INF2990

Projet de logiciel graphique interactif

Tests logiciels

Version 16.1

Auteurs : Mohameth Al-Assane Ndiaye
Abdellatif Amrani
Samir El-Kaoukabi
Bilal Itani
Mehdi Kadi
Said Hmani

Suite de cas de test

Suite de cas de test # 1			
Classe testée	NoeudComposite (NoeudCompositeTest)	**Branche**	Master
Justification			
Le patron Composite permet de manipuler des nœuds feuilles ou branches. Les méthodes de la classe NoeudComposite sont à la base de l'arbre de rendu ou comportements associés à chaque nœud formant l'arbre. Indirectement, cette classe est responsable de la validité de l'affichage des objets à l'écran. Ainsi, cette classe mérite d'être testée puisqu'elle peut modifier les propriétés de tout nœud de l'arbre.			
Cas de test # 1			
Méthode testée	unsigned int NoeudComposite::obtenirNombreEnfants() const		
Justification			
Cette méthode doit être testée en un premier lieu. En effet, elle est appelée lors de l'exécution des tests sur d'autres méthodes de la classe NoeudComposite. Il est primordial de la tester, car elle nous permet d'entériner l'exactitude de nos tests pour les autres méthodes de cette classe : on se sert de ses résultats.			
Explication du cas de test			
Le cas de test consiste en un premier temps d'ajouter à l'arbre de rendu un nœud table. Ensuite, on rajoute à ce dernier, deux nœuds poteaux ayant chacun un nœud mur comme enfant. Enfin, on appelle cette méthode sur le nœud table et on vérifie que le nombre d'enfants retourné est bien correct.			
Cas de test # 2			
Méthode testée	void NoeudComposite::vider()		
Justification			
À chaque fois que le programme est lancé, cette méthode est appelée. Cette dernière se charge de vider l'arbre de rendu de tous ses nœuds. Ainsi, l'ajout de nœuds ne peut se faire sur un arbre vide. Cette méthode mérite d'être testée, en effet elle nous permet de s'assurer après création d'une nouvelle zone de simulation, que l'arbre de rendu ne soit composé que d'un nœud table et flèche de départ. Du coup, les nœuds murs et autres ne pourront être présents en trop.			
Explication du cas de test			
Le cas de test consiste à ce qu'un nœud table soit créé puis ajouté à l'arbre. Après, on crée deux murs et deux poteaux qu'on rajoute comme enfants au nœud table créé précédemment. On appelle après la méthode vider sur le nœud table, puis on vérifie que le nombre d'enfants est bien adéquat.			
Cas de test # 3			
Méthode testée	void NoeudComposite::effacer(const NoeudAbstrait* noeud)		

Justification
Cette méthode est très utilisée. Elle permet de supprimer un nœud enfant ou qui est contenu dans un des enfants. Par conséquent, elle mérite d'être testée, puisqu' il est crucial de bien s'assurer que le nœud que l'on veut effacer soit bien retiré, ainsi que ses descendants dans le cas échéant.
Explication du cas de test
On s'assure que le nœud composite n'a pas d'enfants initialement. Ensuite, on ajoute un nœud mur à un nœud poteau. Puis, le nœud poteau est rajouté à notre arbre de rendu. On efface le nœud mur sous enfant de l'arbre puis vérifie qu'il est bien effacé. Finalement, on rajoute un nouveau nœud mur au poteau, puis efface le nœud poteau enfant de l'arbre. Ainsi, ce dernier cas de test valide la nature récursive de la fonction effacer où un enfant et ses fils sont bien effacés.

Cas de test # 4	
Méthode testée	bool NoeudComposite::ajouter(NoeudAbstrait* enfant)
Justification	
Cette méthode doit être testée, car elle permet de vérifier si l'ajout d'un nœud à l'arbre de rendu est un succès ou non. Néanmoins, en cas de disfonctionnalité de cette dernière l'affichage de nœuds ne serait pas possible.	
Explication du cas de test	
Dans ce test, on s'assure que le nœud composite n'a pas d'enfants initialement. Ensuite, on ajoute un nœud mur à l'arbre, puis vérifie que l'ajout est un succès. Enfin, on ajoute trois nœuds à l'arbre de rendu, puis vérifie que le nombre de nœuds rajoutés est correct.	

Cas de test # 5	
Méthode testée	void NoeudComposite::effacerSelection()
Justification	
En mode Édition, cette méthode est appelée lorsqu'un usager désire supprimer des objets de la table. Elle mérite d'être testée, puisqu'elle exécute un algorithme récursif sur l'arbre de rendu afin de supprimer de ce dernier un nœud sélectionné et ses enfants dans le cas échéant. Cette méthode fait appel sur chaque nœud la méthode estSelectionne() qui a déjà été testée préalablement.	
Explication du cas de test	
Le cas de test consiste à vérifier qu'initialement l'arbre est vide. Ensuite, on ajoute respectivement deux murs noeudMur1 et noeudMur2 à un poteau et un mur noeudMur3 à un autre poteau. Puis, on rajoute à l'arbre de rendu les deux poteaux. Par suite, on sélectionne un petit fils de l'arbre de rendu, et on s'assure que les nœuds possèdent le bon nombre d'enfants. Ensuite, on supprime le nœud sélectionné et vérifie que les nœuds ont le bon nombre d'enfants. Dans la suite du test, on sélectionne un nœud enfant de l'arbre et le supprime, du coup la vérification du nombre d'enfants de chaque nœud valide par la même occasion la nature récursive de cette fonction. Enfin, on sélectionne le dernier nœud enfant de l'arbre, puis l'efface. Ainsi notre arbre sera vide.	

Cas de test # 6	
Méthode testée	bool NoeudComposite::estDansLaTableApresDeplacement(glm::dvec3 deplacement)

Justification

Cette méthode mérite d'être testée puisqu'elle est appelée sur chaque nœud lors de l'exécution des déplacements. Elle est très importante, car c'est elle qui permet de savoir si un déplacement est acceptable ou non. Cette méthode exécute un algorithme qui permet de vérifier si un nœud est à l'extérieur de la table ou non.

Explication du cas de test

Le cas de test consiste à ajouter des nœuds à l'arbre de rendu, ensuite on vérifie qu'ils sont sur la table. Par suite, on déplace un nœud enfant à l'extérieur de la table et on s'assure que le nœud est en dehors. Enfin, on positionne un deuxième nœud enfant aux limites de la table, et on s'assure que l'objet est encore sur la table.

Cas de test # 7	
Méthode testée	NoeudAbstrait* NoeudComposite::chercher(const std::string& typeNoeud)

Justification

Cette méthode mérite d'être testée, car elle est utilisée pour valider d'autres tests. Donc, il est prudent de s'assurer que cette méthode est bien fonctionnelle, au risque de biaiser les méthodes qui l'appellent. En plus, elle effectue un algorithme de recherche récursif. Or, l'on sait que la récursivité est propice aux erreurs.

Explication du cas de test

Le cas de test consiste à vérifier qu'initialement le nœud n'a pas d'enfants. Ensuite, on ajoute noeudPoteau2 à un nœud mur. Enfin, on rajoute ce mur puis noeudPoteau1 à l'arbre de rendu. On rappelle qu'on a sélectionné noeudPoteau2 pour le différencier de noeudPoteau1. Finalement, on vérifie que le nœud poteau retourné correspond au premier poteau rajouté à l'arbre, soit noeudPoteau2.

Suite de cas de test # 2			
Classe testée	FacadeModele (FacadeModeleTest)	**Branche**	Master

Justification

La classe FacadeModele est la façade qui sert de lien entre l'interface et la logique du programme. Dans cette classe, la plupart des méthodes portant sur les objets de l'arbre, sont appelées. Ainsi, cette classe ne contient pas beaucoup de logique. Néanmoins, certaines méthodes avec une lourde implémentation lui ont été rajoutées durant la réalisation des livrables 1 et 2. Donc, il est impératif de tester quelqu'unes de ces méthodes. Ainsi, on s'assurera que leur implémentation respective ne nuit pas au bon fonctionnement du programme.

Cas de test # 1	
Méthode testée	void FacadeModele::ajouterNoeudSimple(char* nom, int x, int y)

Justification

Cette méthode est appelée lorsque l'on désire rajouter des nœuds à l'arbre de rendu. La vérification de son fonctionnement est importante, car cette méthode est tout le temps appelée lorsqu'on rajoute des objets en mode Éditeur. On doit donc, s'assurer qu'elle modifie l'arbre de rendu adéquatement.

Explication du cas de test

Le cas de test consiste à vider d'emblée la table et s'assurer qu'elle est vide. Ensuite, on lui ajoute des enfants. Ainsi, un poteau lui est rajouté, puis un mur. Enfin, on vérifie à travers des tests rigoureux à chaque phase d'ajout que le nœud correspondant a été effectivement rajouté à l'arbre.

Cas de test # 2

Méthode testée	bool FacadeModele:: `assignerRotationPourObjetSelectionne` (double angle)

Justification

Cette méthode effectue une rotation sur les objets sélectionnés. Son implémentation mérite d'être testée. En effet, lorsqu'un seul objet est sélectionné, elle tourne ce dernier au tour de son centre. Cette logique de traitement nécessite d'être testée afin de vérifier que la rotation est bien conforme aux requis.

Explication du cas de test

Le cas de test consiste à vider d'emblée la table. Ensuite, on vérifie qu'initialement aucun angle de rotation n'est appliqué au poteau que nous avons rajouté à l'arbre. De plus, on sélectionne le poteau, puis appelle la fonction `assignerRotationPourObjetSelectionne` afin de vérifier si le poteau est tourné. Enfin, on désélectionne le poteau, puis rappelle la fonction `assignerRotationPourObjetSelectionne` et nous assurer que l'objet reste à l'ancienne position.

Suite de cas de test # 3

Classe testée	NoeudPoteau (NoeudPoteauTest)	**Branche**	Master

Justification

Cette classe mérite d'être testée, puisqu'elle est responsable de la sélection d'un nœudPoteau. D'ailleurs, son bon fonctionnement doit être garanti, sans quoi la sélection d'un poteau deviendrait non fonctionnelle. Donc, on ne pourrait couvrir un grand nombre de fonctionnalités pour un nœud poteau.

Cas de test # 1

Méthode testée	void NoeudPoteau::changerScale(double facteur)

Justification

Cette méthode effectue une mise à échelle sur les poteaux sélectionnés. Dès lors, il serait pertinent de la tester pour vérifier si les requis du rapport d'élicitation sont respectés. En d'autres termes, il suffit de vérifier si la mise à échelle sur un poteau s'est fait sur son rayon uniquement.

Explication du cas de test

Le cas de test consiste à vérifier si les dimensions initiales du noeudPoteau_ sont respectées. Ensuite, on effectue une mise à échelle de FACTEUR sur le noeudPoteau_. Enfin, on s'assure du bon résultat de la mise à échelle en x, y et z. Autrement dit, on s'assure de la mise à échelle du rayon et de la conservation de la hauteur.

Cas de test # 2

Méthode testée	bool NoeudPoteau::estDansLaTable()

Justification

Cette méthode mérite d'être testée puisqu'elle est appelée sur un nœud poteau, pour bien s'assurer de sa présence sur la table. Elle est très importante, car c'est elle qui permet de savoir si un poteau déborde ou non. D'ailleurs, l'expérience nous a montré que le simple fait que le chargé de laboratoire ait réussi à faire déborder un nœud poteau sur notre scène lors du livrable 1, plusieurs de nos fonctionnalités sont devenues subitement dysfonctionnelles.

Explication du cas de test

On s'assure que le nœud est initialement sur la table. Ensuite, on le déplace au coin supérieur droit de la table. Puis, on vérifie que le poteau est sur la table. Enfin, on fait déborder de peu notre poteau et puis on vérifie qu'il est bien en dehors de la table.

Suite de cas de test # 4

Classe testée	AideCollision (AideCollisionTest)	**Branche**	Master

Justification

Cette classe mérite d'être testée, puisqu'elle implémente des méthodes appelées lors des collisions. Ainsi, on détectera aisément le point de contact entre objets. Malgré, que ses méthodes nous aient été fournies, nous avons jugé nécessaire d'en tester quelques-unes afin de confirmer leur bon fonctionnement. Ainsi, nous serons plus en mesure de bâtir des méthodes relatives aux collisions entre un robot et un mur ou un poteau fiables.

Cas de test # 1

Méthode testée	bool AideCollision::intersectionAvecSegment(const glm::dvec2& point1, const glm::dvec2& point2)

Justification

Cette méthode est appelée pour détecter s'il y a une intersection entre deux segments. Dans notre projet, les segments sont les lignes formant les pourtours des capteurs ou murs. Cette méthode nous a été fournie à la base. D'ailleurs, il est prudent de bien nous assurer de son bon fonctionnement puisqu'elle est appelée constamment.

Explication du cas de test

Le cas de test consiste à former des segments et à vérifier s'il y a une intersection entre eux. D'abord, nous positionnons deux segments de façon perpendiculaire, puis nous vérifions qu'ils se coupent. Enfin, nous vérifions qu'il ne peut y avoir d'intersection entre deux segments parallèles.

Cas de test # 2

Méthode testée	DetailsCollision calculerCollisionSegment(const glm::dvec2& point1,const glm::dvec2& point2,const glm::dvec2& position, double rayon, bool collisionAvecPoints = true)

Justification
Cette méthode mérite d'être testée puisqu'elle est appelée afin de détecter la collision entre les poteaux et les capteurs de distance du robot. Cette méthode doit être fonctionnelle, en effet elle participe à l'élaboration du suivi de comportements du robot. Étant donné, que nous n'avons pas implémenté cette méthode, il est crucial de nous assurer qu'elle produit les résultats escomptés.
Explication du cas de test
Le cas de teste consiste à vérifier s'il existe une collision entre un segment et un objet circulaire, en particulier un poteau. D'abord, nous simulons une absence de collision, puis une collision entre un segment et un cercle où le point de contact serait le premier point du segment. Enfin, nous simulons une collision entre un segment et un cercle où l'intersection est le segment.

Cas de test # 3	
Méthode testée	DetailsCollision calculerCollisionCercle(const glm::dvec2& centreCercle, Double rayonCercle, const glm::dvec2& positionObjet, double rayonObjet)

Justification
Cette méthode mérite d'être testée puisqu'elle est appelée afin de détecter la collision entre deux cercles. Cette méthode doit être fonctionnelle, en effet elle participe à l'élaboration du suivi de comportements du robot. Étant donné, que nous n'avons pas implémenté cette méthode, il est crucial de nous assurer qu'elle produit les bons effets.
Explication du cas de test
Le cas de teste consiste à vérifier s'il existe une collision entre deux objets circulaires. D'abord, nous simulons une absence de collision, puis nous montrons la subtilité de la collision entre deux cercles tangents, autrement dit il n'est pas réaliste d'appesantir nos efforts sur ce cas en pratique. Enfin, nous simulons une collision entre deux cercles sécants.

Suite de cas de test # 5			
Classe testée	NoeudRobot (NoeudRobotTest)	**Branche**	Master

Justification
Cette classe mérite vraiment d'être testée, puisqu'elle implémente les méthodes de l'objet qui est le cœur du livrable 2, soit l'objet robot. D'ailleurs, le robot se déplace, et réagit à son environnement grâce à ce qu'il perçoit avec ses capteurs. Ainsi, il est prudent, sinon logique de s'assurer que les méthodes coordonneesParRapportAOrigine ou obtenirNombreSenseursActifs soient fonctionnelles. En effet, elles sont constamment appelées pour bien s'assurer du bon fonctionnement du robot tout au long d'une simulation. Ces méthodes sont extrêmement prioritaires.

Cas de test # 1	
Méthode testée	glm::dvec2 NoeudRobot::coordonneesParRapportAOrigine(glm::dvec2 point, glm::dvec2 positionOrigine, double theta) const

Justification
Cette méthode est un algorithme qui permet de connaître la position d'un point donné initialement sur un référentiel (qui est en relation avec le référentiel de base dont le centre

est l'origine de la table) dans le référentiel de base. Ainsi, cette méthode nous permet de récupérer la position des capteurs du suiveur de ligne ou de distance qui sont à la base par rapport au référentiel du robot, dans le référentiel de la table. En outre, cette méthode devrait bien fonctionner, si nous voulons assurer de bonnes collisions entre le robot et les poteaux (ou murs) ou avoir de façon précise des détections de lignes par le biais du suiveur de ligne.

Explication du cas de test

Le cas de test consiste à obtenir les coordonnées d'un point donné dans le référentiel du robot par exemple, dans le référentiel de la table. D'abord, nous effectuons un test avec un angle thêta égal à zéro degré. Du coup, on s'assure de la bonne fonction de l'addition entre vecteurs. Enfin, nous vérifions que la méthode donne de bons résultats lorsque le référentiel du robot est tourné d'un certain angle par rapport au repère de base. Nous avons calculé les résultats escomptés à la main, puis avons vérifié si elles concordaient avec le déroulement de la méthode.

Cas de test # 2

Méthode testée	int NoeudRobot::obtenirNombreSenseursActifs()

Justification

Cette méthode permet de déterminer le nombre de senseurs actifs du suiveur de ligne. Dès lors, il est important de la tester afin de s'assurer que son implémentation est correcte. D'ailleurs, sa bonne implémentation garantit un parfait suivi de ligne.

Explication du cas de test

Le cas de test consiste à activer et désactiver les senseurs puis de vérifier si le nombre de senseurs actifs est correct. Ainsi, on vérifie initialement qu'aucun senseur n'est actif. Enfin, on évalue le nombre de senseurs actifs à travers un jeu d'activation ou de désactivation des senseurs.

Suite de cas de test # 6

Classe testée	ProjectionOrtho (ProjectionOrthoTest)	**Branche**	master

Justification

Nous testons ici une projection orthogonale. En effet, cette classe constitue l'interface de projection définie dans la classe Projection. Du coup, elle implante entre autres, certaines fonctionnalités propres à une projection orthogonale, telles que le zoom autour d'un point ou le redimensionnement de la fenêtre graphique.

Cas de test # 1

Méthode testée	`void ProjectionOrtho::redimensionnerFenetre(const glm::ivec2& coinMin, const glm::ivec2& coinMax)`

Justification

Il est primordial de tester cette méthode, puisque le rapport d'aspect des objets affichés à l'écran doit être conservé lorsqu'un usager redimensionne la fenêtre graphique comme il l'entend. Nous testons l'algorithme qui permet de conserver la relation de proportionnalité entre l'agrandissement de la fenêtre virtuelle par rapport à l'agrandissement de la clôture. Ainsi, les objets garderont la même grandeur apparente lorsque la fenêtre graphique est redimensionnée.

Explication du cas de test
On s'assure que le rapport d'aspect est initialement conservé. Donc, cela stipule que nos conditions initiales sont satisfaisantes. Ensuite, on assigne un redimensionnement non nul à la fenêtre et on s'assurera que les coordonnées de la fenêtre virtuelle en fonction du redimensionnement de la fenêtre sont correctement ajustées. Autrement dit, le rapport d'aspect des objets est conservé.
Cas de test # 2

Méthode testée	`void ProjectionOrtho::zoom()`

Justification
Il est impératif de tester cette méthode puisque le zoom, en particulier avant, permet de mieux scruter le rendu des objets affichés à l'écran. Dès lors, on peut mieux examiner d'éventuels débordements des objets déposés sur la table, grâce à l'effet de loupe. Nous testons l'algorithme qui permet de conserver la relation de proportionnalité entre la modification de la région visible du monde virtuel par rapport à sa taille. En outre, nous testons entre autre, l'effet selon lequel l'application d'un nombre égal de zooms avant et arrière ne modifie pas la taille de la région visible.
Explication du cas de test
On s'assure qu'effectivement aucun zoom n'est initialement appliqué. Donc, cela stipule que le facteur de zoom est unitaire. Ensuite, on effectue un zoomIn et on s'assurera que les coordonnées supérieures en X et Y de la fenêtre virtuelle ont été rapetissées. Puis, inversement on vérifiera que les bornes inférieures en X et Y de la fenêtre ont été agrandies. Enfin, on s'assurera que le facteur de zoom est bien incrementZoomTest_. Ainsi, cela montre une bonne relation de proportionnalité entre la modification de la région visible par rapport à sa taille. En toute fin, on applique un nombre égal de zooms avant et arrière. Autrement dit, le facteur zoom deviendrait unitaire.
Suite de cas de test # 7

Classe testée	Utilitaire (UtilitaireTest)	**Branche**	Master

Justification
Nous testons ici la classe Utilitaire. En effet, la structure de l'arbre dépend dans sa version de base du projet Utilitaire. Cette classe contient entre autres, une foule de petites fonctions très intéressantes. Par exemple, elle implante entre autres, certaines fonctionnalités comme testEGAL_ZERO() pour vérifier la nullité d'un nombre ou testDANS_LIMITESXY() qui s'assure de l'appartenance d'un point à un carré.
Cas de test # 1

Méthode testée	bool EGAL_ZERO(double nombre)

Justification
Il est crucial de tester cette méthode, puisque la nullité pure n'est pas toujours garantie. Néanmoins, il est important de s'assurer des marges permises quant à l'acception du nombre zéro en projet 2. D'ailleurs, notons que cette définition de zéro est utilisée dans une pléthore de classes, par exemple FacadeModele ou Plan3D.
Explication du cas de test

On s'assure que la nullité pure est correctement définie. Ensuite, on s'assure de la bonne fonctionnalité de la valeur limite EPSILON définie dans cette classe. Enfin, on vérifie jusqu'à quel point un nombre peut être considéré comme nul.	
Cas de test # 2	
Méthode testée	bool DANS_LIMITESXY(double x, double xMin, double xMax, double y, double yMin, double yMax)
Justification	
Il est important de tester cette méthode, puisqu'elle permet de savoir si un point est contenu dans un carré. D'ailleurs, elle est appelée dans la classe FacadeModele, afin de tester si le curseur de la souris se trouve effectivement sur la table. En outre, elle est usitée dans VisiteurVerifierDeplacementDuplication. En effet, il est impératif de bien vérifier si les objets dupliqués seront à l'intérieur ou non de la zone de dessin.	
Explication du cas de test	
On s'assure que le point arbitraire que nous avons défini est contenu dans le carré que nous avons spécifié. Ensuite, on s'assure de l'échec du test au cas où l'abscisse ou ordonnée du point initialement défini se trouve à l'extérieur de notre carré.	