Toutes les classes et les tests sont présents sur la branche master. Cependant, nous avons laissé la branche d'origine dans ce document à titre de référence.

Suite de cas de test # 1			
Classe testée	Droite3D (Droite3DTest)	**Branche**	f15_physique
Justification			
L'objet Droite3D est très utile pour le calcul de collisions. Il s'agit d'une droite en 3 dimensions au sens mathématique, c'est-à-dire infinie. On ne l'utilise concrètement que pour 2 dimensions cependant, soit x et y. Pour déterminer si une collision entre le robot et un poteau a lieu, on calcule la droite perpendiculaire à un segment de la boîte englobante du robot passant par le centre du poteau, et s'il y a intersection à l'intérieur des limites du cylindre, il y a collision. Pour déterminer si une collision entre le robot et un mur a lieu, on calcule les droites passant par leurs segments, et s'il y a intersection à l'intérieur des intervalles des segments, il y a collision. La grande importance de la classe Droite3D pour la physique demande qu'elle soit bien comprise et testée.			

Cas de test # 1	
Méthode testée	Droite3D::lireVecteur (testVecteur)
Justification	
Le vecteur directeur d'une droite est utilisé pour déterminer la pente de son équation linéaire. Il est aussi utilisé pour facilement calculer un second point faisant partie de la droite à partir d'un premier. Puisqu'il est important d'obtenir la pente d'une droite pour les calculs d'intersection, on doit s'assurer que le vecteur directeur de la droite est bon.	
Explication du cas de test	
Le test est simple : calculons manuellement le vecteur directeur non normalisé de la droite, égal à la différence vectorielle entre les deux points utilisés pour générer la droite. Si le vecteur retourné est le même, le calcul est celui attendu. Afin de renforcer le test, on utilise deux différentes droites.	
Cas de test # 2	
Méthode testée	Droite3D::intersectionDroite (testIntersection)
Justification	
L'intersection (un point dans l'espace) entre deux droites est un élément des plus essentiels dans le calcul de collisions. Puisque chaque segment du robot, chaque mur, chaque droite perpendiculaire passant par le centre d'un poteau peut être représenté par une droite, et que deux droites non parallèles dans un plan génèrent une intersection, on désire utiliser cette intersection souvent et s'assurer de sa fiabilité.	
Explication du cas de test	
On prend les deux droites utilisées au test précédent et on calcule l'intersection entre elles. Si on convertit leurs vecteurs directeurs en pentes, on peut calculer l'intersection avec les pentes et coordonnées à l'origine (y(x=0)). Si l'intersection est celle attendue, alors le calcul est bon.	

Suite de cas de test # 2			
Classe testée	CollisionTool (CollisionToolTest)	**Branche**	f15_physique
Justification			
Le coeur de la physique des collisions est dans la classe CollisionTool, qui est en fait un visiteur. Lorsque le robot bouge, il vérifie s'il entre en collision avec un objet sur la table. Pour cela, des calculs d'algèbre linéaire sont sans cesse effectués. Entre autres, on désire tester la rotation d'un point A autour d'un centre O, puisqu'il s'agit d'un calcul de complexité plus élevée. On voudra vérifier que la longueur d'un vecteur est bonne, puisqu'elle est facilement testable et souvent utilisée.			

Cas de test # 1	
Méthode testée	CollisionTool::rotate (testRotate)
Justification	
Afin de déterminer en tout temps les coordonnées de la boîte englobante du robot en mouvement, on doit appliquer sur les points critiques une série de transformations algébriques (matricielles). La plus complexe est la rotation des points critiques autour du centre relatif du robot. La rotation peut être complexe à calculer, mais très importante, alors il importe de la tester sous plusieurs conditions : à l'origine ou pas, à angles rectangles et surtout à autres angles.	
Explication du cas de test	
On considère deux tests : un d'angle rectangle autour de l'origine; un d'un angle arbitraire autour d'un point arbitraire. Le calcul matriciel est le suivant : $X' = R * (X – X0) + X0$. Puisqu'il s'agit de calculs arrondis par l'ordinateur sur des doubles, il importe de vérifier un très faible intervalle autour des valeurs cibles, et non la valeur précise. Dans le second cas, on choisit tout de même des valeurs qui donneront un résultat non entier, mais facile à arrondir.	
Cas de test # 2	
Méthode testée	CollisionTool::length (testLength)
Justification	
Afin de déterminer si une intersection veut réellement dire une collision, puisque les droites sont en réalité infinies, on compare la longueur du vecteur d'impact avec la longueur du segment cible. Si le vecteur est plus grand, il dépasse l'intervalle du segment de la droite, et l'intersection ne représente pas une collision. Il importe donc de mesurer une bonne longueur d'un vecteur, ou d'un segment qui est en réalité un vecteur résultant de la différence vectorielle entre deux points.	
Explication du cas de test	
Le test est très simple : déterminer la longueur d'un vecteur arbitraire et la comparer avec la longueur retournée par la fonction. On s'intéresse à deux cas : un vecteur non nul et un vecteur nul. La longueur est déterminée par $\|\|v\|\| = sqrt(x^2 + y^2 + z^2)$.	

Suite de cas de test # 3			
Classe testée	utilitaire (UtilitaireTest)	**Branche**	master
Justification			
L'espace utilitaire fournit plusieurs fonctions et références très utiles que l'on utilise partout dans le programme. Il définit entre autres la constante pi, la différence entre deux nombres réels (epsilon), la conversion d'angles entre radians et degrés, la validation d'un nombre dans un intervalle, et permet également de vérifier si un nombre réel est « égal » à zéro, à une approximation minimale près. Ce sont ces dernières fonctions que l'on désire tester, puisqu'on les utilise non seulement à plusieurs reprises dans le code principal du programme, mais aussi dans les tests unitaires. Afin de rendre les tests unitaires valides, il est très important de s'assurer que les fonctions utilitaires soient testées et fonctionnelles.			

Cas de test # 1	
Méthode testée	utilitaire ::RAD_TO_DEG (testRad2deg)
Justification	
Les fonctions trigonométriques utilisent des angles en radians, mais OpenGL utilise des angles en degrés. La conversion de radians en degrés est donc essentielle pour transformer les angles calculés en angles utilisables par OpenGL.	
Explication du cas de test	
On effectue 4 tests : un angle nul, un angle positif, un angle négatif, et un angle dépassant 2π radians. Dans tous les cas, on s'attend à calculer un angle résultant de façon linéaire, sans optimisation. Par exemple, 2π rad = 360°, et non 0°.	
Cas de test # 2	
Méthode testée	utilitaire::DEG_TO_RAD (testDeg2rad)
Justification	
Les fonctions trigonométriques utilisent des angles en radians, mais OpenGL utilise des angles en degrés. La conversion de degrés en radians est donc essentielle pour transformer les angles d'OpenGL en angles utilisables par les fonctions trigonométriques.	
Explication du cas de test	
On effectue 4 tests : un angle nul, un angle positif, un angle négatif, et un angle dépassant 360 degrés. Dans tous les cas, on s'attend à calculer un angle résultant de façon linéaire, sans optimisation. Par exemple, 360° = 2π rad, et non 0 rad.	
Cas de test # 3	
Méthode testée	utilitaire::EGAL_ZERO (testEgalZero)
Justification	
Puisque des opérations mathématiques par ordinateur sur des nombres réels peuvent causer des erreurs résiduelles ou d'approximation, un nombre supposé être nul théoriquement pourrait avoir une petite valeur, par exemple x = 0,000000000000001. Dans ce cas, on ne peut utiliser l'expression x == 0, car bien qu'elle devrait être vraie, elle ne le sera pas. La méthode EGAL_ZERO est donc nécessaire pour déterminer si un nombre réel est virtuellement nul.	
Explication du cas de test	
On s'intéresse à 5 cas pour valider le test. Un nombre réel clairement non nul, un petit nombre et le cas limite de détection de valeur nulle d'un réel doivent être calculés comme non nuls; un nombre parfaitement nul et un nombre inférieur à la limite doivent être calculés comme nuls.	

Cas de test # 4	
Méthode testée	utilitaire::DANS_INTERVALLE (testIntervalle)

Justification

Pour la même raison que dans le test précédent, un nombre réel sensé être égal à une valeur peut ne pas y être parfaitement égal, par exemple x = 6.0000000000000001 ≠ 6.0. ÉGAL_ZERO ne s'applique que dans le cas où la valeur attendue est à 0.0, mais dans le cas d'un autre nombre, on utilise DANS_INTERVALLE ainsi que EPSILON pour effectuer un traitement similaire.

Explication du cas de test

On doit d'abord s'assurer que DANS_INTERVALLE détermine bien si un nombre est ou n'est pas dans un intervalle $x1 \leq y \leq x2$. Ensuite, on vérifie qu'un nombre y est dans l'intervalle $y - \varepsilon \leq y \leq y + \varepsilon$ où ε est la limite de détection pour qu'un réel soit non nul.

Suite de cas de test # 4			
Classe testée	CenterTool (CenterToolTest)	**Branche**	f3.2_visitor

Justification

Avec l'outil Rotation, on peut pivoter un nœud autour de son axe central (sur lui-même). Si plusieurs nœuds sont sélectionnés, ils doivent alors pivoter autour de leur centre commun. Il existe plusieurs méthodes afin de calculer ce centre, mais dans notre cas d'objets de masse virtuellement identique, on ne s'intéresse qu'au centre géométrique. Il est donc raisonnable de tester notre méthode utilisée sous différents cas afin de la valider et d'assurer une bonne rotation.

Cas de test # 1	
Méthode testée	CenterTool::getCenter (testCenter)

Justification

Cette méthode est la raison d'être de la classe CenterTool. C'est elle qui retournera le centre calculé après le traitement. Elle est destinée à être appelée après avoir visité les nœuds de la table, et n'intervient pas elle-même dans le traitement des nœuds, mais on peut l'utiliser afin de valider un cas secondaire spécifique du traitement effectué par visit.

Explication du cas de test

Dans les cas principaux, on veut confirmer que le centre d'aucun nœud n'explose pas (donc qu'il est nul), et que le centre des nœuds sélectionnés est égal à la somme vectorielle de chaque position, puis chaque composante est divisée par le nombre de nœuds traités.
Dans le cas secondaire, on utilise la méthode getCenter pour confirmer que des nœuds non sélectionnés ne sont pas traités.

Suite de cas de test # 5			
Classe testée	ProjectionOrtho (ProjectionOrthoTest)	**Branche**	master
Justification			
Cette classe gère la projection orthographique et est essentielle pour la transformation de la vue. Elle gère le rapport d'aspect, le redimensionnement de la fenêtre, le zoom. Elle modifie les valeurs de fenêtre et de clôture.			

Cas de test # 1	
Méthode testée	ProjectionOrtho::zoomerIn() (testZoomIn)
Justification	
Il est intéressant de regarder les nouvelles grandeurs de la projection orthographique après un agrandissement pour vérifier la bonne incrémentation de celles-ci. Ainsi, les nouvelles valeurs d'agrandissement conservent le centre de la vue en ajoutant la moitié de l'incrément aux quatre côtés de la fenêtre.	
Explication du cas de test	
On donne des valeurs bidons à la fenêtre d'une projection orthographique et on effectue un appel à zoomerIn. On vérifie alors si la valeur du zoom est bien égale à l'ancienne valeur plus l'incrément du zoom.	
Cas de test # 2	
Méthode testée	ProjectionOrtho::zoomerIn(glm::dvec3, glm::dvec3) (testZoomInRec)
Justification	
La fonction zoomerIn avec un rectangle élastique effectue deux traitements. Premièrement, elle affecte les nouvelles valeurs de fenêtre à partir des dimensions obtenues du coin min et coin max en paramètres. On regarde si l'affectation change bien la dimension de notre fenêtre. Deuxièmement, on teste la fonction privée ajuster qui modifie la clôture pour conserver le rapport d'aspect. Il est essentiel que le rapport d'aspect soit conservé pour conserver l'échelle de la vue et qu'elle ne soit pas déformée.	
Explication du cas de test	
On assigne de nouvelles valeurs à la fenêtre à partir des paramètres d'un rectangle élastique. On teste alors les nouvelles valeurs pour vérifier si le rapport d'aspect est bien conservé.	
Cas de test # 3	
Méthode testée	ProjectionOrtho::redimentionnerFenetre (testRedimentionnementDeLaFenetre)
Justification	
Lors d'un redimensionnement de la fenêtre, il est primordial que les dimensions s'ajustent en fonction du redimensionnement pour conserver l'échelle et le rapport d'aspect.	
Explication du cas de test	
On redimensionne la fenêtre avec deux nouvelles valeurs et on acquiert les dimensions de clôture et de fenêtre et on teste si le rapport est le même pour les deux.	

Suite de cas de test # 6			
Classes testées	NoeudCylindre, NoeudMur, NoeudSegmentConcret (NoeudAbstraitTest)	**Branche**	develop

Justification
Ce test est différent des autres, en ce sens où on teste une méthode de NoeudAbstrait surchargée dans trois différentes classes dérivant de NoeudAbstrait. Puisque ces classes implémentent leur propre version de la méthode setScale, il est intéressant de valider la valider dans chaque classe. Il ne fait aucun doute que la classe NoeudAbstrait est une des plus pertinentes pour les tests, cependant elle est déjà testée en partie. On y ajoutera donc un nouveau test sur les enfants de la classe.

Cas de test # 1	
Méthode testée	NoeudCylindre::setScale, NoeudMur::setScale, NoeudSegmentConcret::setScale (testScale)

Justification
La méthode setScale est importante puisque la mise à l'échelle est une des transformations à effectuer sur les nœuds en mode édition, c'est-à-dire modifier l'échelle des nœuds, qui ont des restrictions différentes. Il est d'autant plus pertinent de tester setScale suite à notre évaluation du premier livrable, car il était possible d'élargir un segment de ligne alors qu'il n'aurait pas dû être possible. Si nous avions fait ce test unitaire plus tôt, nous aurions découvert ce problème et n'aurions pas perdu de points au premier livrable.

Explication du cas de test
Pour un poteau, l'échelle s'applique sur son rayon. On désire donc s'assurer qu'un changement d'échelle n'affectera pas la composante en z, et que les valeurs d'échelle en x ainsi qu'en y soient égales à la nouvelle valeur en x. Pour un mur, l'échelle s'applique sur sa longueur. On désire donc s'assurer qu'un changement d'échelle n'affectera ni la composante en z ni celle en x (largeur), mais uniquement celle en y (longueur). Pour une ligne, l'échelle n'est pas modifiable. On désire donc s'assurer qu'un changement d'échelle n'affectera pas les propriétés d'un segment de ligne.

Suite de cas de test # 7			
Classe testée	NoeudComposite (NoeudCompositeTest)	**Branche**	develop
Justification			
La classe NoeudComposite est elle aussi une classe de nœud particulièrement importante, car l'arbre de rendu en hérite. Elle utilise le patron composite, donc plusieurs méthodes sur le composite affectent tous ses enfants. Il est important de tester efficacement une classe ayant des répercussions sur plusieurs autres.			

Cas de test # 1	
Méthode testée	NoeudComposite ::obtenirNombreEnfants (testObtenirNombreEnfants)
Justification	
Puisque cette méthode est largement utilisée dans les autres tests sur l'ajout et la suppression de nœuds dans un composite, il importe de la tester afin d'isoler les problèmes potentiels. Une méthode non testée ne peut servir à valider un autre test.	
Explication du cas de test	
On crée 5 nœuds, dont 3 sont immédiatement ajoutés comme enfants d'un composite, et on s'assure qu'il possède 3 enfants. On lui ajoute les 2 nœuds restants, et on s'assure qu'il en possède maintenant 5.	
Cas de test # 2	
Méthode testée	NoeudComposite::vider (testVider)
Justification	
La méthode vider est utilisée à chaque initialisation ou réinitialisation de l'arbre de rendu. Il est nécessaire de confirmer qu'un nœud composite vidé est réellement vide et ne comporte aucun nœud résiduel, sans quoi il serait possible d'avoir plusieurs robots sur la table suite à plusieurs chargements successifs de zones de sauvegarde.	
Explication du cas de test	
Le test est très simple : ajoutons arbitrairement trois nœuds comme enfants d'un composite, assurons-nous qu'il contienne trois enfants avant d'être vidé, et aucun après.	
Cas de test # 3	
Méthode testée	NoeudComposite::effacer (testEffacer)
Justification	
La méthode effacer est plus précise que vider ou effacerSelection,	
Explication du cas de test	
On crée 5 nœuds, dont un est composite. On les ajoute tous comme enfants sous un composite, à l'exception d'un nœud terminal. Il devrait avoir alors 4 enfants. On efface ensuite un enfant, et on s'assure qu'il a désormais 3 enfants. Dans un second cas, on tente d'effacer le nœud orphelin, et on s'assure que le composite possède toujours 3 enfants, puisqu'il ne peut supprimer un nœud dont il n'est pas le parent. Dans un troisième cas, on efface l'enfant composite du nœud composite, et on s'assure que ce dernier possède alors 2 enfants. Dans le cas final, on procède à l'effacement des 2 enfants restants, et on s'assure qu'à la fin il ne contienne aucun enfant.	
Cas de test # 4	
Méthode testée	NoeudComposite::effacerSelection (testEffacerSelection)
Justification	
Il est pertinent de tester cette méthode pour deux raisons principales : d'abord parce qu'elle utilise une logique qui peut sembler peu intuitive avec la STL, et parce que les tests ont fait leurs preuves. Nous avons trouvé un problème dans la méthode grâce aux tests. On veut s'assurer que seuls les nœuds sélectionnés sont supprimés par cette méthode; pas plus, pas moins.	

Explication du cas de test
Afin de couvrir toutes les bases et de compenser pour le fait qu'une fois la possession d'un nœud transférée au composite, il ne soit plus possible de modifier le nœud terminal original, on utilise une logique en deux étapes. On ajoute d'abord 3 nœuds non sélectionnés au composite, et on vérifie qu'après l'appel à effacerSelection il possède toujours 3 enfants. On ajoute ensuite 2 nœuds dont un est sélectionné, et on vérifie qu'après l'appel à effacerSelection il possède désormais 4 enfants.

Cas de test # 5	
Méthode testée	NoeudComposite::chercher(const std::string&) (testChercher)
Justification	
La méthode chercher fournie dans le cadriciel du projet est utile dans le cas où on veut obtenir un nœud unique ou instancié une seule fois. En effet, bien qu'il soit important de pouvoir chercher un type de nœud spécifique, tel que la table dans un arbre de rendu, cette méthode retourne le premier nœud trouvé du type cherché. Elle est donc très peu utile avec les nœuds instanciés plusieurs fois. Il est donc intéressant d'étudier son comportement plus en profondeur.	
Explication du cas de test	
On ajoute deux nœuds du même type dans un nœud composite, et on s'attend à ce que le nœud trouvé par la recherche de ce type de nœud retourne le premier nœud inséré dans le composite, et non le second.	
Cas de test # 6	
Méthode testée	NoeudComposite ::ajouter (testAjouter)
Justification	
La méthode ajouter a déjà été testée pour la classe NoeudAbstrait, mais son implémentation dans la classe NoeudComposite est différente. En effet, cette fois-ci l'ajout doit fonctionner, sans quoi la logique composite ne serait pas fonctionnelle et l'arbre de rendu ne serait rien.	
Explication du cas de test	
On vérifie qu'un ajout d'un nœud à un composite retourne un résultat positif, signifiant que l'ajout s'est bien effectué, et que le composite contienne un enfant de plus.	
Cas de test # 7	
Méthode testée	NoeudComposite ::selectionnerTout (testSelectionnerTout)
Justification	
La méthode selectionnerTout est utile pour l'utilisateur ou pour effectuer un traitement, valide uniquement sur les nœuds sélectionnés, sur tous les enfants valides. Cependant, puisque certains nœuds peuvent être impossibles à sélectionner, on désire confirmer que ces derniers ne puissent pas être accidentellement sélectionnés. La fonction d'assignation de sélection est déjà testée pour un NoeudAbstrait, mais en testant selectionnerTout d'un NoeudComposite, on s'assure que la sélection d'un nœud est valide et qu'aucune erreur ne soit possible tant de la part du composite que du nœud cible.	
Explication du cas de test	
On crée 3 nœuds, dont 2 sont sélectionnables et 1 ne l'est pas, que l'on assigne comme enfants à un composite. Après avoir appelé selectionnerTout, on vérifie que les nœuds sélectionnables sont tous sélectionnés, mais que le nœud non sélectionnable n’est pas sélectionné.	

Suite de cas de test # 8			
Classe testée	FacadeModele (FacadeModeleTest)	**Branche**	master

Justification

Notre FacadeModele est l'intermédiaire entre notre FacadeInterfaceNative et toutes nos classes principales du C++. Cependant, il est difficile d'effectuer des tests unitaires pour cette classe, puisque peu de méthodes sont unitaires. On peut tout de même tester une méthode unitaire, ce qui nous aidera également à confirmer le bon comportement général de la classe.

Cas de test # 1	
Méthode testée	FacadeModele::isOnTable (isOnTableTest)

Justification

Cette méthode détermine si un nœud est entièrement compris dans les limites de la table. Elle est importante dans le mode édition, et comprend un cas limite intéressant. On veut la tester afin de s'assurer qu'un nœud qui n'est pas entièrement compris à l'intérieur des limites de la table (si une partie dépasse) soit considéré comme n'étant pas sur la table.

Explication du cas de test

On s'intéresse à trois cas. D'abord, on instancie une table de taille connue et on place un nœud sur la table, à une position connue comme étant valide sur la table, et on vérifie le résultat de la méthode. Ensuite, on déplace le nœud complètement hors de la table, et on vérifie le résultat. Finalement, on place le nœud sur le bord de la table de sorte qu'il soit majoritairement sur la table, mais qu'une partie dépasse, et on vérifie le résultat, en s'attendant à un résultat négatif.