

# Compte Rendu Informatique Graphique

Nicolas WALLET  
L3 Informatique

Mai 2020

## Table des matières

<b>1</b>	<b>Three JS</b>	<b>3</b>
1.1	Segement . . . . .	3
1.1.1	Ligne droite . . . . .	4
1.1.2	Virage . . . . .	4
1.1.3	Bezier . . . . .	4
1.1.4	Boost . . . . .	4
1.1.5	Points de contrôle . . . . .	4
1.2	Carte . . . . .	4
1.3	Terrain . . . . .	5
1.4	Une Voiture . . . . .	5
<b>2</b>	<b>Environnement</b>	<b>6</b>
2.1	Lampadaire . . . . .	6
2.2	Arbres et Buissons . . . . .	6
2.3	Forêt . . . . .	6
<b>3</b>	<b>Login et GUI</b>	<b>7</b>
3.1	Page de connexion . . . . .	7
3.2	Statistiques . . . . .	7
3.3	Contrôles . . . . .	7
<b>4</b>	<b>Server</b>	<b>8</b>
4.1	Cacher les joueurs . . . . .	8
<b>5</b>	<b>Amélioration</b>	<b>8</b>
<b>6</b>	<b>Sources</b>	<b>9</b>
<b>A</b>	<b>Annexe</b>	<b>10</b>

## Abstract

Les fichiers ci-après évoqués sont disponible sur le lien Git suivant : [Github.com](#) Il suffit de cloner le dépôt.

Une version en ligne est aussi disponible. ([ici](#) )

Par convention les noms des variables et fonctions sont en anglais.

Si vous le souhaitez, vous pouvez voir les éléments au fur et à mesure, un fois node lancé, en bas de la page dans la section "Notes" , cliquez sur "Demonstration des composants".

## Introduction

Plusieurs objectif pour notre petit jeu : Cote Client Avoir une

- Une voiture qui gère une pyshique de base ( accélération, freinage )
- Un route (délimitant les limites)
- Réapritation au dernier point de controle (respawn / checkpoint )
- Génération de carte

Interface graphique (GUI)

- Nombre de tours
- Temps du dernier tour
- Temps au meilleur tour
- Vitesse actuelle
- (M) Liste des joueurs

## 1 Three JS

Pour l'ensemble des éléments j'ai choisi un style 'low poly', c'est-à-dire avec peu de faces, c'est simple à réaliser, ça garantit certaines performances. Donc, de manière générale, j'utilise du flatShading. Pour les ombres, les lumières et éléments au-dessus du sol peuvent castShadow et les autres éléments reveive-Shadow.

### 1.1 Segement

On pourrai croire que l'élément a réaliser en premier est le joueur ( ie : notre voirure ) mais vu qu'on veut restreindre dans la plupart des cas le joueur à la route , le mieux est de créer un outils pour créer une route sans trop se prendre la tête, en plus on pourra faire plusieurs circuits différents sans trop de difficulté.

Une 'Map' c'est quoi ?

Une route definit par des coordonnée, on va definir plusieurs type de voies ( Pour le moment route ) Plusieurs type de formes 'shape' ( ligne droite , intesertion , courbe ) Des checkpoints pour verifier que le joueur passe au bon endroit, mais surtout pour respawn.

Le visuels 'route' sera le meme pour chaques type de route. Une bordure couleur beton. Un sol couleur Bitume , et une ligne de signalisation blanche au milieu de la route. On utilisera des PhongMaterial de couleur.

Toute nos route seront de la même largeur pour permettre une unifomrité.

### 1.1.1 Ligne droite

Notre premier segment est une ligne droite (Linear). C'est le plus simple. Le segment est définie avec une longueur (length). On commence avec un plan pour notre sol. Deux geometryBox pour les cotes et la ligne droite sera un plan.

### 1.1.2 Virage

C'est le plus compliqué, on pourrait utiliser RingGeometry pour le sol, mais on souhaite garder nos bordures sur les côtes, or aucune géométrie ne correspond. Les seuls qui correspondent un peu sont TORus et Tube. Mais j'ai aussi la possibilité d'utiliser Shape, qui est beaucoup plus flexible. Et shape fonctionne un peu comme l'API Canvas 2D. Je peux donc utiliser exactement les mêmes formes pour le sol ou les cotes.

Voir figure (1)

### 1.1.3 Bezier

L'idée c'est d'avoir des lignes droites avec un décalage, voir Figure 2 Annexe A. On reprend le code précédent, mais maintenant on a deux points intermédiaires. Notre segment bezier est défini par une longueur et un décalage (positif vers la droite et négatif vers la gauche)

Voir figure (2)

```
Shape.bezierCurveTo(x1,y1,x2,y2,x3,y3)
```

### 1.1.4 Boost

Le boost est une ligne droite. Elle a deux particularités, la première, visuelle, on utilise les vertexcolor pour mettre certaines faces en jaune. La deuxième, c'est qu'on doit se souvenir que ce segment en particulier permet au joueur d'avancer plus vite. Plus tard, on permettra aux joueurs de connaître le numéro du segment sur lequel il route, j'ajoute donc à une variable globale le numéro de segment boost.

### 1.1.5 Points de contrôle

Mes points de contrôle sont surtout des segments utilitaires. La gestion de sa fonction se fait dans la partie suivante. Niveau visuel, c'est une simple route, mais avec une petite arche pour permettre de différencier ce segment des autres.

## 1.2 Carte

On a pas mal de composants nécessaires à la création d'un circuit. On va créer une classe qui va permettre de gérer notre circuit.

```
Class Carte{...}
```

Évidemment chaque instance de 'Carte' sera un monde à part entière. Une carte sera composée de Segments. Ma 'Carte' doit faire en sorte de placer chaque segment de manière à ce qu'ils forment un circuit (ie : sur j'ajoute deux segments linéaires, un premier de taille A et le second de taille B, ma carte commence

en (0,0) on obtient  $\text{segment1}(0,0)$  et  $\text{segment2}(0,A)$ ) Ici, c'est assez simple à gérer, mais ma 'Carte' doit gérer tous mes composants. Ceci implique de calculer le point après un virage, et de changer l'orientation du segment suivant.

Concentrons nous sur nos virages, à partir d'un point (0,0) et d'une taille *SIZE*, d'un côté *SIDE*, et d'une orientation *FACING*. Comment définir le points suivant ? (cf fichier [geogebra.html](#))

$$\begin{aligned} A(0,0) \\ F = (FACING + SIDE)90\pi * /180 \\ B(SIZE * (\cos(F) + \sin(F)), SIZE * (\cos(F) - \sin(F))) \end{aligned}$$

Il y a deux cote ( droite ou gauche ) et 4 orientation .C'est donc 8 cas mais seulement 4 points

### 1.3 Terrain

Terrain low poly. Je pars d'un plan. Je vais simplement modifier les positions des vertices en y et les couleurs des faces (et donc le material devra utiliser les `vertexColors`). Quand je génère ma carte, je définis des AABB box à partir des segments. (Ce sont les [Box3](#)). Je commence par modifier les sommets. Si un sommet se trouve sur un des segment alors je n'y touche pas, si en revanche, il est en dehors et change son y. J'ai joué un peu avec les paramètres pour trouver quelques choses de correct. Pour générer les montagnes, je calcule le sommet 'y' qui est en dehors de tout segment AABB en fonction de sa distance minimale avec les segments. J'ajoute des facteurs d'atténuations et d'aléatoires.

De la même manière, je veux sur certains terrains ajouter des lacs, qui sont des montagnes « inversé » beaucoup plus atténuer. Je passe en paramètre une `Box3` et si la vertice et dans ce point, on applique les positions pour créer un lac. Pour le lac, je rajoute un plan bleu translucide à hauteur -1 ( ma carte est à 0).

Edit : J'ai essayé de faire un segment tunnel, je me suis dit qu'en ne l'ajoutant pas le terrain pourrai générer de la montagne. (Voir annexe). Le problème est qu'on voit certaine face aux entrées du tunnel lorsqu'il y a une montagne. J'ai pensé au clipping, mais au mieux, c'est avec des plans locaux donc pas simple à gérer. Ou j'avais aussi pensé à retirer les faces qui sont à l'intérieur des tunnels, mais ça pose des problème de repositionnement des sommets adjacents. Voir image 3

### 1.4 Une Voiture

Pour la voiture, un modele libre de droit fera l'affaire. Le modèle est composé d'un `.obj` ( geometry ) et un `.mtl` ( material ). Les deux fichiers sont liés, ils doivent être chargés en même temps. ( mtl puis obj ) Il y a des exemples dans la documentation pour tous les loaders, mais pas de mtl et à la fois obj, j'en ai trouvé un ici : [Hofk.de](#)

Pour le déplacement de la voiture, on accède aux positions et à la rotation via l'objet (de la scène) et on déclare une variable avec vitesse, accélération, vitesse max, etc. En fonction des contrôles du joueur, on change les valeurs de la variable, vitesse ou rotation, puis on calcule la nouvelle position en fonction de la variable.

Pour restreindra les déplacements à la carte , j'avais dans un premier temps utiliser les Box3, mais pour les bloc curve et bezier, c'est assez gênants. Après quelques recherches, il semblerait qu'une autre méthode soit plus efficace dans ce cas : raytracing (ici raycaster) . On définit une liste de mesh qui avec laquelle on cherche une collision. On créer une THREE.Raycaster() qu'on définit avec un point d'origine à partir de notre voiture. À partir d'une collision, on connaît un vecteur qui va nous permettre de connaître l'angle de collision.

Une fois qu'il y a collision je regarde la distance avec la collision précédente :

```
* Je garde la distance avec la collision calculé précédente
* Si l'ancienne est plus grand c'est mon joueur essaye de quitter la collision
* càd il s'eloigne , alors je le laisse repartir
* sinon je l'arrete
*
if(old>collisionResults[0].distance)
    car_stats.speed=0
old=collisionResults[0].distance
```

## 2 Environnement

### 2.1 Lampadaire

Proche de nos route

Pour la lumière, un THREE.Spotlight avec un angle réduit, une distance légèrement supérieur au lampadaires , et en fonction de l'intensité désiré on ajuste de le decay.

### 2.2 Arbres et Buissons

Vu qu'on veut faire du low poly, on peut utiliser des geometrie très simple fourni pas THREE.JS. On ecrit donc une fonction qui pose un arbre en x,y,z fourni en paramètre, notre arbres est un objet3D composé d'un tronc et de feuilles. Le tronc sera un cylindre marron avec un rayon sup < rayon inf. Les feuilles seront un dodécaèdre car sphere faisait trop boule même avec peu de segments.

En instanciant plusieurs arbres à la suite je me suis aperçu qu'on voyait très rapidement qu'il etait tous identiques. J'ai donc ajouter des variations pour que ça soit plus jolie. Plusieurs couleur pour les feuilles , dont j'applique aussi une rotation Math.random(). Dans la même idée je fais une souche d'arbre en tant que variation .

Les buissons seront des dodécaèdre avec les variations de couleur et de rotation. Et le nombre de buisson varie aussi.

### 2.3 Forêt

J'en profite pour faire un fonction 'forêt', qui génère un nombre important d'arbres et de buissons.

Vous pouvez voir les différents resultats dans 'Demonstration des composants' > 'Environnement' > Arbres, buissons et forêts.

## 3 Login et GUI

### 3.1 Page de connexion

Cette page permet rentrer un pseudo et de choisir la couleur de sa voiture.

```
document.querySelector('#log').addEventListener('submit', event => {  
  // On initialise le rendu THREE JS  
  init()  
  animate()  
  
  // Pour socket.io voir Chapitre 2  
  socket=io()  
  //...  
})
```

### 3.2 Statistiques

Comme définit en introduction nous voulons connaître notre temps au meilleur tour et dernier tour. On crée une `<div>` qu'on place dans un coin ( haut-gauche ). On utilisera un système de checkpoints pour s'assurer que le joueur suit la bonne route ( ie : dans le bon sens ) et on prendra le temps `THREE.clock`.

Pour le temps, toutes les méthodes sont définies dans la doc ( [Clock](#) )

Imaginons notre circuit comme suit : Il nous faut impérativement ( pour cette solution ) un minimum de 3 checkpoints pour valider le tour. Cette méthode s'applique dans le cas d'un circuit ( ei : départ = arrivé , et départ = checkpoints )

A chaque fois que le joueur passe un cp on le met à 1 . Si le joueur est au départ qu'il recule au dernier cp alors on ne doit pas le mettre à 1 ( car le second cp est encore à 0 ).

Si tous les cp sont à l'état 1 et que nous sommes au départ alors on a fait un tour, on peut rester la variable et on recommence si le nombre de tour est différent de celui défini par la carte.

### 3.3 Contrôles

Si l'utilisateur ne souhaite pas utiliser les touches par défaut il doit pouvoir les changer. On va passer par une variable qui contient les touches associées pour chaque opération.

```
KeyBind = {  
  avancer : 'z',  
  camera : 'a',  
}  
  
//mov est notre tableau de touches qui sont 'keydown'  
  
if( mov[ KeyBind.avancer ] ) //equivaut a if ( mov['z'] )  
  //avancer la voiture
```

Quand notre joueur change de touche, on actualise notre objet `KeyBind`. En pratique on va garder `event.key` et `event.code` dans `keybind`. Le code pour

les conditions et key pour l’affichage ( parce que code ne dépend pas du clavier EN/FR - alors que key oui). Si l’utilisateur change de carte / refresh il perd sont keyBinding. Pour bien faire , l’idéal serait d’avoir des comptes utilisateurs pour stocker le keyBinding ou d’utiliser le localStorage.

## 4 Server

Le principe est de mettre le jeu en multijoueur, pour un peu de compétition. Chacun peu rejoindre une carte quand il le souhaite.

Comme cette partie n’est pas demandé et ce n’est pas du THREE.JS je vais expliquer seulement le principe. J’utilise socket.io() pour la connexion client-server.

Nous avons plusieurs carte donc si un joueur se connecte sur la carte 1 il ne doit pas voir un joueur sur la carte 2. Socket.io propose deux solutions pour séparer les connexions , les Namespace qui créent un channel de connexions pour chaque namespace et les rooms qui permettent de diviser les namespaces. Les rooms sont plus adaptés à nos ‘serveurs’. Lorsqu’un joueur lance une carte il se connecte au server ( socket.io ) en lui donnant le numéro de la carte qu’il veut se connecter. Le server l’ajoute alors à la room. Le server envoie la liste des joueurs ayant déjà rejoint la room ( et le client les affichera ).

Lorsque qu’un client se déplace il envoie sa position au server qui la distribue aux autres joueurs de la room ( ‘emit’ envoie ajouter les joueurs et ‘broadcast’ l’envoi à tous sauf la personne qui l’a envoyé , et to(X) permet de l’envoi à X, X peut être une socket ou une room )

Lorsqu’un client se déconnecte , le serveur en informe les joueurs de la room qui le supprimeront.

Voir image 4

### 4.1 Cacher les joueurs

Ajout de fonctionnalité : On veut pouvoir cacher les joueurs , j’ajoute une touche dans KeyBinding. Étant donné qu’on a une liste d’id de tous les joueurs présents, on met leur visibilité à l’inverse lors du triggers.

## 5 Amélioration

Quelques améliorations possibles mais non réalisées par manque de temps.

Améliorer l’éditeur de carte de manière à pouvoir ajouter les objets (arbres, forêt, etc...) directement ( et via un gui ). Avoir une base de données pour enregistrer les temps de chaque joueur et ainsi réaliser un scoreboard.

Il y a aussi du code que j’ai fourni, mais pas implémenter. Comme le cube, et la mine, ou encore le lampadaire. Je n’ai pas terminé leur implémentation. Aussi, parce que j’essaie de faire un jeu plutôt compétitif donc au final les blocs et item, c’est un autre style.

Le résultat n’est pas vraiment celui que j’attendais, mais je continuai de travailler dessus.



## 6 Sources

Route Center Origine : [StackOverflow](#) ( pas exactement cette technique que j'utilise mais ça m'a aider)fa [Anton Moek](#) pour la voiture low poly ( fichiers OBJ et MTL )

Exemples sur le Raycaster :

— [Lee Stemkoski](#)

— [Klaus Hofk](#)

Les différents graphiques annexes sont réalisé à l'aide du site [GeoGebra](#) et de l'outils path SVG [CodePen.io](#)

## A Annexe

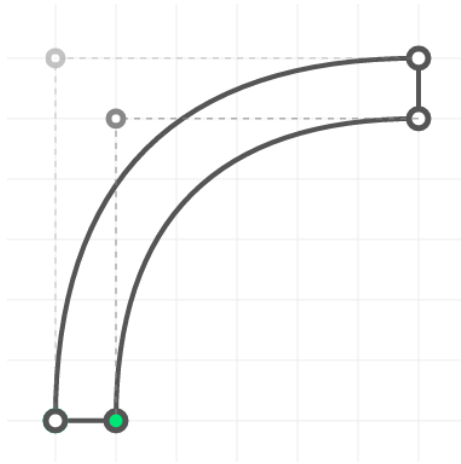


FIGURE 1 – Bezier 1 point (Quadratique)

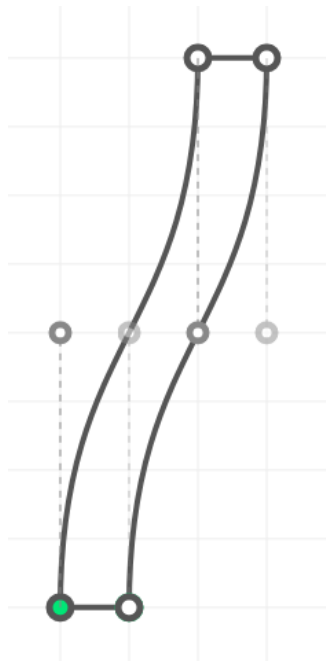


FIGURE 2 – Bezier 2 points

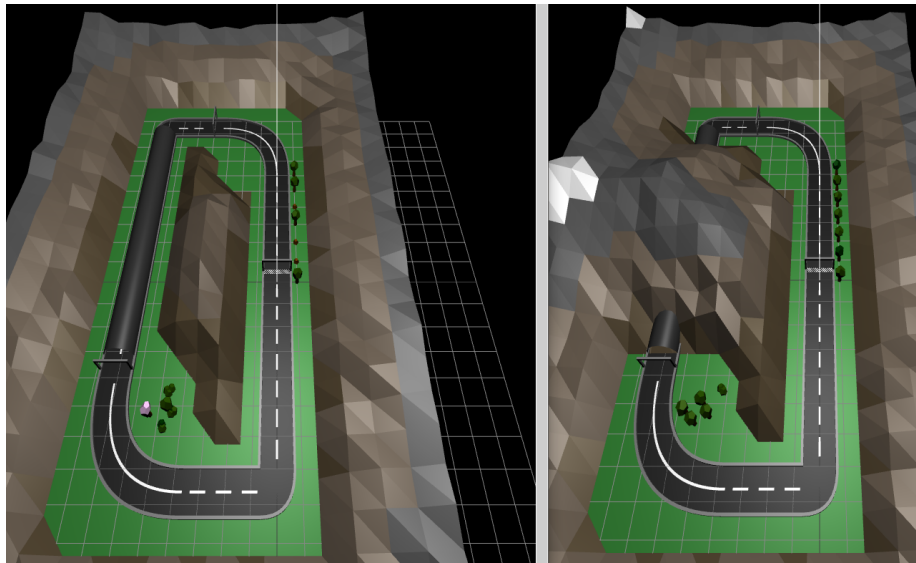


FIGURE 3 – AABB road sur tunnel pour la génération du terrain

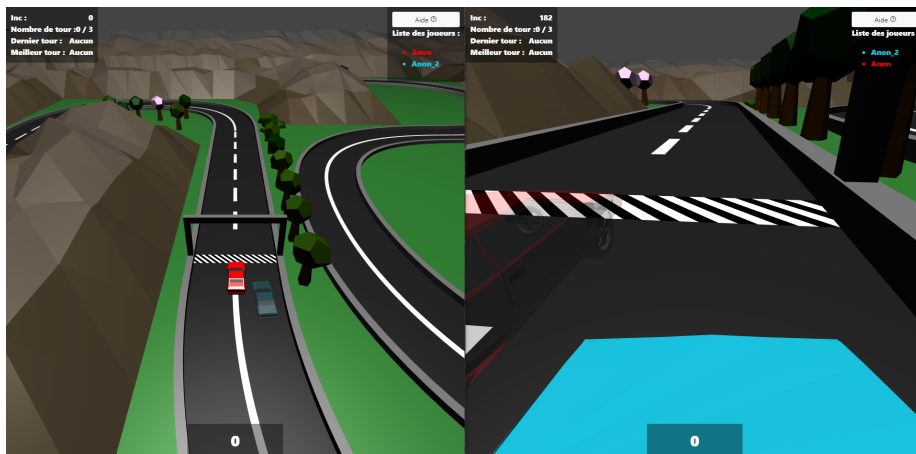


FIGURE 4 – POV de deux joueurs sur le même serveur