Roll-a-ball tutoriel Godot

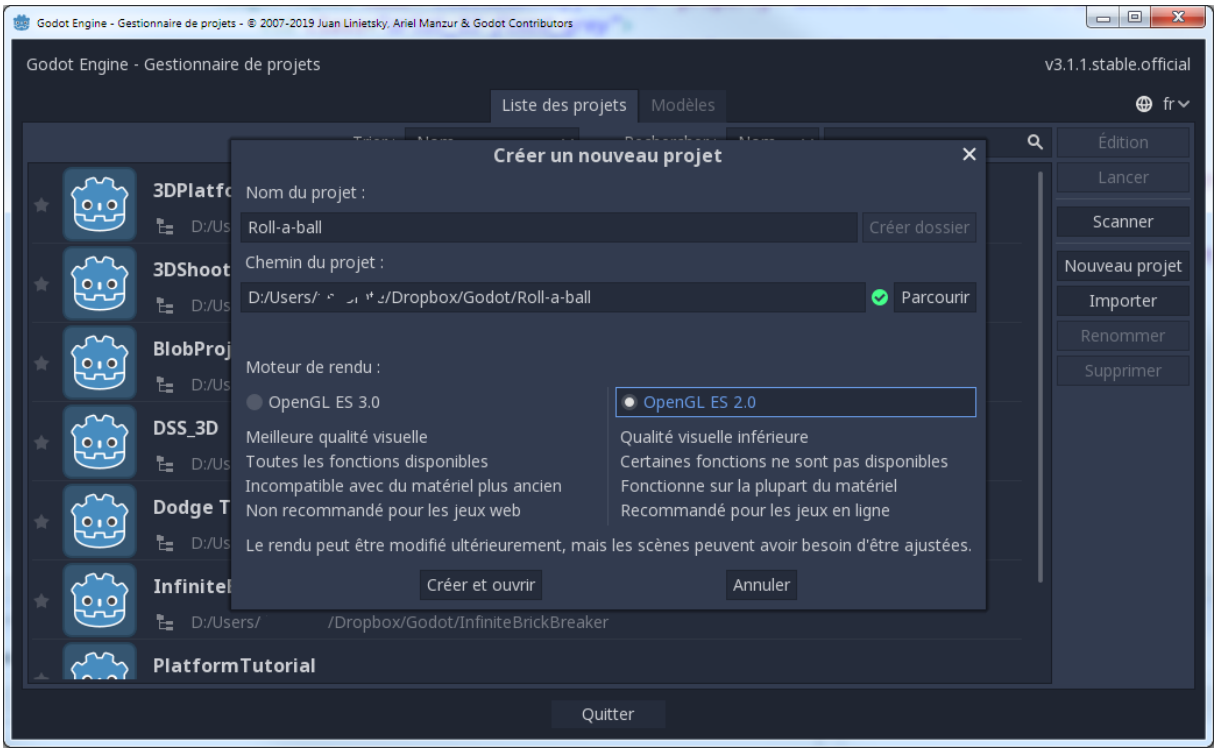
Tutoriel basé sur celui disponible pour le moteur Unity(https://unity3d.com/fr/learn/tutorials/s/roll-ball-tutorial) et adapté pour Godot 3.1.

Le but de ce tutoriel est de créer un jeu de « balle roulante ». Il permettra de se familiariser avec les principes de base du moteur 3D de Godot. Il utilise le moteur physique de Godot pour gérer les mouvements, le code utilisé sera donc très simple puisque nous n'aurons aucun mouvement à gérer à la main.

Aucun asset n'est requis pour ce tutoriel.

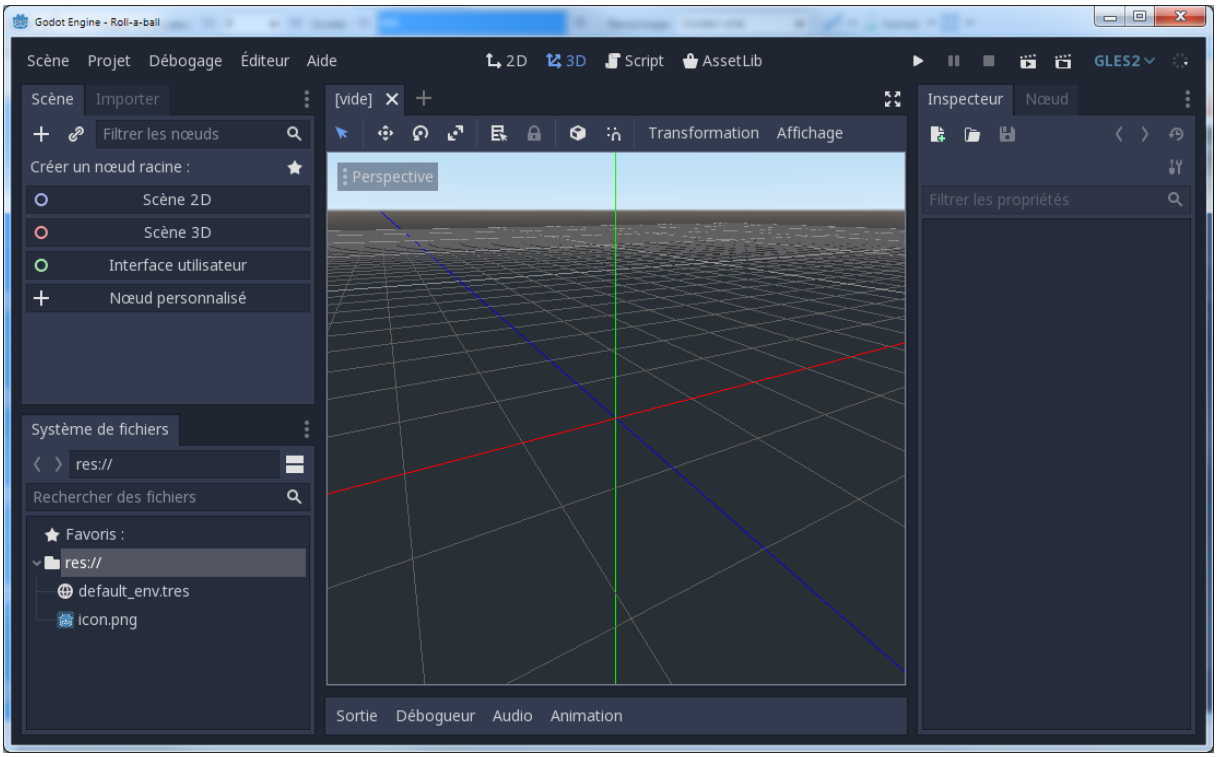
Démarrer un nouveau projet

Après avoir téléchargé et extrait l'exécutable Godot, le lancer et créer un « Nouveau Projet ».



Nom du projet : le nom du projet (obvious)
Chemin du projet : répertoire dans lequel seront stockés les fichiers du projet. Ce répertoire doit être vide. Il est possible de sélectionner un répertoire non vide (genre, celui dans lequel on place tous ses projets Godot) et de cliquer sur « Créer dossier » pour qu'un dossier portant le nom du projet soit automatiquement créé.
Moteur de rendu : choix de OpenGL 2 ou 3 pour le rendu du jeu. Pour un jeu standalone, graphiquement au top et nécessitant de bonnes machines, utiliser le 3 (mais on oublie le jeu sur navigateur). Sinon, choisir le 2. Pour ce projet, nous allons choisir OpenGL ES 2.0. Il est toujours possible de modifier le moteur par la suite, mais cela peut demander des modifications des assets du jeu (matériaux, particules, …) pour que le rendu soit le même.

Cliquer sur « Créer et ouvrir ». On arrive sur l'interface de Godot :



Il est possible que l'arrangement des onglets ne soit pas le même que sur l'écran ci-dessus. La disposition des onglets est configurable et chacun trouvera celle qui lui convient le mieux.

Scène : graphe de la scène courante.
Inspecteur : propriétés du nœud actuellement sélectionné.
Nœud : propriétés et signaux du nœud actuellement sélectionné.
Système de fichier : navigateur des fichiers du projet.
Au centre : la vue 3D de la scène.

Créer un jeu consiste à créer une ou plusieurs scènes et à créer une hiérarchie d'éléments (graphe) représentant les éléments du jeu. Il est alors possible d'ajouter des scripts (des comportements programmés) aux différents nœuds pour rendre le tout interactif.

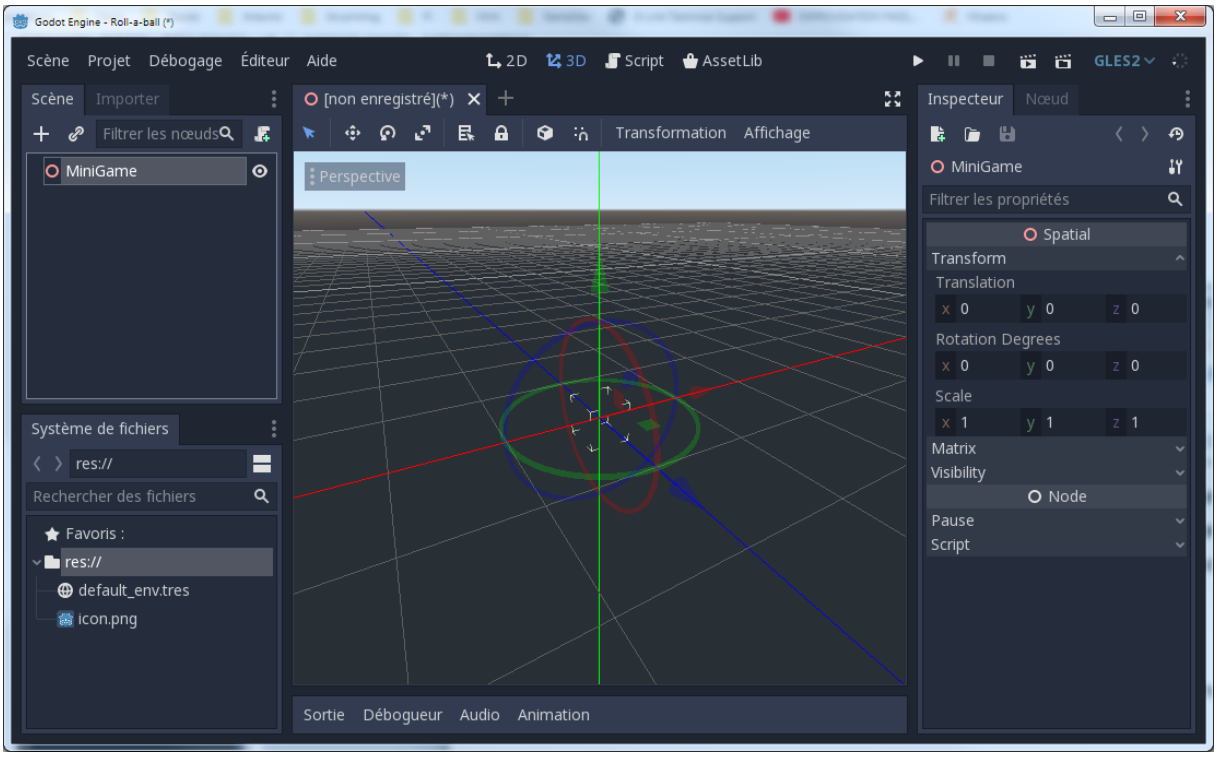
Environnement et joueur

Création du nœud racine

Dans l'onglet « Scène », créer un nœud racine de type « Scène 3D ». Cela créé un nœud de type « Spatial ».

Spatial est le type de nœud générique pout tout élément se trouvant dans une scène en trois dimensions. Ces propriétés se résument à sa position et ses transformations dans l'environnement (cf onglet Inspecteur, propriété Transform du Spatial).

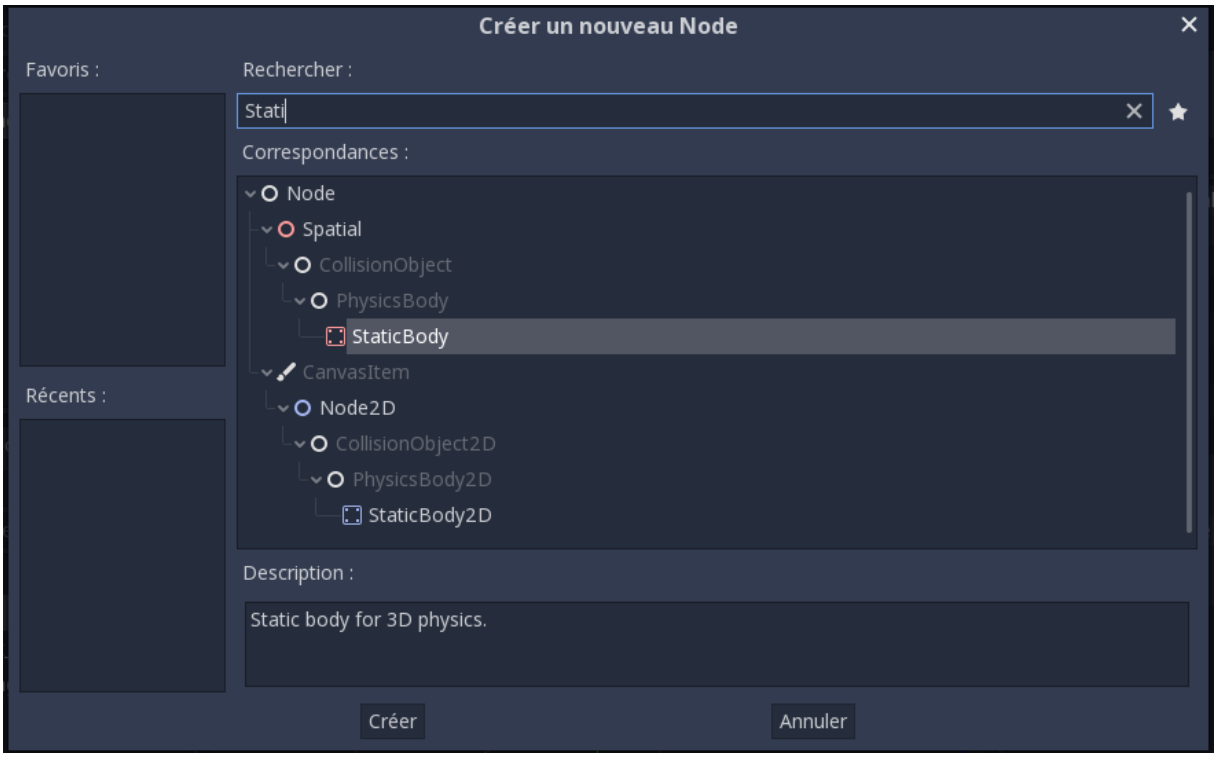
Double cliquer sur le nœud dans le graphe pour le renommer en « MiniGame ».



Création du sol

Faire un clic droit sur le nœud MiniGame et sélectionner « Ajouter un nœud enfant » (la même opération est possible en sélectionnant le nœud et Control + A).

Le sol sera un simple plan 2D dans l'environnement 3D. Il ne sera soumis à aucune physique mais participera aux collisions (avec la balle) et aura une position fixe. Ce nouveau nœud sera donc de type StaticBody. Le StaticBody se trouve dans Node / Spatial / Collision Object / StaticBody. Il est également possible d'utiliser le champs de recherche pour le trouver plus rapidement.



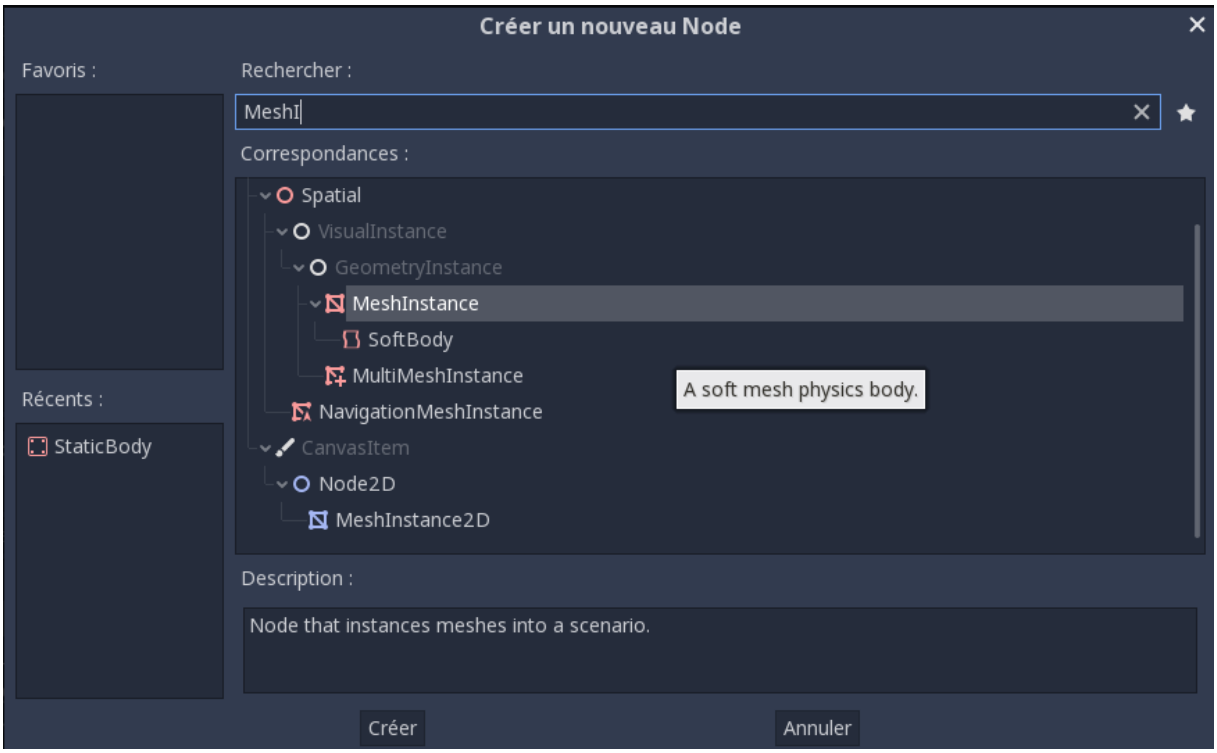
Cliquer sur « Créer ».

Un nœud StaticBody a été ajouté au graphe de la scène. Mais pour l'instant, la vue 3D n'affiche pas de sol. Il nous faudra ajouter un nœud pour préciser quel objet afficher pour figurer le sol. En attendant, renommer ce nouveau nœud en « Sol ».

Le warning à droite du nœud signifie qu'il n'est pas complet. Un StaticBody doit avoir une zone de collision définie pour participer à la collision dans le jeu. Nous nous occuperons de cela un peu plus tard.

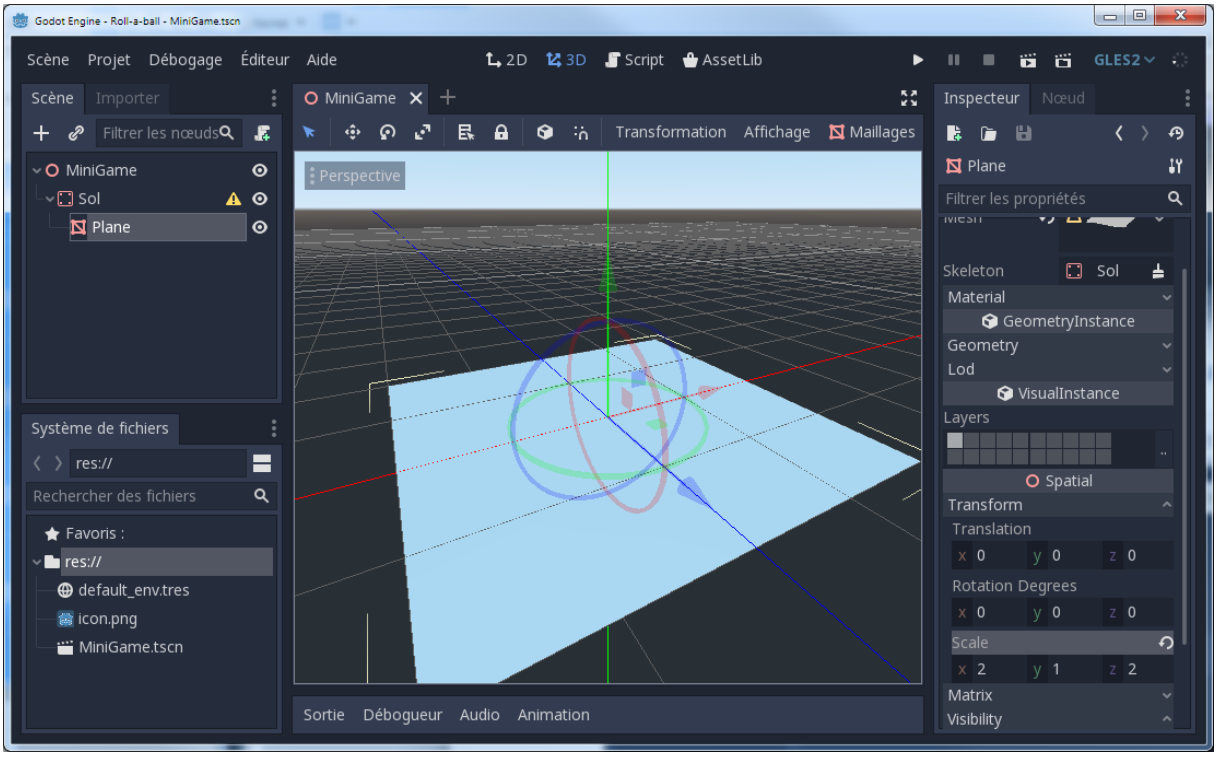
Il est temps de sauvegarder la scène : « menu Scène / Enregistrer la scène » ou Control + S. La scène est sauvegardée dans le système de fichier du projet (un projet peut comporter plusieurs scènes).

Ajouter un nœud de type MeshInstance au nœud « Sol »



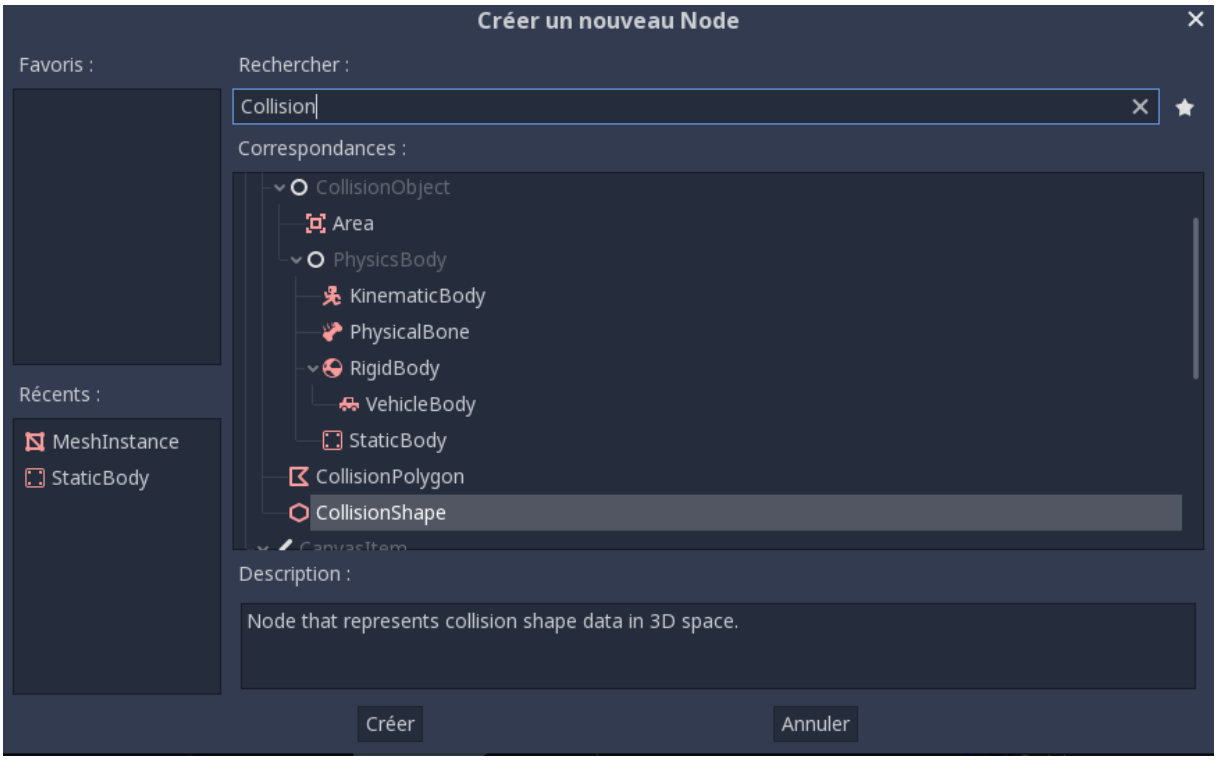
Les nœuds de type MeshInstance permettent de donner une représentation 3D à un nœud.

Renommer le nouveau nœud en « Plane » et dans ses propriétés, sélectionner un Mesh de type « Nouveau PlaneMesh ». Dans les propriétés du type Spatial du Mesh, modifier le Scale en X et Z à 2 pour agrandir le plan.



On va désactiver la grille pour plus de visibilité : dans la vue du milieu > Affichage / décocher Afficher la grille.

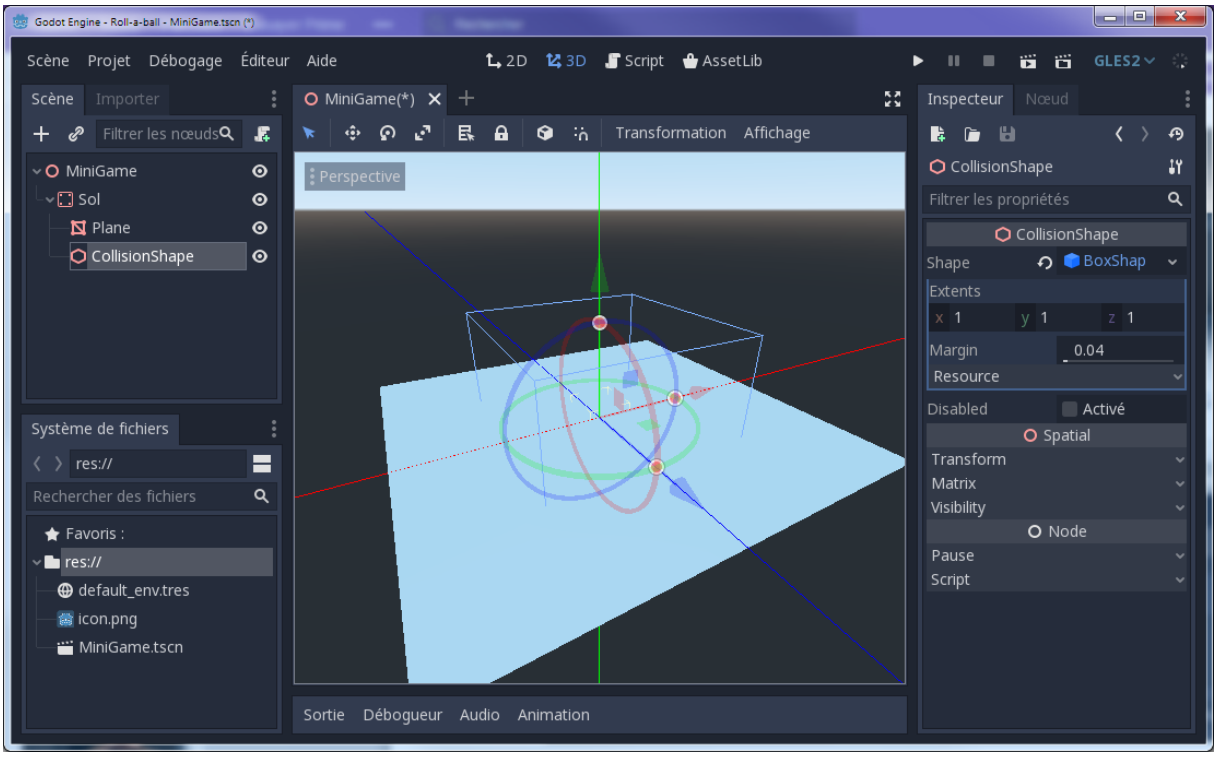
Le plan qui s'affiche n'a aucune existence physique pour l'instant. Ce n'est qu'un maillage affiché à l'écran qui serait traversé par les autres et qui ne participe pas au moteur physique. On va remédier à cela.
Ajouter un nœud de type CollisionShape au nœud Sol.



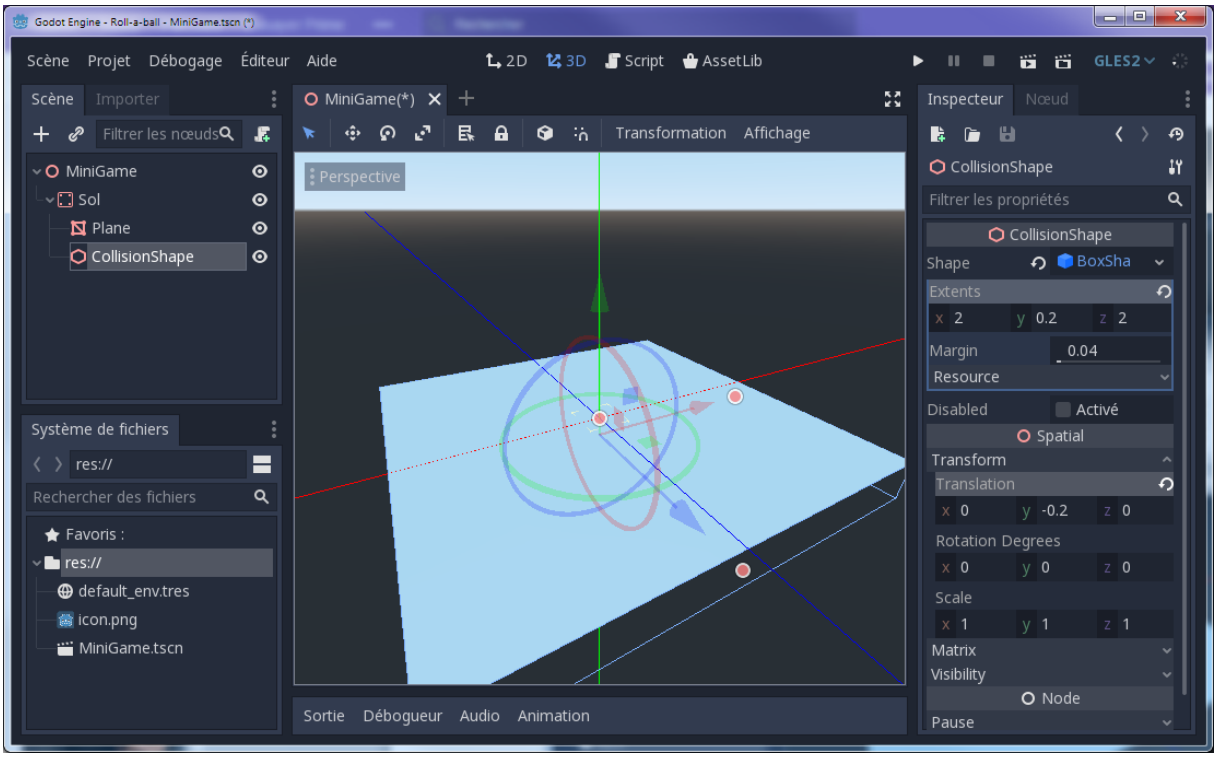
Dans les propriétés de la CollisionShape, sélectionner la forme (Shape) BoxShape. Un cube apparait à travers le plan. Il s'agit de sa boite de collision. Il faut la redimensionner pour correspondre à la forme du plan.

ATTENTION : pour redimensionner une forme de collision, il ne faut jamais toucher à son scale en tant que Spatial. Cela casserait le calcul de la physique. Une forme de collision a toujours un scale de (1,1,1).

Cliquer sur « BoxShape » pour afficher ses Extents. Ces valeurs correspondent à la taille de la boite. Elles peuvent également être modifiées en utiliser les points rouges affichés dans la vue centrale.



Dans Extents, mettre les valeurs X=2, Y=0.2, Z=2. On obtient une dalle. Il faut maintenant la déplacer suivant l'axe Y pour faire correspondre le haut de la boite avec le plan. Pour cela, utiliser la flèche de translation verte, ou éditer la translation du Spatial de notre boite en mettant la valeur -0.2 en Y.

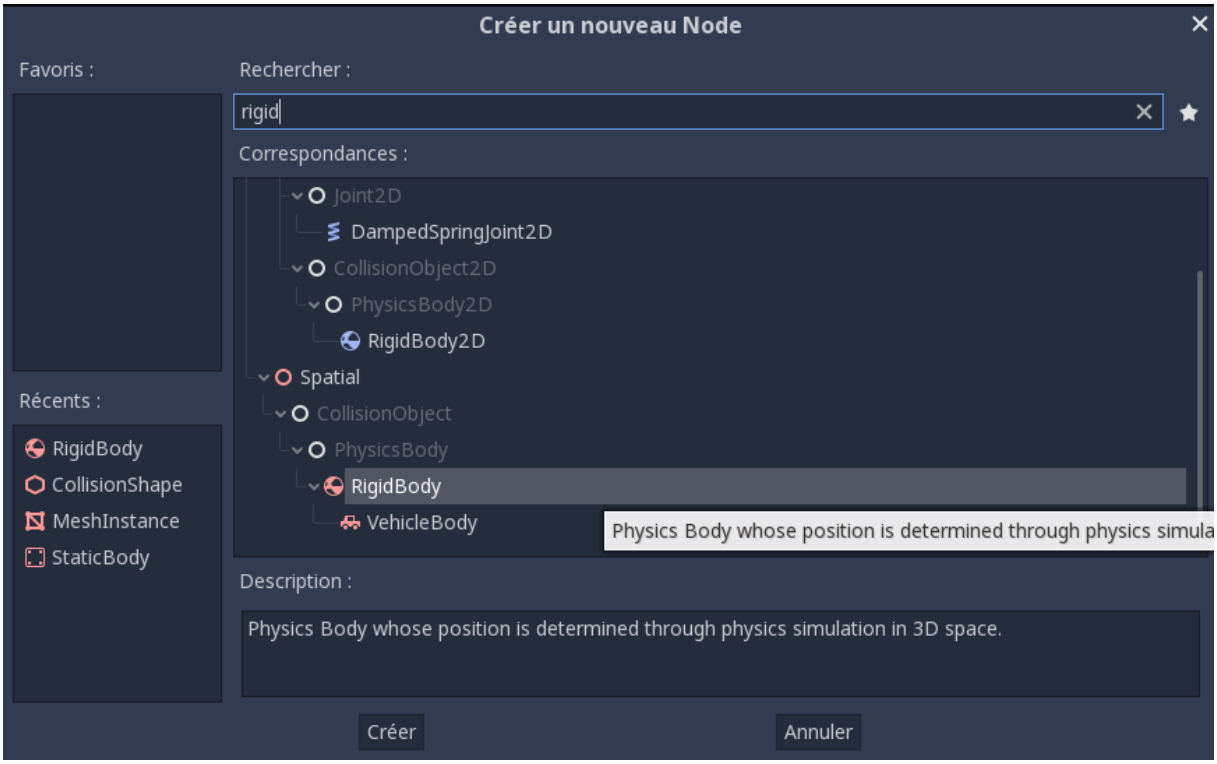


Notre sol est prêt.

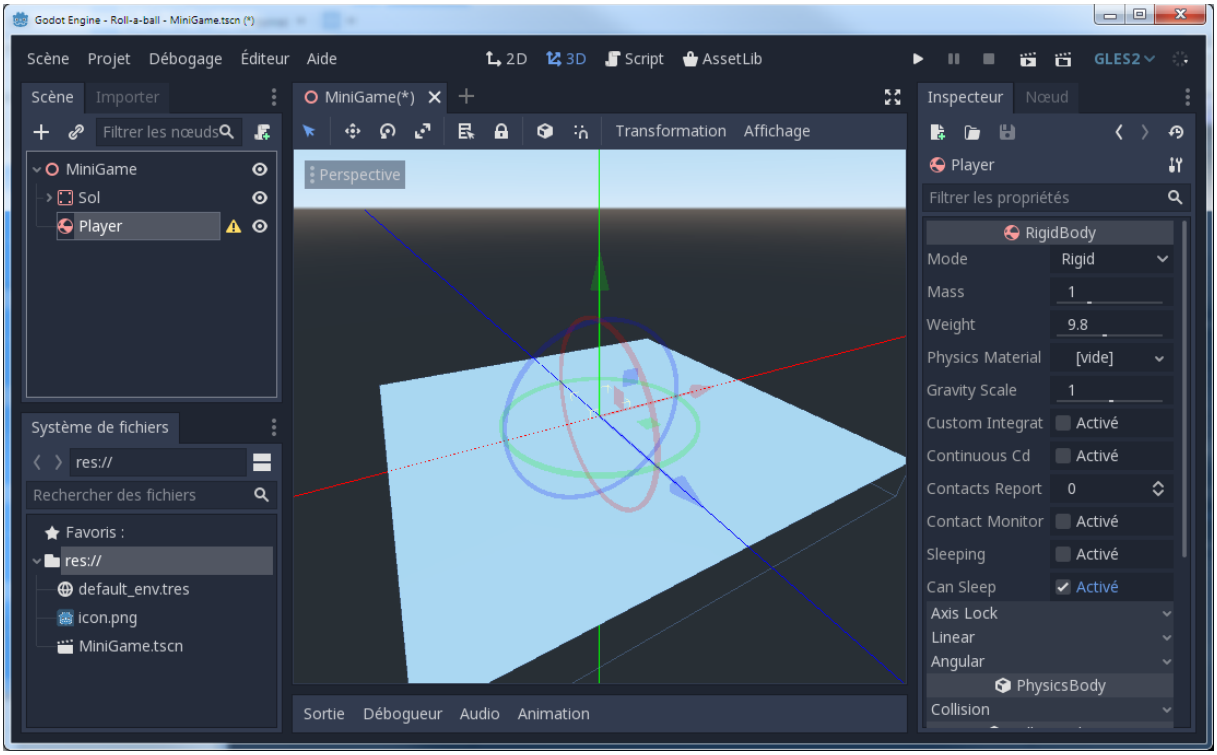
Ajout du joueur

Le « joueur » sera la balle, donc une sphère. Ses mouvements ne seront pas directement contrôlés par le joueur. Il sera soumis à la physique et on pourra modifier son comportement en lui appliquant des forces. On va donc utiliser un nœud de type RigidBody.

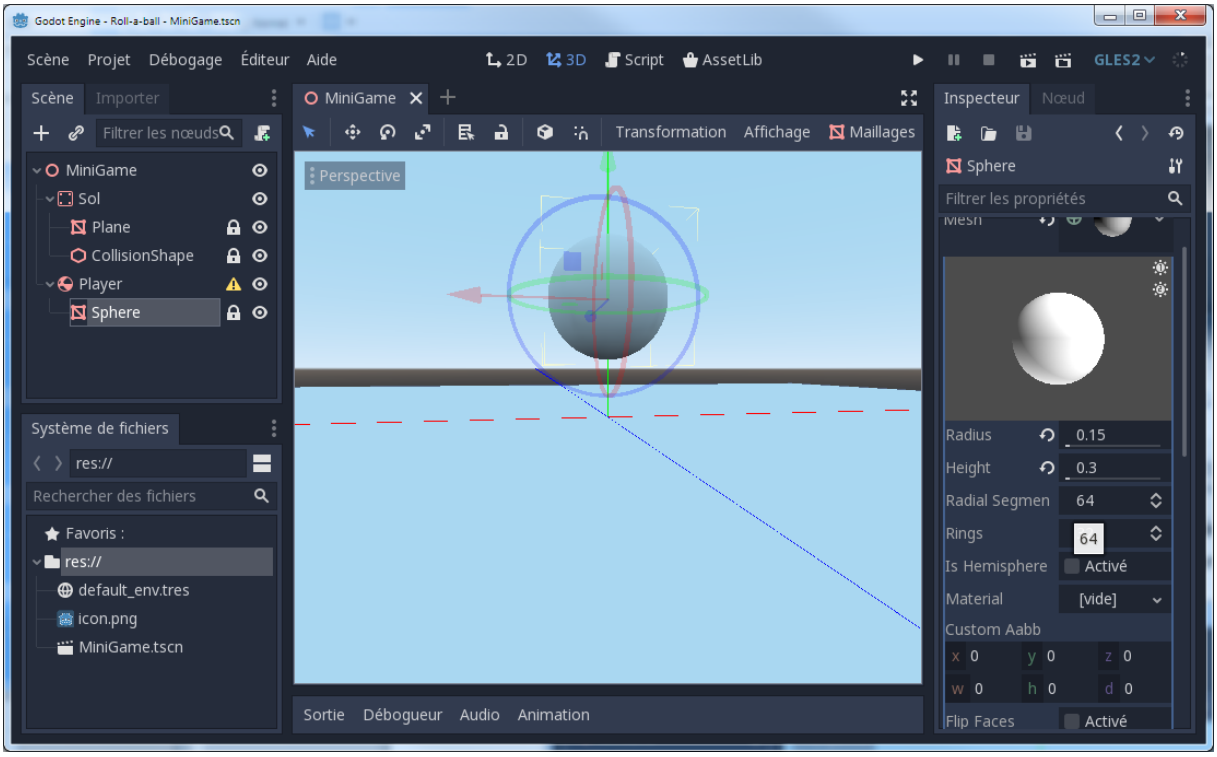
Créer un nouveau nœud sous « MiniGame » :



Renommer le nœud en « Player ».



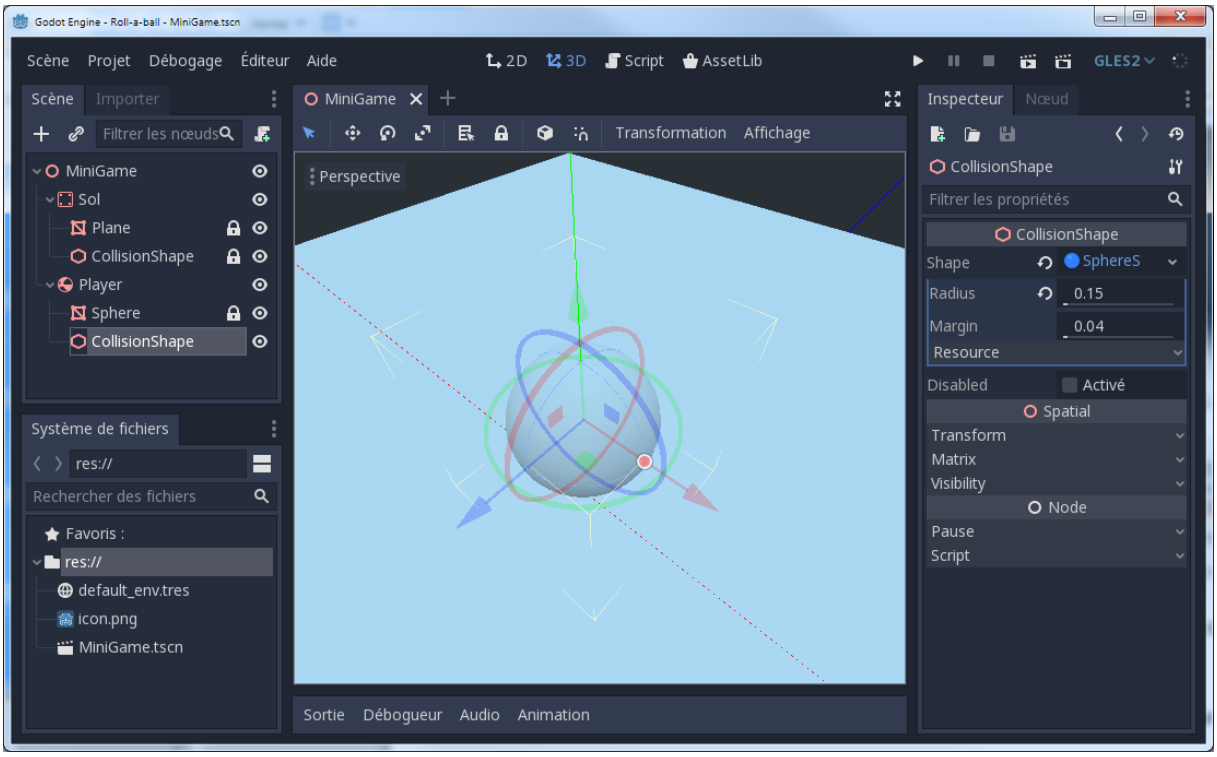
Ajouter un MeshInstance à Player en lui donnant une forme de sphère. Modifier la forme pour lui donner un rayon de 0.15 et une hauteur de 0.3. Modifier son nom en « Sphere ».



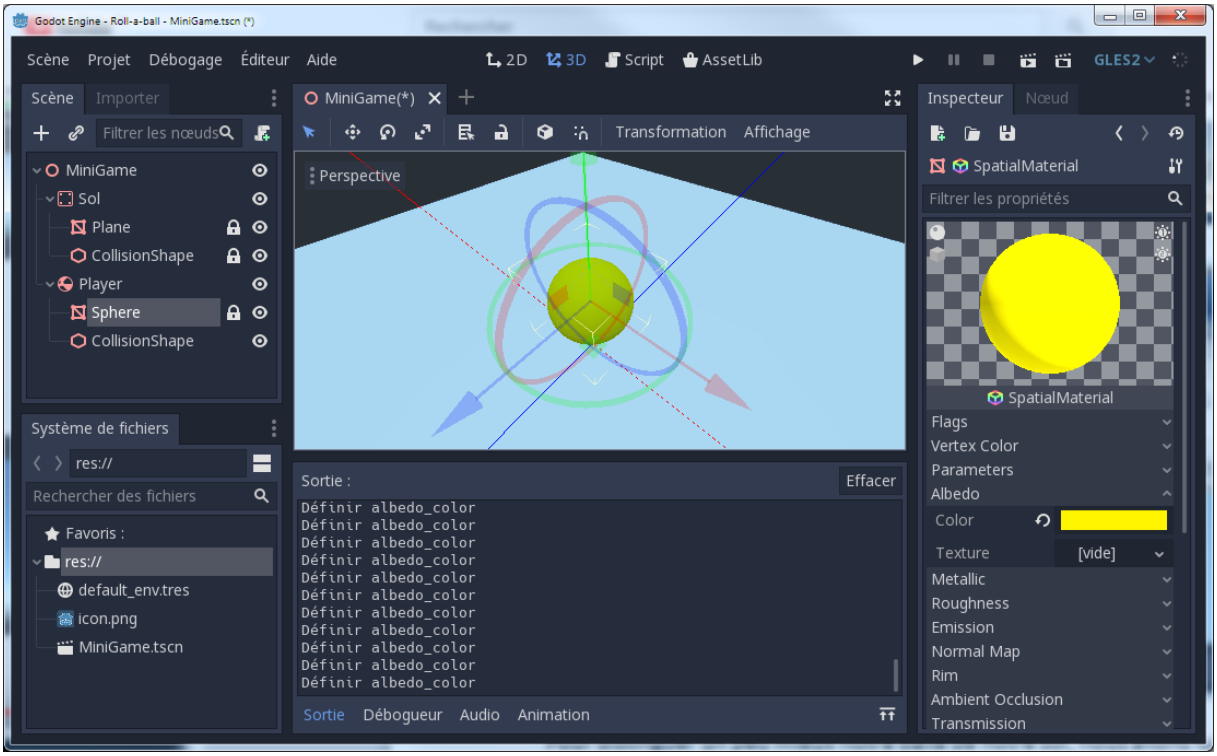
Les cadenas sur la copie d'écran ci-dessus correspondent au nœuds qui ont été « lockés » (clic sur le cadenas en haut). Cela permet d'interdir le déplacement de ces neouds, relativement à leur parent et ainsi éviter toute mauvaise manipulation qui déplacerait le mesh sans déplacer la boite de collision. Il est bon de le faire quand les nœuds enfants sont bien placés relativement à leur parent.

Modifier le transform / Translation / Y du nœud Player (surtout pas de la sphère) pour que la balle se trouve au-dessus du sol (y=0.3).

Ajouter une CollisionShape à Player, de forme sphérique en reprenant les mesures du Mesh.

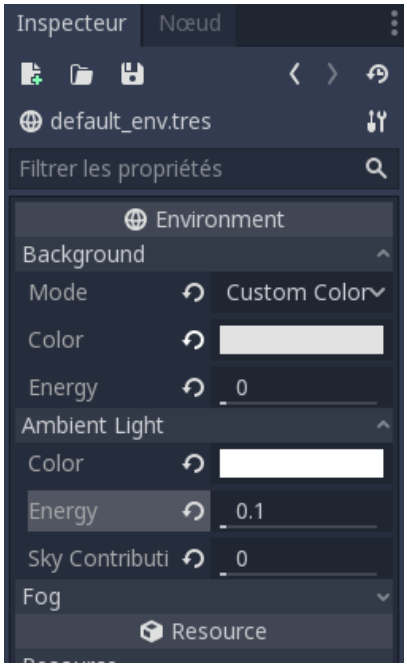


Pour distinguer un peu mieux notre balle de notre sol, nous allons lui allouer un material (une matière en français ?). Cliquer sur le nœud Sphere, ouvrir le volet Material et cliquer sur le [vide] pour ajouter un nouveau Material de type SpatialMaterial. Cliquer sur l'aperçu du material (la sphère blanche) pour modifier ses propriétés. Modifier l'albedo pour lui donner la couleur souhaitée.



A noter que notre sol n'a pas de material associé, il est donc totalement blanc. Sauf que la lumière ambiante par défaut est bleue… Libre à chacun de remédier à cela en lui donnant une belle couleur.

Ajouter une source lumineuse et une caméra



Pour l'instant , notre scène n'est éclairée que par la lumière ambiante portée par la skybox décrite dans le fichier default_env.tres. Double cliquer sur ce fichier afin de modifier l'environnement.

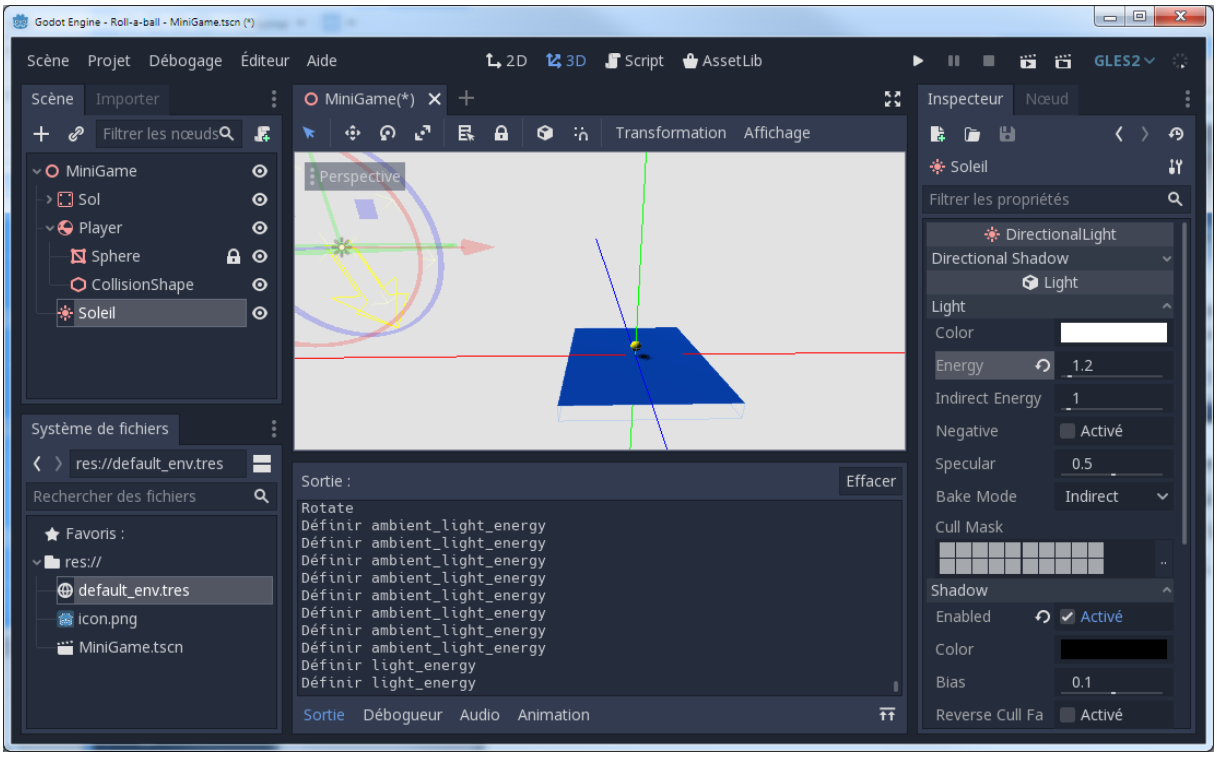
Mode : sélectionner Custom Color pour spécifier une couleur unique pour le background

Color : sélectionner un gris très clair

Ambiant Light / Energy : 0.1

Ajouter une source de lumière de type DirectionalLight au nœud MiniGame.

Appeler ce nouveau nœud « Soleil ». Il peut être placé n'importe où (localisation en XYZ dans son transform). La seule chose qui compte, c'est la rotation qui lui est appliqué et qui indique le sens par lequel arrive la lumière. Il faut également penser à activer les ombres qui seront portées par cette source lumineuse (Shadow / enabled dans ses propriétés).

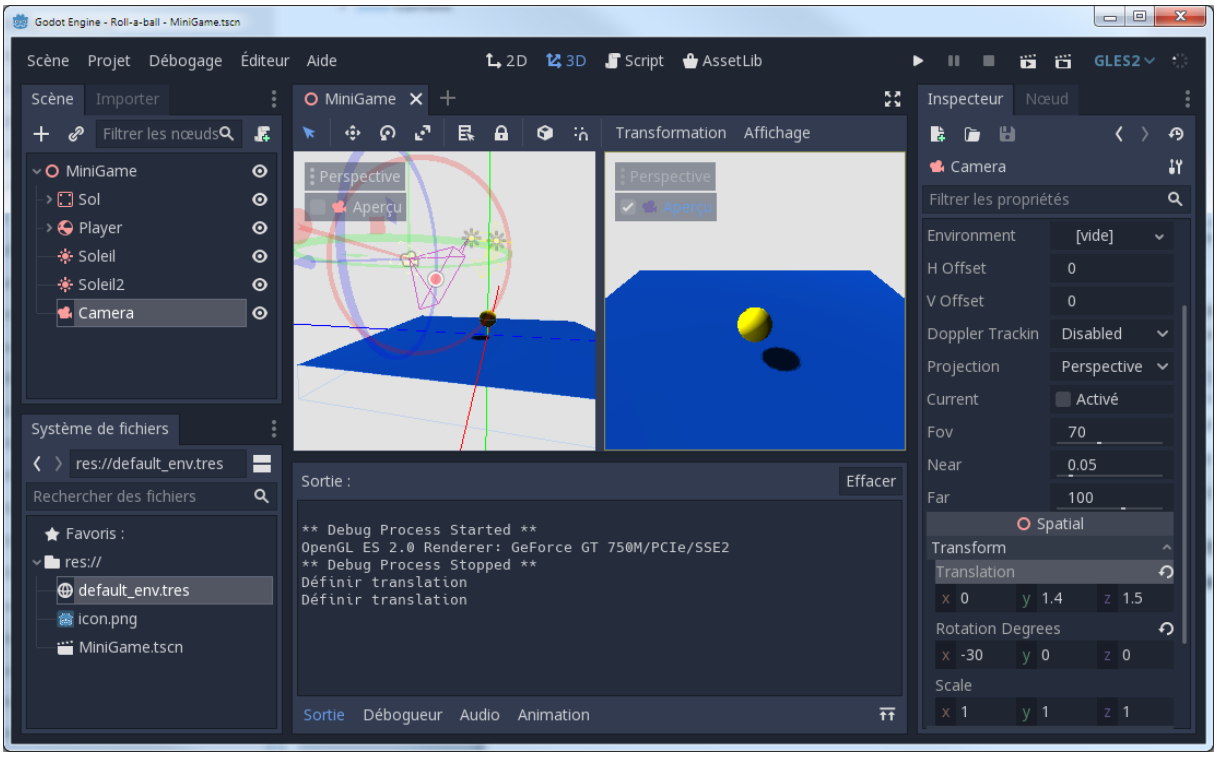


Dupliquer le soleil (Control+D avec le nœud sélectionné) pour créer une autre source lumineuse. La rendre moins puissante et désactiver les ombres. Lui donner une orientation permettant de déboucher les zones sombres là où notre premier soleil ne porte pas.

Maintenant que la scène est éclairée, il faut une caméra pour l'afficher.

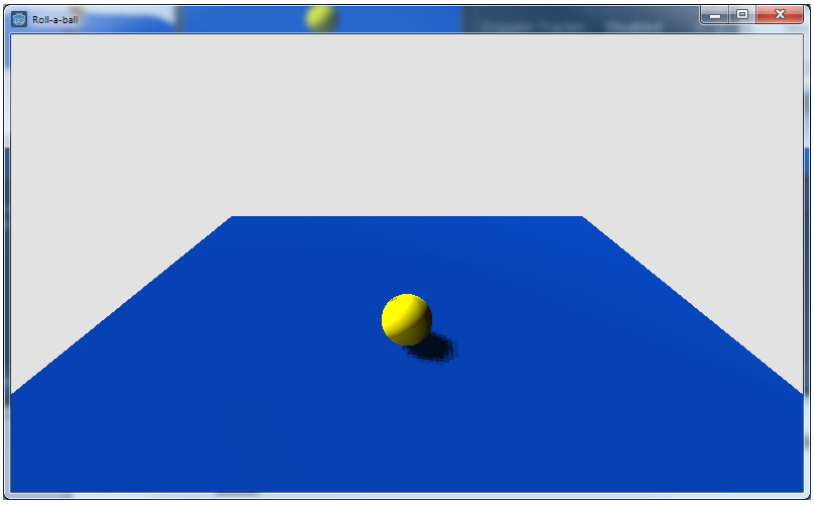
Une scène sans caméra ne peut pas être rendue. Une scène doit donc avoir au moins une caméra. Plus précisément, chaque ViewPort doit avoir au moins une caméra. Plusieurs caméras peuvent exister pour un même ViewPort mais une seule peut être active à la fois.

Ajouter un nœud de type Camera dans MiniGame. La positionner afin de voir la balle. Pour cela, il possible de cliquer sur « Aperçu » quand la caméra est sélectionnée pour avoir une vue depuis la caméra.



Il maintenant possible de tester le jeu, en cliquant sur le bouton de démarrage de la scène courante :

La balle tombe un peu et s'arrête sur le sol… C'est un début.

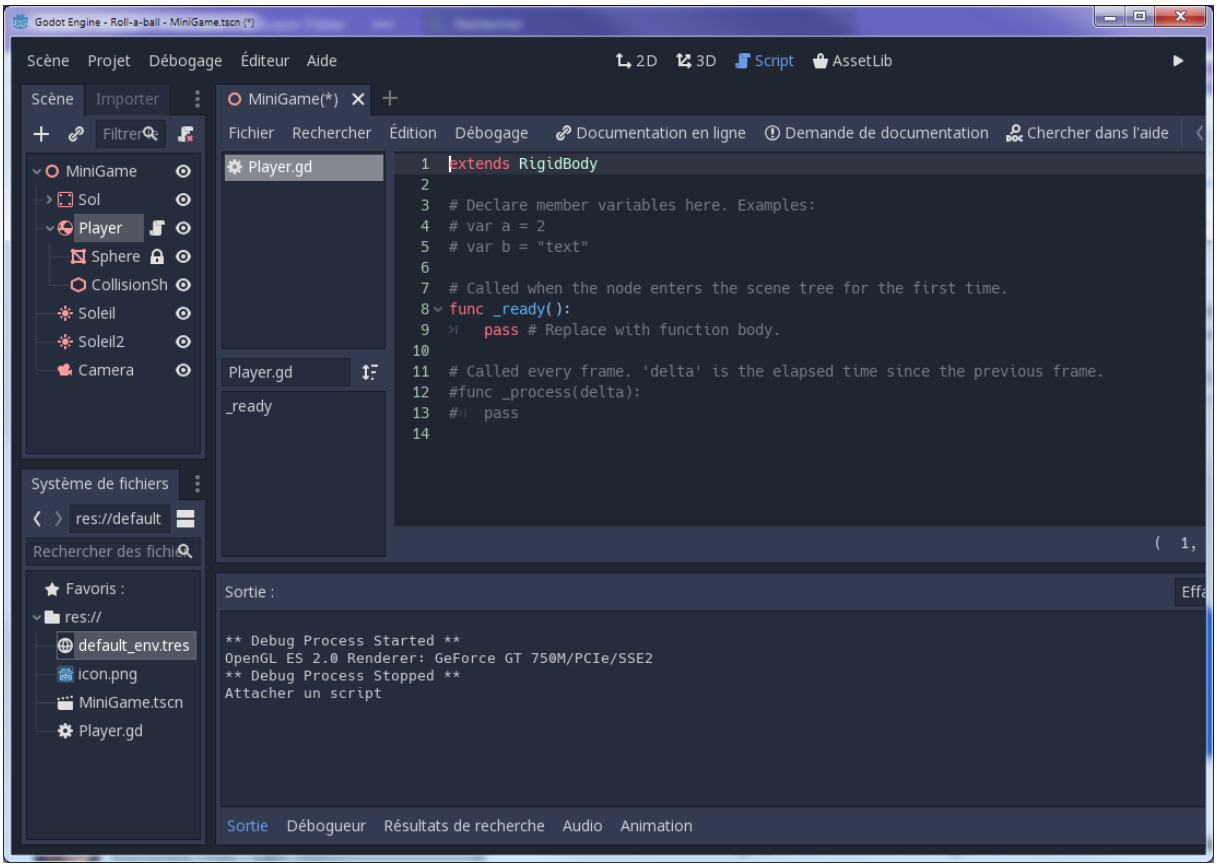


Déplacements

Déplacer la balle

La suite logique consiste à « diriger » la balle. Comme on fait appel au moteur physique, il s'agira plutôt d'appliquer des forces sur la balle.

Associer un script au nœud Joueur : sélectionner le nœud, clic droit, Attacher un script. Valider la popup. L'éditeur de GDScript s'affiche.



Il est possible de switcher entre le script et la vue 3D avec boutons se trouvant en haut de l'interface.

La fonction _ready est déjà déclarée, il s'agit d'une méthode invoquée à lé création de l'objet. Dans notre cas, elle ne nous sert à rien. Ce qui nous intéresse c'est de vérifier les touches appuyées à chaque frame pour diriger la balle. Et comme on utilise le moteur physique, la fonction à implémenter est _physics_process.

```
1  extends RigidBody
2
3  # Called when the node enters the scene tree for the first time.
4  func _ready():
5      pass # Replace with function body.
6
7  func _physics_process(delta):
8      pass
```

Dans cette fonction, on vérifie les touches appuyées par l'utilisateur pour déterminer la direction de la force à appliquer à la balle. Si aucune touche n'est appuyée, on ralentit la balle en lui appliquant une force inverse à sa vitesse. On vérifie également que la balle ne puisse pas dépasser une vitesse maximum.

```gdscript
extends RigidBody

const MAX_SQUARED_VELOCITY = 100

# Called when the node enters the scene tree for the first time.
func _ready():
    pass # Replace with function body.

func _physics_process(delta):

    var force = Vector3.ZERO

    if Input.is_action_pressed("ui_up") :
        force.z = -1
    elif Input.is_action_pressed("ui_down"):
        force.z = 1

    if Input.is_action_pressed("ui_left") :
        force.x = -1
    elif Input.is_action_pressed("ui_right"):
        force.x = 1

    if force == Vector3.ZERO:
        force = -linear_velocity / 2

    if linear_velocity.length_squared() < MAX_SQUARED_VELOCITY :
        add_force(force, transform.origin)
```

On peut tester le jeu. La balle doit se déplacer en fonction des touches du pavé directionnel. Par contre, si la balle tombe, on n'a plus qu'à fermer le jeu. Autre problème, si la balle passe derrière la caméra, on ne la voit plus…

Déplacer la caméra

Une solution simple pour que la caméra suive le joueur, est de déplacer le nœud Camera pour en faire un enfant du nœud Player.

Cependant, si on fait cela, on se rend compte que la caméra suit également les rotations de la balle. Or, comme la balle roule, la caméra se retrouve à rouler également… Ce qui rend le jeu un peu compliqué à jouer.

La solution va donc consister à code un script s'assurant que la caméra est toujours est la même position relativement au joueur.

S’assurer que le nœud Camera est bien un enfant direct de MiniGame. Attacher un nouveau script à Camera.

```
1  extends Camera
2
3  # référence sur le joueur
4  onready var player = get_node("../Player")
5  # décalage de la caméra par rapport au joueur
6  var offset
7
8  func _ready():
9      offset = translation - player.translation
10
11 func _physics_process(delta):
12     translation = player.translation + offset
13
```

A l’initialisation, la caméra repère la position du joueur et la compare à sa propre position pour déterminer le décalage à conserver (offset).
A chaque frame, la position de la caméra est recalculée à partir de la position du joueur.

Si on teste, on constate que la caméra suit parfaitement la balle.

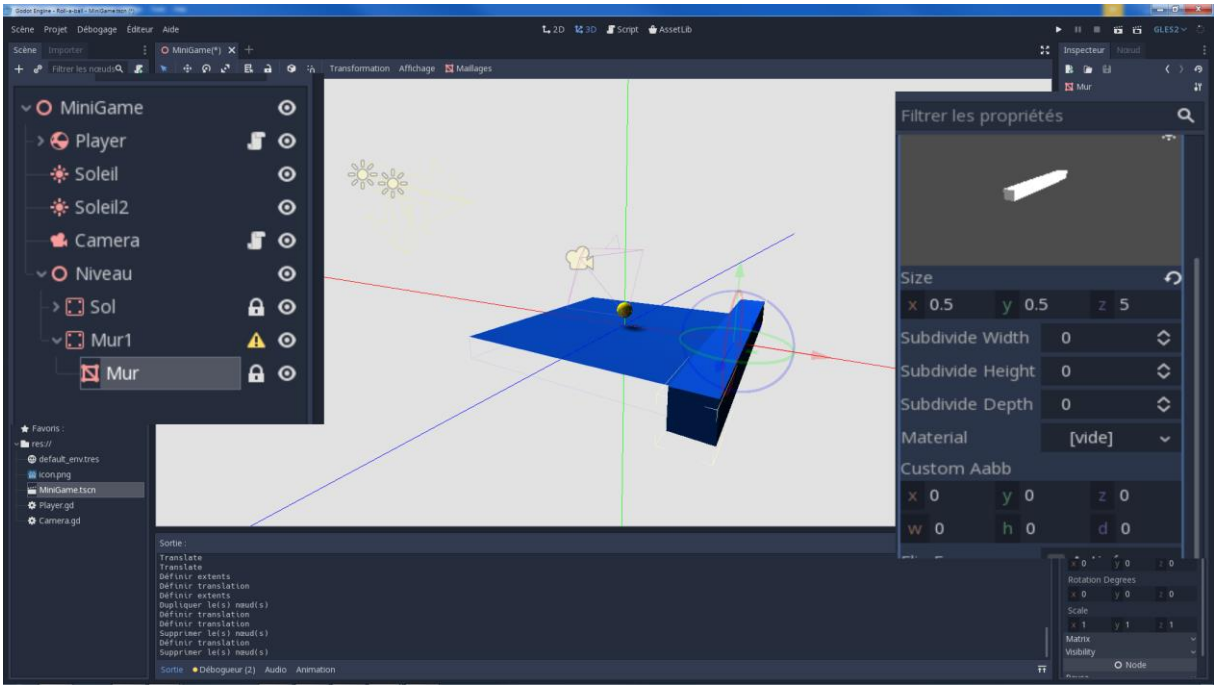
Ajout d’éléments

Ajout de murs

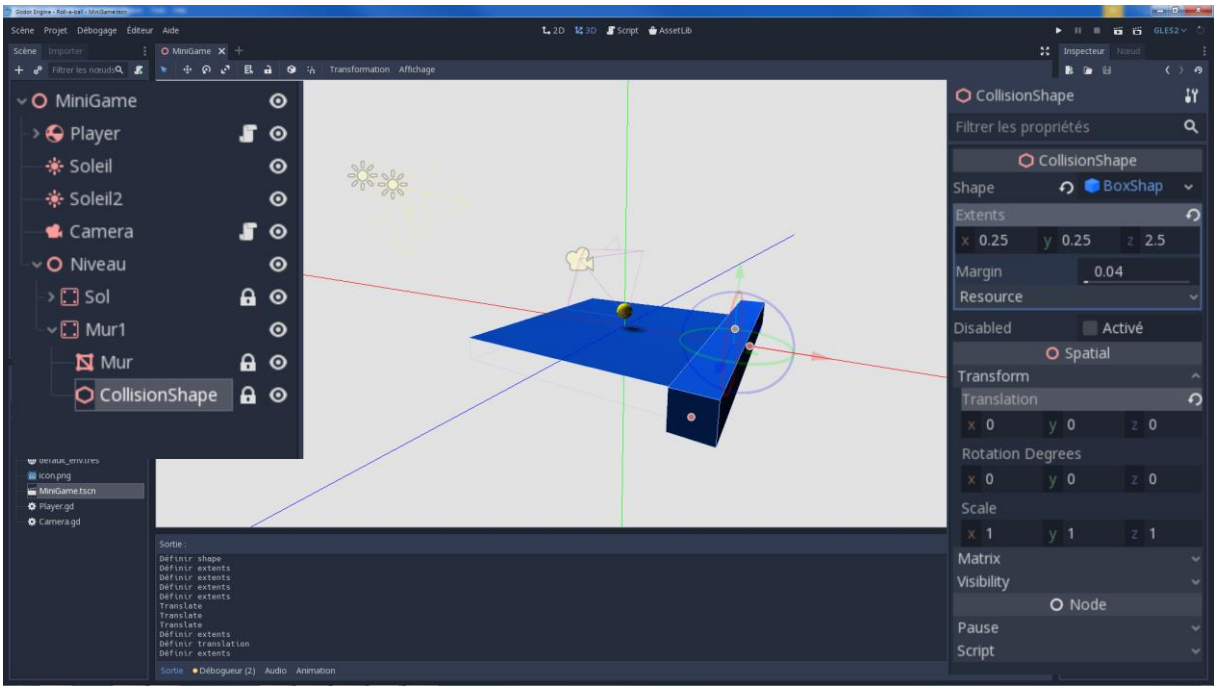
Ajoutons des murs afin d’éviter que notre balle ne tombe du plateau de jeu. Ce seront des cubes, déformés, équipés de boites de collision afin d’empêcher la balle de tomber. On va donc utiliser des nœuds de type StaticBody (comme le sol).

Ajouter un nœud de type Spatial nommé « Niveau » et déplacer le nœud « Sol » existant en tant qu’enfant de ce nouveau nœud (cela afin de conserver une hiérarchie de nœuds à peu près propre).
Créer un nœud de type StaticBody, nommé « Mur1 » en tant qu’enfant du nœud « Niveau ».

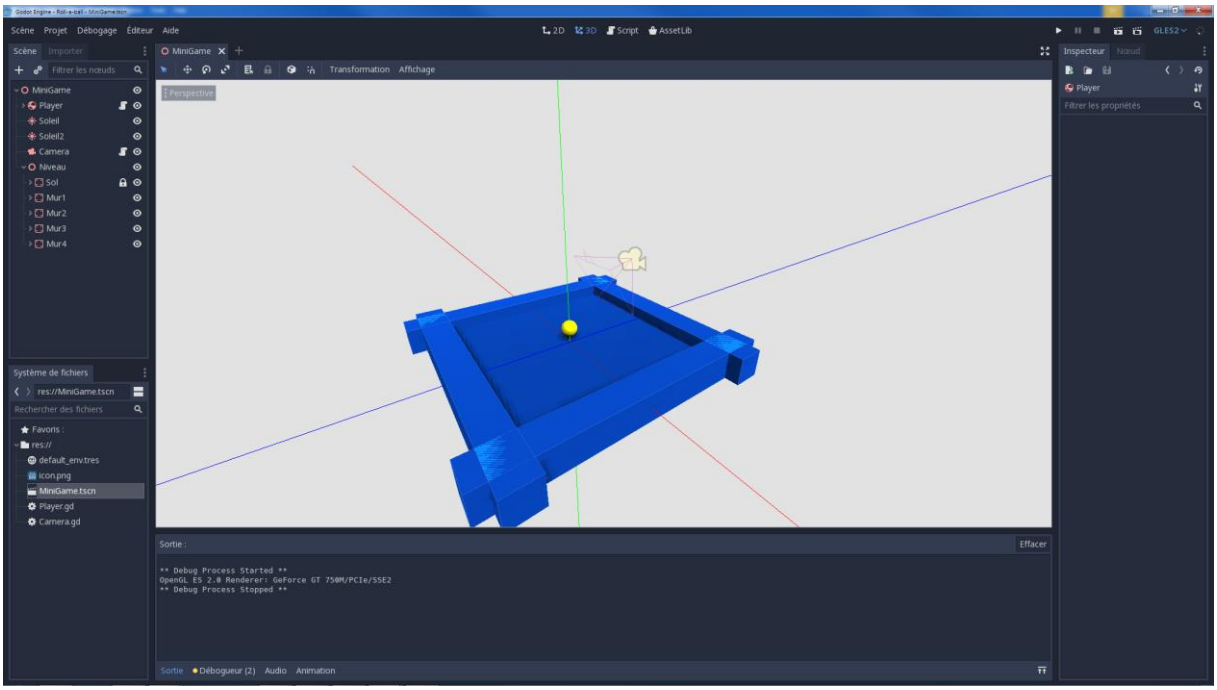
Ajouter un MeshInstance en forme de cube, le déformer et le déplacer pour qu'il devienne l'un des murs du plateau



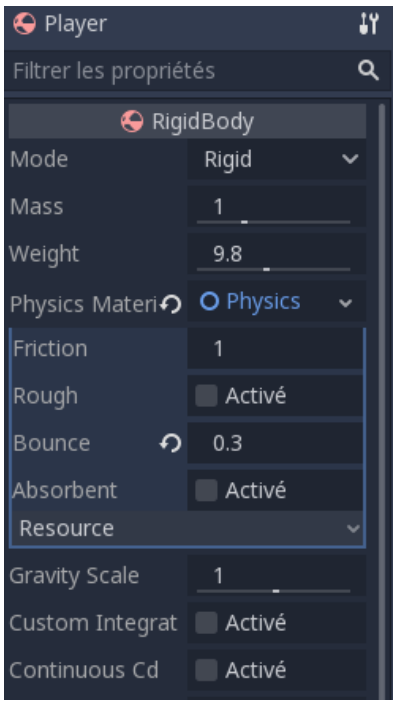
Ajouter à « Mur1 » un nœud de type CollisionShape, de forme « BoxShape » et dont les Extents correspondent aux dimensions du mur. Penser à « cadenasser » la position des nœuds enfants du mur (pour éviter toute fausse manipulation par la suite). Deplacer le nœud « Mur » vers un des bords du plateau.



Dupliquer le mur 3 fois (en utilisant Control+D avec le nœud « Mur sélectionné) et déplacer / tourner les nouveaux murs pour réaliser les 4 murs du plateau.



La balle ne peut maintenant plus sortit du plateau. Dès qu'elle touche un bord, elle est stoppée net.



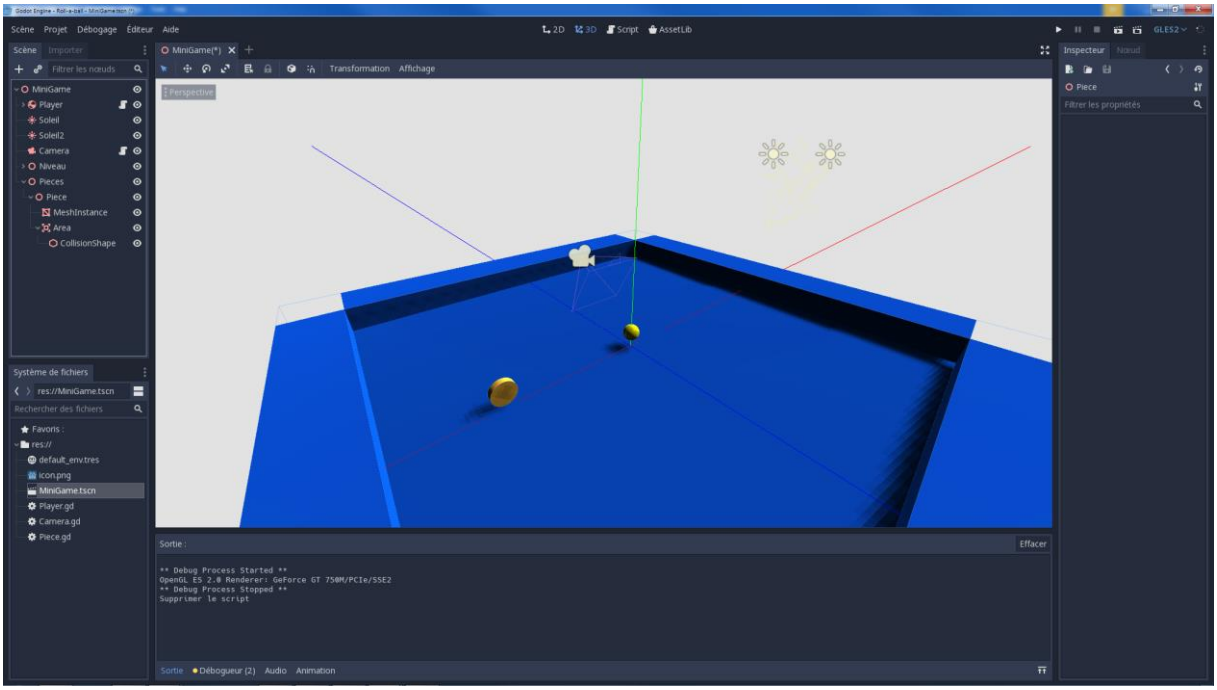
Ce comportement ne fait d'ailleurs pas très réaliste. On peut ajouter un peu de rebond dans cette balle en éditant le nœud « Player », en lui ajoutant un nouveau Physics Material en en précisant un niveau de rebond (Bounce).

Ajout de collectables

On va ajouter quelques élements à collecter sur le plateau.

Si vous avez suivi le tutoriel à la lettre, le plateau se trouve être un peu petit. On double la taille du sol (scale) et on redimensionne et déplace les murs en conséquence (les murs étant des duplicas les uns des autres, on n'a besoin de n'en redimensionner qu'un). Ne pas oublier de modifier les insets des boites de collision des murs.

On va ajouter des pièces sur le plateau. Afin de bien organiser la hiérarchie, on va ajouter un nœud Spatial nommé « Pièces » dans lequel se trouveront toutes les pièces à collecter. On commence par ajouter une première pièce : nœud de type Spatial, contenant un MeshInstance en forme de cylindre (tourné et dimensionné comme il se doit et doté d'un material permettant de le distinguer sur le plateau). Cela permet d'afficher une pièce sur le plateau, mais si on souhaite détecter que la balle passe dessus, il lui faut une boite de collision (mais pas physique…). On ajoute donc à notre pièce un nœud de type Area auquel on attache un nœud de type Collision Shape (de forme cylindrique tournée et dimensionnée de façon à correspondre à la pièce)



Pour rendre le tout plus vivant, on peut attacher un script à la pièce pour la faire tourner :

```
1  extends Spatial
2
3  const vitesse_rotation = PI
4
5  func _process(delta):
6      rotate_y(vitesse_rotation * delta)
7
```

Tester. La pièce est bien sur le plateau, elle tourne, mais la balle la traverse sans que rien ne se passe. Il va falloir détecter le passage de la balle sur la pièce et pour cela, nous allons utiliser les signaux.

Un signal est transmis par un objet à un autre objet, quand il est témoin d'un évènement. Dans notre cas, le nœud Area de notre pièce se rend compte se rend compte qu'il rentre en collision avec quelque chose et va donc en informer la pièce en invoquant une de ses méthodes.

Sélectionner le nœud « Area » de la pièce, et dans l'onglet « Nœud » / « Signaux » faire un clic droit sur « body_entered(Node body) ». Cliquer sur « Connecter » et sélectionner le nœud « Piece ». Valider en cliquant sur « Connecter ».

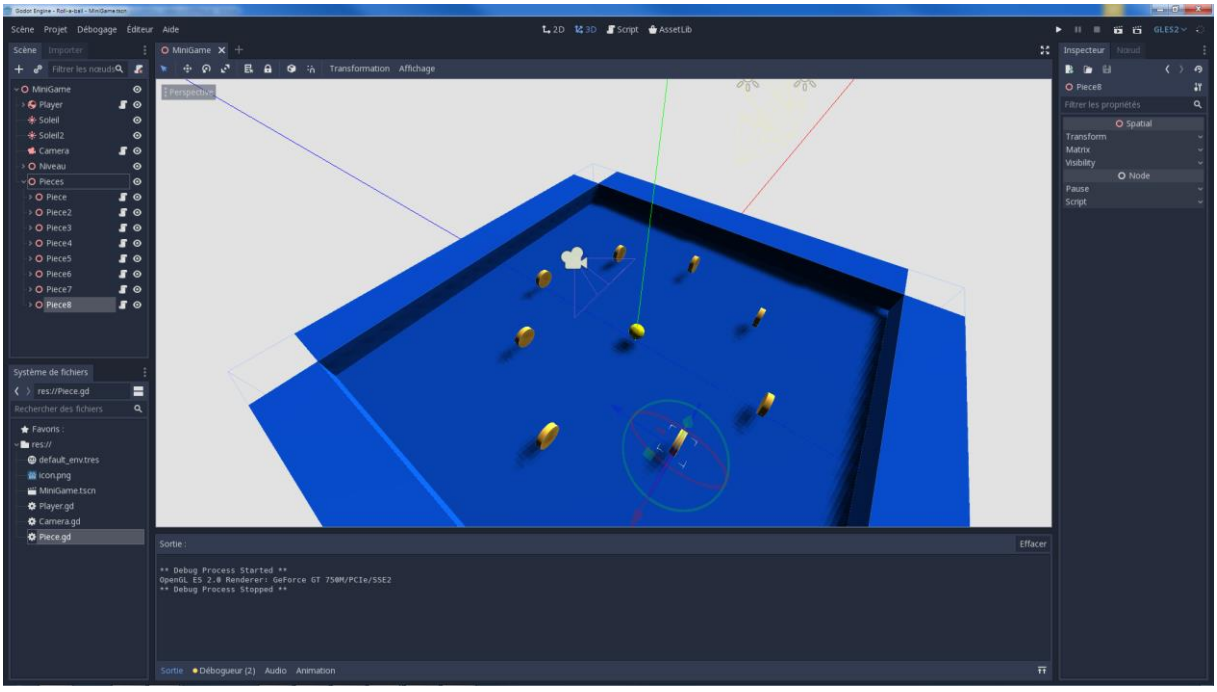
Cela a eu pour effet d'ajouter une fonction dans le script de Piece : _on_Area_body_entered(body). Cette fonction est celle qui sera appela dès qu'un objet entrera dans la boite de collision de la pièce. Le paramètre body est l'objet qui est entré en collision. Dans cette méthode, on va tester qui est entré en collision et s'il s'agit du joueur, on va supprimer la pièce de la scène

```
1  extends Spatial
2
3  const vitesse_rotation = PI
4
5  func _process(delta):
6      rotate_y(vitesse_rotation * delta)
7
8  func _on_Area_body_entered(body):
9      if body.name == "Player" :
10         queue_free()
11
```

queue_free() demande à supprimer l'objet courant dès que possible. Ici, la fonction free() ne fonctionnerait pas car on demanderait à supprimer un objet (la pièce) alors qu'un signal qu'elle gère n'a pas encore été totalement résolu.

Tester. Quand la balle passe sur la pièce, la pièce disparait.

Ajouter plusieurs pièces sur le plateau en dupliquant celle-ci.



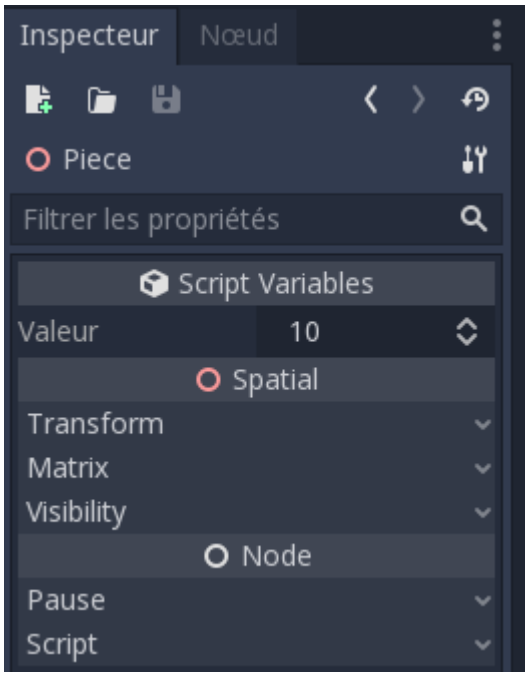
Gestion du score

Mise à jour du score

Ajoutons une gestion de score à notre jeu. Le score sera contenu dans l'objet Player. A chaque fois qu'il ramasse une pièce, sont score est augmenté d'une certaine valeur.
Dans le script Piece.gd, ajouter une variable « valeur » déclarée comme ci-dessous :

```
1  extends Spatial
2
3  export(int) var valeur = 10
4
5  const vitesse_rotation = PI
6
7  func _process(delta):
8      rotate_y(vitesse_rotation * delta)
9
10 func _on_Area_body_entered(body):
11     if body.name == "Player" :
12         queue_free()
13
```

La variable « valeur » a une valeur par défaut de 10, mais le mot clé export utilisé lors de sa déclaration fait que cette valeur peut être modifiée directement depuis l'onglet « Inspecteur » en sélectionnant la pièce :



Ajoutons une variable score au Player :

```
var score = 0
```

et modifions ce score à chaque fois qu'une pièce est ramassée :

```
10 func _on_Area_body_entered(body):
11     if body.name == "Player" :
12         body.score += valeur
13         print(body.score)
14         queue_free()
15
```

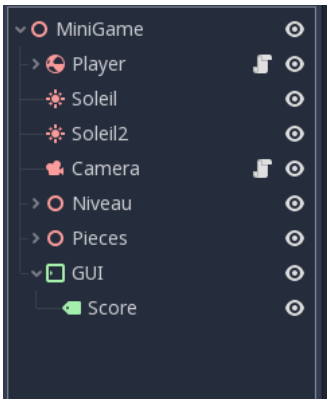
Si on teste, on voit dans la vue « Sortie » le score du joueur qui augmente de 10 à chaque pièce.

Affichage du score

Nous allons utiliser des nœuds de type « Control » pour gérer l'affichage du score.

Une des contraintes de ce tutoriel étant de ne pas nécessiter d'assets, l'affichage du score ne sera pas des plus esthétiques…

Ajouter sous MiniGame un nœud de type MarginContainer. Le nommer GUI. Cliquer sur le bouton « Disposition sur l'écran » se trouvant en haut de la vue du milieu et sélectionner « Full Rect » pour le container prenne la totalité de l'écran. Ajouter à GUI un nœud de type Label nommé « Score ».



On va modifier les scripts de Piece et Player pour rendre la modification du score plus propre (appel d'une fonction de player plutôt que manipulation directe de sa variable score) et pour modifier l'affichage du Label Score :

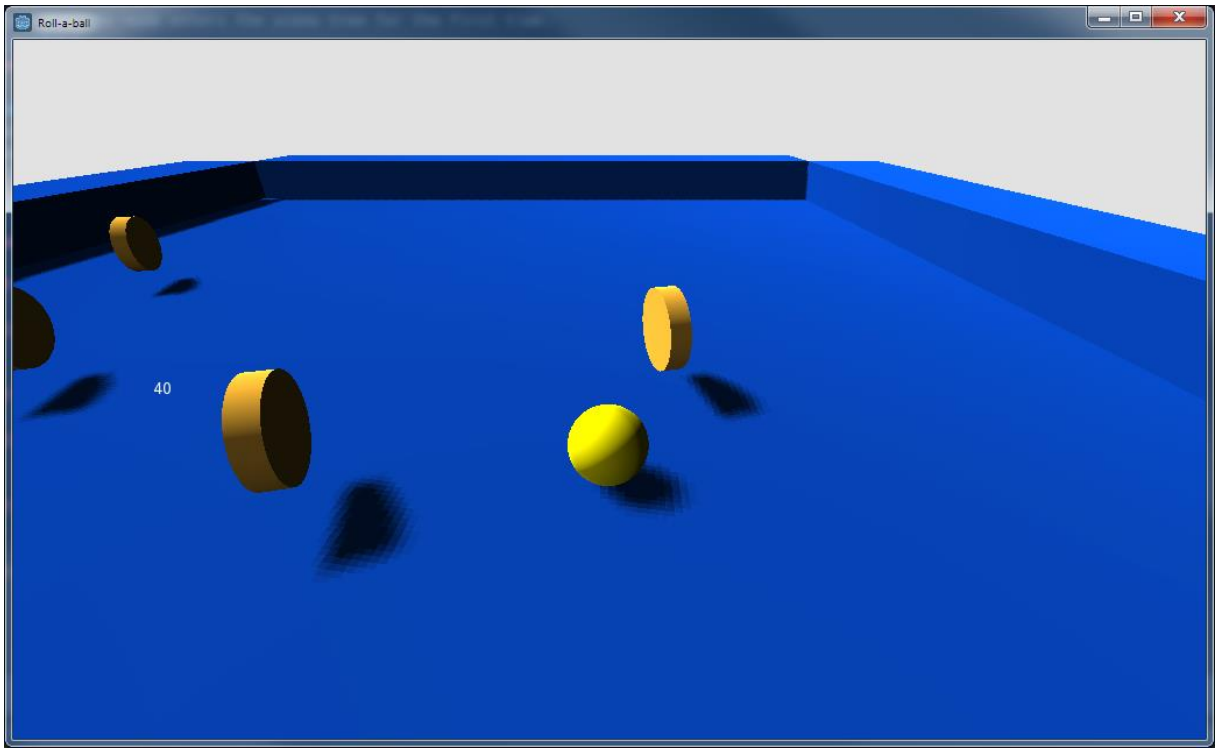
Piece.gd

```
1  extends Spatial
2
3  export(int) var valeur = 10
4
5  const vitesse_rotation = PI
6
7  func _process(delta):
8      rotate_y(vitesse_rotation * delta)
9
10 func _on_Area_body_entered(body):
11     if body.name == "Player" :
12         body.addPoints(valeur)
13         queue_free()
14
```

Player.gd

```
extends RigidBody

const MAX_SQUARED_VELOCITY = 100

onready var scoreLabel = get_node("/root/MiniGame/GUI/Score")

var score = 0

# Called when the node enters the scene tree for the first time.
func _ready():
    pass # Replace with function body.

func addPoints(nbPoints):
    score += nbPoints
    scoreLabel.text = str(score)

func _physics_process(delta):

    var force = Vector3.ZERO

    if Input.is_action_pressed("ui_up") :
        force.z = -1
    elif Input.is_action_pressed("ui_down"):
        force.z = 1

    if Input.is_action_pressed("ui_left") :
        force.x = -1
    elif Input.is_action_pressed("ui_right"):
        force.x = 1

    if force == Vector3.ZERO:
        force = -linear_velocity / 2

    if linear_velocity.length_squared() < MAX_SQUARED_VELOCITY :
        add_force(force, transform.origin)
```

Maintenant, le score s’affiche à l’écran :


Roll-a-ball
40