



POLYTECHNIQUE  
MONTRÉAL

LE GÉNIE  
EN PREMIÈRE CLASSE

# INF2705 Infographie

## Travail pratique 3

### *Rendu de scènes complexes*

Département de génie informatique et génie logiciel  
Polytechnique Montréal  
Tristan Rioux, Automne 2025

## Table des matières

<b>1 Description globale</b>	<b>2</b>
1.1 But . . . . .	2
1.2 Travail demandé . . . . .	2
<b>2 Exigences</b>	<b>8</b>
2.1 Exigences fonctionnelles . . . . .	8
2.2 Exigences non fonctionnelles . . . . .	9
2.3 Remise . . . . .	9

# 1 Description globale

## 1.1 But

Le but de TP est de permettre à l'étudiant de mettre en pratique les notions d'approximation et d'utilisation courbes, de shaders de géométrie et tessellation, puis de calcul parallèle sur la carte graphique. Il sera en mesure de dessiner une approximation de courbe bezier, de générer du gazon sur n'importe quelle surface et de faire des simulations de particules sur la carte graphique.

Le travail fera l'utilisation des fonctions d'OpenGL `glPatchParameteri`, `glDispatchCompute`, puis l'utilisation des shaders de géométrie, tessellation et de calcul.

## 1.2 Travail demandé

Le tp3 est une fois de plus une continuation du tp2. Il faudra reprendre à partir de votre ancien code du tp2 ou tp1, libre au choix. Les éléments du tp2 ne sont pas requis au fonctionnement des éléments du tp3, mais rend le rendu plus agréable. Seulement les nouvelles parties de code seront évaluées.

Quelques fichiers et fonctions de bases sont fournis.

### Partie 1 : Spline b茅zier

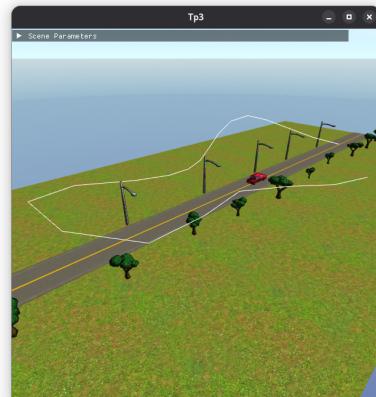
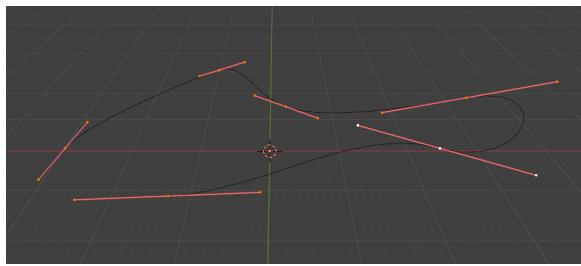


FIGURE 1 – Spline bezier

Bien que les triangles puissent nous permettre de dessiner la majorité des objets, ils ne sont pas adaptés pour représenter toutes les formes. Les courbes sont un bon exemple.

Dans un premier lieu, on fera le dessin d'une spline b茅zier. Cette spline est constitu茅 de 5 courbes b茅zi茅res 脿 interpolation cubique (donc 4 points de contr猫le par courbe).

Pour ce faire, le code de chargement de la courbe ressemblera beaucoup à celui de la partie 1 du tp1. Il est recommandé de procéder comme suit :

- Allouer suffisamment d'espace pour les sommets des courbes pour la durée totale du programme à l'initialisation. Les points faisant parties de deux courbes peuvent être dédoublé sans problème.
- Faire le calcul d'évaluation d'un point sur la courbe.
- Approximer une courbe selon le nombre de division (algorithme au choix).
- Approximer chaque courbe pour former la spline.
- Envoyer les données à jour à la carte graphique.
- Dessiner la courbe d'une couleur uniforme.

N'importe quel shader pourrait être utiliser pour le dessin de celle-ci (un nouveau très simple ou ancien). Pour que l'illumination n'ait pas d'impact dessus, le matériel `bezierMat` peut être utilisé avec le shader `CelShading`.

Une fois qu'on est certain du calcul pour approximer cette spline, on pourra animer la caméra pour suivre une trajectoire sur la spline.

Il sera pertinent d'utiliser une matrice de vue qui fixe toujours la voiture. Par la suite, on pourra déplacer la caméra selon la trajectoire.

## Partie 2 : effet de gazon procédural

Bien que la scène soit peuplée d'arbres et de lampadaires, le reste de la surface du sol semble très vide. Ajoutons-y du gazon !



FIGURE 2 – Résultat final de l'effet de gazon procédural

Pour ce faire, il faudra définir une région sur laquel générer notre gazon. À l'initialisation, on créera un mesh de plusieurs carrées constitué de triangles. Ceux-ci seront nos patches en entrée pour le shader de tessellation.

Pour commencer, il est fortement recommander de déboguer cette partie en utilisant un affichage de fil de fer afin de bien voir les divisions de la tessellation (`glPolygonMode`).

Avec le shader de contrôle de tessellation, on pourra définir le niveau de division de la patch traitée en fonction de la distance par rapport à la caméra.

Une fois qu'on sera capable de voir les divisions correctements sur le long des arêtes, on changera le mode de primitive du shader de tessellation pour générer des points (changer la taille des points dans le shader ou c++). Ces points seront l'entrée du shader de géométrie, qui pourra les convertir en brin d'herbes.

Au niveau du code, il faudra faire les modifications suivantes :

- Création du nouveau shader pour le gazon (n'oublier pas j'ai ajouté l'appel à `reload` dans `drawFrame`).
- À l'initialisation, générer les patches triangulaires de façon à avoir des carrées sur la surface du sol
- Dessin des patches, n'oublier pas d'envoyer les matrices nécessaires.
- Réalisation des shaders pour voir et déboguer les divisions.
- Modification des shaders pour générer le gazon.

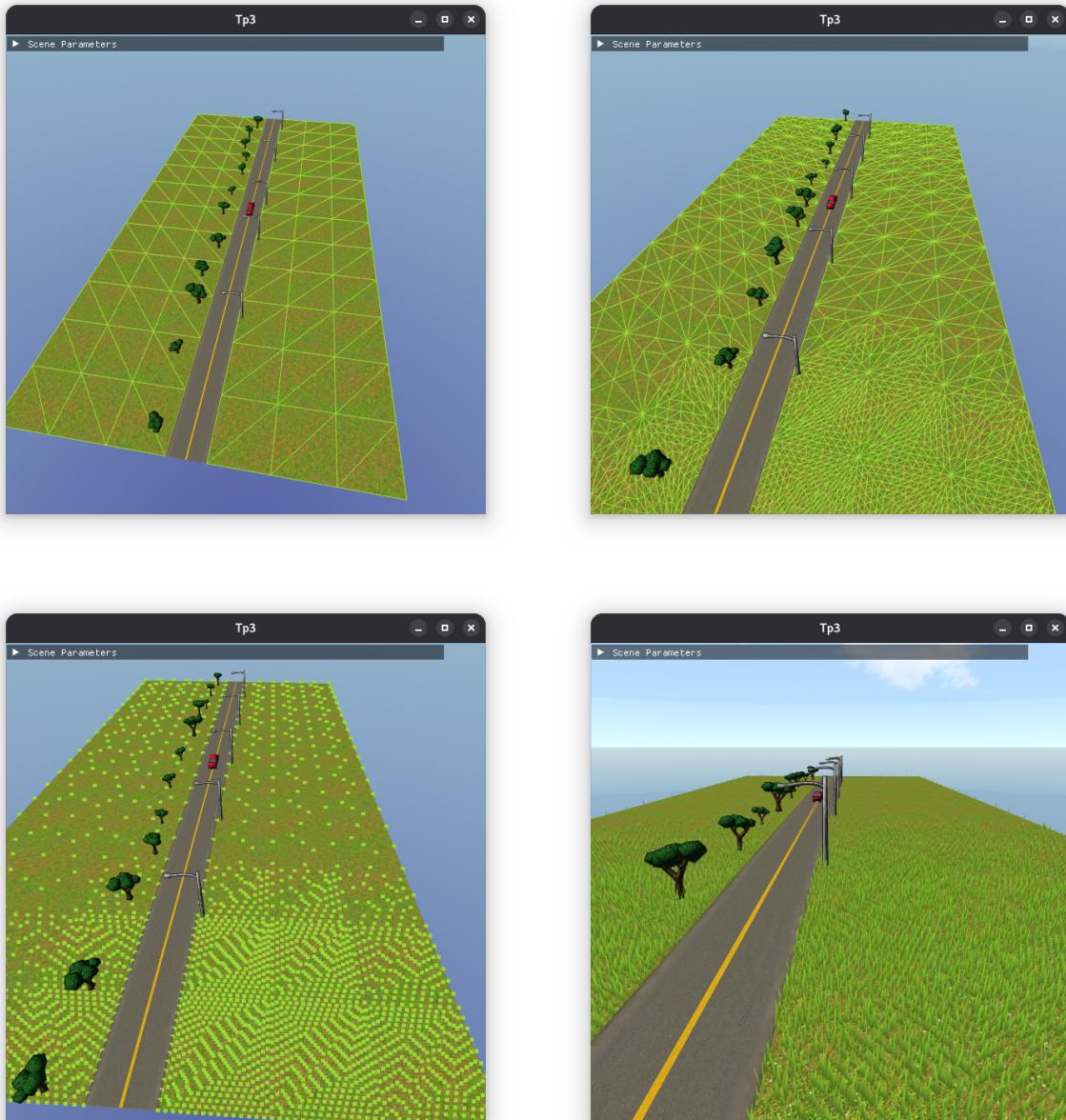


FIGURE 3 – Développement de l'effet de gazon

### Partie 3 : Effet de particules

Les effets spéciaux de particules sont souvent utilisés en infographie. Cependant le nombre de particules est souvent une limitation pour ceux-ci. On peut remédier à ce problème en utilisant le shader de calcul, qui permet de faire la mise à jour des particules sur le GPU, ce qui augmente grandement le parallélisme de l'application. Cela a aussi comme avantage d'avoir les données entièrement gérées par la carte graphique, ce qui évite de fréquent transfert d'information entre le CPU-GPU.

On demande de faire un système de particule simple pour visualiser la fumée du tuyau d'échappement de la voiture.

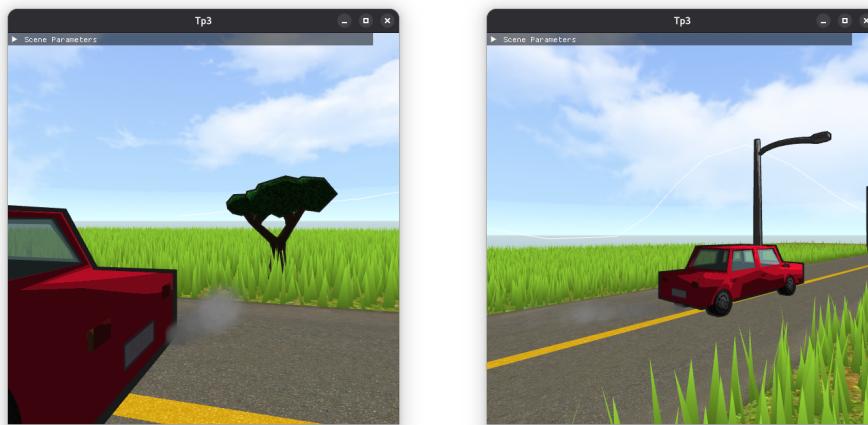


FIGURE 4 – Effet de particules de fumée réalisée par shader de calcul.

### Partie 3a : Configuration CPU du système de particules

Une nouvelle classe de buffer (SSBO, shader storage buffer object) a été fait pour vous afin d'envoyer et de recueillir les données dans le shader de calcul.

On fera l'initialisation des différents objets graphiques au démarrage. N'oubliez pas d'activer les attributs du vao.

Dans la boucle principale, on implémentera la mise à jour des particules et le dessin des particules.

Pour la mise à jour, on mettra en oeuvre une technique de "double buffering". On aura un buffer en entrée et un second en sortie. Cela aura comme avantage de faire la mise à jour en parallèle du dessin des particules sans à avoir à attendre qu'elle soit complètement effectué. À la fin de chaque frame, on pourra interchanger les buffers. Celui qui contient maintenant les données à jour devient en entrée, alors que les anciennes données d'entrée deviennent la destination d'écriture, elles seront écrasées par la mise à jour de la prochaine frame. Malheureusement, puisque le dessin des particules utilise les données en entrée de la frame précédente, cette technique entraîne 1 frame de retard comparée au reste de la simulation.

Pour le dessin des particules, il sera important spécifier la source des données avant de faire le dessin. De plus, on activera le blending classique ( $\alpha_{src}$  et  $1 - \alpha_{src}$ ) et désactivera l'écriture du tampon de profondeur.

### Partie 3b : Shader de calcul

Le shader de calcul initialise et met à jour les données des particules. Lorsque leur temps de vie est en dessous de 0.0, on réutilise la particule en l'initialisant de nouveau. Sinon, on fait la mise à jour habituelle.

Vous pouvez customiser votre effet de particule comme vous le souhaitez, mais il sera important d'avoir les transformations suivantes :

- L'initialisation des parties contient des valeurs aléatoires pour au moins une propriété.
- Les valeurs uniformes sont utilisées pour la position et vitesse initial.
- Il y a un gradient pour l'opacité pour éviter du "pop in/out" (apparaît/disparaît abruptement). Utiliser `smoothstep() * 1-smoothstep()` pour produire cette courbe.
- La position et orientation sont mise à jour avec la méthode d'Euler ( $position + vitesse * dt$ ).
- Il y a un dégradé de couleur selon la durée de vie des particules.
- Il y a une augmentation de la taille selon la durée de vie des particules.

Dans le cas d'une modification, justifié brièvement afin de montrer que l'effet est voulu.

### Partie 3c : Shader de dessin des particules

Il y a trois shaders à implémenter : vertex, géométrie et fragment.

Le shader de vertex doit calculer la position de chaque particule dans le référentielle de la vue. Par la suite, le shader de géométrie doit transformer les points en carrés centrés à leur position toujours orienté vers la caméra, pour ensuite être colorié par le fragment shader qui donne la couleur finale à l'aide du texel teinté par l'attribut couleur.

Afin de valider que le programme est fonctionnel, il est pratique d'envoyer des constantes aux variables out du vertex shader au lieu d'utiliser les données en entrée. Cela permettra de s'assurer d'avoir un rendu correct sans dépendre des calculs de rétroaction.

## 2 Exigences

### 2.1 Exigences fonctionnelles

Partie 1 :

- E1. L'allocation et destruction des objets graphiques en lien avec la courbe est fait correctement (pas de réallocation, bon mode d'usage, mise à jour d'une sous section). [0.5 pt]
- E2. Il est possible de visualiser la spline de bázier et d'augmenter ou réduire dynamiquement le nombre de points qui la constitue. [2.5 pts]
- E3. La caméra suit la courbe de façon fluide, tout en fixant l'automobile. À la fin de l'animation, la caméra est sur le dernier point de la spline. [1 pt]

Partie 2 :

- E4. Le mesh des patches constituées de triangles couvre le gazon. [1 pt]
- E5. Les brins d'herbes sont visibles des deux côtés. [0.5 pt]
- E6. Les niveaux de tessellation extérieurs sont identiques d'une patch à l'autre. [2 pts]
- E7. Le niveau de détail est dynamique et change selon la distance par rapport à l'observateur. [2 pts]
- E8. Les calculs dans le shader de tessellation de contrôle sont fait qu'une seule fois par patch. [0.5 pt]
- E9. Le shader de tessellation d'évaluation émet des points répartis sur la surface. [0.5 pt]
- E10. Le shader de géométrie convertit les points en brin d'herbe. [0.5 pt]
- E11. Les brins d'herbe sont de tailles variables et orientés de façon aléatoire en y et x. [0.5 pt]
- E12. Un dégradé de vert est appliqué de bas en haut sur les brins d'herbes. [0.5 pt]

Partie 3 :

- E13. L'initialisation des SSBOs alloue les buffers correctement. [0.25 pt]
- E14. Le mode d'utilisation des SSBOs correspond à un mode utile pour **toutes ses utilisations**. [0.5 pt]
- E15. La configuration de la mise à jour des particules permet de modifier les points. [0.75 pt]
- E16. La configuration du dessin permet de voir les particules. [0.75 pt]
- E17. Le blending est activé et les particules n'écrivent pas le tampon de profondeur. [0.25 pt]
- E18. Dans le compute shader, l'initialisation des particules donne des valeurs par défaut correct à tous les attributs. [0.25 pt]
- E19. La mise à jour de la position des particules est fait avec la méthode d'Euler. [0.25 pt]
- E20. La mise à jour de l'orientation est fait à un rythme constant. [0.25 pt]
- E21. La mise à jour de la couleur passe de gris à blanc de façon linéaire en fonction du temps. [0.25 pt]
- E22. La mise à jour de la transparence à un fade in/out et est en temps normal transparente. [0.25 pt]
- E23. La mise à jour de la taille agrandit la taille des particules de façon linéaire en fonction du temps. [0.25 pt]
- E24. Seulement les attributs nécessaires sont utilisés en entrée/sortie pour le dessin des particules. [0.25 pt]

- E25. Le dessin des particules convertie les points en carrée de taille appropriée orienté dans l'écran. [0.5 pt]
- E26. Les particules tournent sur elles-mêmes. [0.5 pt]
- E27. Les texels invisible ne sont pas traîtés. [0.25 pt]
- E28. La couleur des particules est la texture teintée par la couleur de la particule. [0.5 pt]

## 2.2 Exigences non fonctionnelles

De façon générale, le code que vous ajouterez sera de bonne qualité. Évitez les énoncés superflus (qui montrent que vous ne comprenez pas bien ce que vous faites !), les commentaires erronés ou simplement absents (lorsque nécessaire), les mauvaises indentations, etc. Retirer les commentaires de TODOs dans la remise. [2 pts]

## 2.3 Remise

Créer une archive nommée « **INF2705\_remise\_TPn.zip** » que vous déposerez ensuite dans Moodle. (Moodle ajoute automatiquement vos matricules ou le numéro de votre groupe au nom du fichier remis.)

Ce fichier zip contient tout le code source du TP que vous avez modifié (`main.cpp`, `models.cpp`, `model_data.hpp`, `shaders.cpp`, `shaders.hpp`, `car.cpp`, `car.hpp`, `textures.cpp`, `shaders/*.glsl`).