel**
- Accélération : **acc**

Mise à jour du mouvement

À chaque frame, le mouvement est calculé selon les étapes suivantes :

1. Ajout de l'accélération à la vitesse :

`vel = vel + acc`

2. Limitation de la vitesse maximale :

$|vel| \leq maxSpeed$

3. Application d'une friction pour simuler les frottements :

$vel = vel \times friction$

4. Mise à jour de la position :

$pos = pos + vel$

5. Réinitialisation de l'accélération :

$acc = 0$

Cette logique permet un mouvement fluide et progressif, évitant des changements brusques de direction ou de vitesse.

2. Calcul de la direction et de la rotation

La direction de la voiture est définie par un **angle** (en radians).

Lors d'une accélération, un vecteur directionnel est calculé à partir de l'angle :

$direction = (\cos(angle), \sin(angle))$

La force appliquée à la voiture est alors :

$force = direction \times puissance$

La rotation est contrôlée par :

$angle = angle + (directionRotation \times coefficient)$

Cela permet à la voiture de tourner progressivement, comme un véhicule réel.

3. Détection de collision avec les obstacles

3.1 Collision voiture – obstacle circulaire (plots)

Les plots sont modélisés comme des **cercles**.

La collision est détectée par la distance entre le centre de la voiture et le centre du plot :

$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Collision si :

`distance < rayonVoiture + rayonObstacle`

3.2 Collision voiture – obstacle rectangulaire (murs)

Les murs sont modélisés comme des **rectangles**.

La méthode utilisée consiste à :

1. Trouver le point du rectangle le plus proche du centre de la voiture
2. Calculer la distance entre ce point et la voiture

Collision si :

`distance < rayonVoiture`

Cette méthode est robuste et largement utilisée dans les jeux 2D.

4. Calcul de la distance de proximité avec les obstacles

Pour chaque obstacle, on calcule la distance minimale entre la voiture et l'obstacle.

Pour un obstacle circulaire :

`distanceProximité = distanceCentre - (rayonVoiture + rayonObstacle)`

La distance minimale parmi tous les obstacles est conservée :

`distanceMin = min(distanceProximité1, distanceProximité2, ...)`

Cette valeur est utilisée pour :

- afficher un avertissement visuel,
 - informer le joueur de sa proximité avec un obstacle,
 - ajuster dynamiquement certains effets (sons, alertes).
-

5. Zone de danger et seuil de proximité

Un seuil de danger est défini :

`dangerDistance = constante (ex: 25 pixels)`

Si :

`distanceMin < dangerDistance`

Alors :

- un message d'avertissement est affiché,
- une animation visuelle est activée,
- le joueur est informé qu'il est trop proche d'un obstacle.

6. Logique de stationnement (parking réussi)

Le stationnement est validé uniquement si **toutes les conditions suivantes sont respectées**.

6.1 Transformation dans le repère du parking

La position de la voiture est convertie dans le repère local de la place de parking :

```
localPos = positionVoiture - positionParking  
localPos = rotation(-angleParking) × localPos
```

Cela permet de vérifier correctement l'alignement, même pour une place verticale.

6.2 Condition de position

La voiture doit être entièrement à l'intérieur de la place :

```
|localPos.x| < largeurParking / 2 - marge  
|localPos.y| < hauteurParking / 2 - marge
```

6.3 Condition de vitesse

La vitesse doit être suffisamment faible pour éviter un stationnement violent :

```
|vel| < vitesseMaxAutorisee
```

6.4 Condition d'orientation (niveau 3)

Pour les places verticales, l'angle de la voiture doit être proche de $\pm 90^\circ$:

```
|angleVoiture -  $\pi/2$ | < tolérance  
ou  
|angleVoiture +  $\pi/2$ | < tolérance
```

6.5 Validation finale

Le parking est réussi si :

`positionOK ET vitesseOK ET orientationOK`

7. Gestion du temps

Un chronomètre est utilisé pour chaque niveau :

`tempsRestant = tempsLimite - (tempsActuel - tempsDébut)`

Si :

`tempsRestant ≤ 0`

Alors :

- le niveau est échoué,
 - la partie est bloquée jusqu'à redémarrage.
-

8. Calcul du score et des étoiles

Le nombre d'étoiles dépend du temps restant :

- 3 étoiles si :

`tempsRestant > 60 % du tempsLimite`

- 2 étoiles si :

`tempsRestant > 30 % du tempsLimite`

- 1 étoile sinon

Le score total est calculé par :

`score = score + (étoiles × 100)`

9. Logique de génération des obstacles (slalom)

Les obstacles sont générés automatiquement à l'aide de boucles.

Slalom vertical

Pour une ligne verticale :

`y = yDépart ; y ≤ hauteurÉcran ; y += espaceVertical`

Slalom multiple

Pour plusieurs lignes :

```
x = xBase + indexLigne * espacementLignes  
y = yBase + décalageVertical + indexObstacle * espacementVertical
```

Des décalages différents sont utilisés pour rendre le parcours plus complexe.

10. Apport pédagogique

L'ensemble de ces formules permet de :

- comprendre la gestion du mouvement vectoriel,
- appliquer des calculs de distance,
- maîtriser les collisions simples,
- structurer une logique de jeu complète,
- relier mathématiques et programmation interactive.