

INF2990

Projet de logiciel graphique interactif

Tests logiciels

Version 16.3

**Fait par : Équipe 11**

10 Novembre 2016

|  |  |  |  |
| --- | --- | --- | --- |
| **cas de test # 1,23,4,5,6** | | | |
| **Classe testée** | NoeudComposite (NoeudCompositeTest) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| Comme une grande partie de notre architecture se base sur les nœuds composites et ses méthodes, donc l’idée de tester cette classe peut être pertinent.  On se base dans cette classe de test sur la classe « nœudTable » puisque la table est l’objet qui va contenir le plus d’objets dans l’application. | | | |

|  |  |
| --- | --- |
| **Cas de test # 1** | |
| **Méthode testée** | NoeudComposite::vider (testVider()) |
|  | |
| **Justification** | |
| Il est primordial d’initialiser la scène de jeu à l’ouverture de notre application, à travers la méthode « initialiser » qui fait appel à son tour à vider () du nœud Composite.  La méthode « vider » permet donc d’effacer tous les nœuds fils d’un nœud composite (Nœud Table dans notre cas). | |
|  | |
| **Explication du cas de test** | |
| Dans ce test, on ajoute un nœud enfant (un bonus accélérateur) au nœudTable, on assure que le nombre d’enfant de la table est correct (1 dans notre cas), après on appelle la méthode vider () et revérifie encore le nombre des enfants (doit être 0) | |
|  | |
| **Cas de test # 2** | |
| **Méthode testée** | NoeudComposite::ajouter (testAjout()) |
|  | |
| **Justification** | |
| L’ajout des objets est un élément essentiel pour le fonctionnement de l’application, puisque la plupart des fonctionnalités manipulent des objets dans la scène. Donc on a jugé qu’il sera une fonctionnalité pertinente à tester. | |
| **Explication du cas de test** | |
| On prend le nœud Table comme père, on vérifie d’abord qu’il n’a pas d’enfants, on crée ensuite deux nœuds Muret et Bonus et on fait appel à la méthode « ajouter ». Finalement on vérifie que le nœud Table a le bon nombre de descendance. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test # 3** | |
| **Méthode testée** | NoeudComposite::effacerSelection (testSuppression()) |
|  | |
| **Justification** | |
| Puisqu’on a testé l’ajout des objets, il est intéressant de vérifier si la suppression des objets est bien fonctionnelle, on a pris on compte aussi le fait qu’il n’est pas possible de supprimer un seul portail. | |
| **Explication du cas de test** | |
| On commence par ajouter un noeudBonus et une paire des portails, on vérifie que le nombre des enfants du Table est exacte, puis on sélectionne un seul portail et on appelle la méthode effacerSelection.  On vérifier finalement, que la fonction n’a enlever que le portail sélectionné et son frère. | |
|  | |
| **Cas de test # 4** | |
| **Méthode testée** | NoeudComposite::Chercher (testTrouverObjet();) |
|  | |
| **Justification** | |
| Vu le nombre illimité de nœuds fils qu’on peut avoir, le test de la méthode Chercher (2 surcharges : indice et type) sera nécessaire.  Cette méthode peut être appeler en passant comme paramètres l’indice ou le type du nœud cherché et retourne le nœud en question s’il est trouvé. | |
|  | |
| **Explication du cas de test** | |
| On commence par ajouter trois nœuds au nœud Table, on vérifie que le nombre d’enfants est correct, puis on appelle la fonction chercher avec le type du nœud comme paramètre (bonus par exemple) et l’élément retournée par cette fonction correspond à l’élément crée au début.  De même avec Chercher (avec indice comme paramètre).  On a testé aussi le cas où l’indice ne correspond à aucun élément (out of range) | |
|  | |
| **Cas de test # 5** | |
| **Méthode testée** | NoeudComposite::selectionnerTout(testSelectionMultiple()) |
|  | |
| **Justification** | |
| Avec un grand nombre des enfants, il sera plus efficace d’appeler une méthode qui fait sélectionner tous les enfants du nœud composite au lieu de parcourir l’arbre et faire la sélection de chaque nœud. | |
|  | |
| **Explication du cas de test** | |
| On insère deux éléments au nœud Table, on appelle la fonction selectionnerTout sur le nœud père puis on vérifie que les trois nœuds (table + 2 fils) sont sélectionnés. | |
|  | |

|  |  |
| --- | --- |
| **Cas de test # 6** | |
| **Méthode testée** | NoeudComposite::setScale (testScale()) |
|  | |
| **Justification** | |
| Une des fonctionnalités du mode édition est la mise en échelle des objets, cette fonctionnalité permet d’agrandir ou diminuer un objet. | |
|  | |
| **Explication du cas de test** | |
| On crée un objet Bonus qui a initialement le scale (1,1,1), on a agrandi et on vérifie la valeur de Scale (on essaie aussi diminuer l’objet) | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cas de test # 7** | | | |
| **Classe testée** | PointsControle(PointControleTest) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| Cette classe permet de modifier la position des points de contrôle, il permet de sélectionner un point de contrôle et modifier sa position, puis, on modifier la position du point de contrôle symétrique. | | | |

|  |  |
| --- | --- |
| **Cas de test # 7** | |
| **Méthode testée** | NoeudTable::getPc(testPositionPoint()) |
|  | |
| **Justification** | |
| La position des points de contrôle se change en paire, c’est-à-dire, on chaque fois qu’on modifie la position d’un nœud, son nœud opposé doit changer sa position en conséquence. | |
|  | |
| **Explication du cas de test** | |
| On crée un nœud table avec ses 8 points de contrôle, puis on prend la position du point de contrôle (haut-gauche) et et on vérifie que le point de contrôle (haut-droite) a les mêmes coordonnées en y et z, et x inversé. | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cas de test # 8, 9** | | | |
| **Classe testée** | VisiteurCollision(CollisionTest) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| Toute la physique du jeu est basée sur l’utilisation de le visiteur VisiteurCollision ; il est donc important de garantir que ses fonctions de detection des colisions sont fonctionnelles. | | | |

|  |  |
| --- | --- |
| **Cas de test # 8** | |
| **Méthode testée** | VisiteurCollision::collisionSegment(testCollisionSegment) |
|  | |
| **Justification** | |
| Il s’agit de la méthode de test de collision de base, qui vérifie si un objet est en collision avec un ensemble de segments continus : elle est primordiale pour tester les collisions des maillets et rondelle avec les objets rectangulaires et les parois de la table. | |
|  | |
| **Explication du cas de test** | |
| On créé un ensemble de segments formant un rectangle puis place l’objet du visiteur (dont on teste les collisions) a différents endroits en attendant ou pas la détection d’une collision. | |
| **Cas de test # 9** | |
| **Méthode testée** | VisiteurCollision::visiterNoeudCercle(testvisiterNoeudCercle) |
|  | |
| **Justification** | |
| Cette fonction est responsable de la détection des collision entre deux objets circulaires (rondelle/maillet, rondelle/portail, etc), vérifier des résultats corrects est primordiale pour la physique du jeu. | |
|  | |
| **Explication du cas de test** | |
| On crait un objet portail puis vérifie sa collision avec un objet fictif (le visiteur a seulement besoin d’une position et d’un rayon pour ce test), avec des différentes positions et mises à échelle. | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cas de test # 10,11, 12** | | | |
| **Classe testée** | Utilitaire(Utilitaire Test) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| La rotation des objets, la vérification de leur position comme étant sur la table, et d’autres fonctionnalités du livrable dépendent des fonctions du namespace utilitaire. | | | |

|  |  |
| --- | --- |
| **Cas de test # 10** | |
| **Méthode testée** | Utilitaire:: CalculerAngle3D (testCalculerAngle3D) |
|  | |
| **Justification** | |
| Il s’agit de la méthode permettant de calculer l’angle formé par trois points ; la fonction testée plus tard (MdansTriangleABC) est basée sur cette fonction. | |
|  | |
| **Explication du cas de test** | |
| On créé un ensemble de points dont on teste les angles connus d’avance. | |
| **Cas de test # 11** | |
| **Méthode testée** | Utilitaire::MdansTriangleABC(testMdansTriangleABC) |
|  | |
| **Justification** | |
| Cette fonction est permet la vérification d’un point comme étant à l’intérieur d’un triangle (ABC) ; elle est utilisé notamment pour vérifier si les objets sont dans la table. | |
|  | |
| **Explication du cas de test** | |
| On créé un ensemble de points qu’on utilise pour tester l’appartenance à un triangle formé de points issus du même ensemble. | |
| **Cas de test # 12** | |
| **Méthode testée** | Utilitaire::rotater (testRotater) |
|  | |
| **Justification** | |
| Cette fonction est utilisée notamment pour la rotation des objets et différentes manipulations. | |
|  | |
| **Explication du cas de test** | |
| On créé un ensemble de points qu’on utilise pour le résultat de différentes rotations. | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cas de test # 13** | | | |
| **Classe testée** | ProjectionOrtho (RedimensionnementTest) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| On test dans cette classe le redimensionnement de la fenetre | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cas de test # 13** | | | |
| **Méthode testée** | ProjectionOrtho::redimensionnerFenetre(testFenetre) | | |
|  | | | |
| **Justification** | | | |
| Cette méthode ajuste la fenêtre virtuelle par apport aux dimensions de la clôture afin de garder le rapport d’aspect. Par conséquence, les objets de la scène ne se déforment pas et conservent leur taille, malgré que la taille de la fenêtre a changé . | | | |
|  | | | |
| **Explication du cas de test** | | | |
| Pour tester cette méthode, on a assigné des valeurs à la clôture. Puis, on a calculé à la main la nouvelle dimension de la fenêtre virtuelle et on l’a comparer avec les valeurs retourner par cette fonction. | | | |
| **Cas de test # 14 , 15** | | | |
| **Classe testée** | Camera (CameraTest) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| Le but de ce cas de test, est verifier le deplacement du camera selon les axes X, Y, et Z. | | | |
| **Cas de test # 14** | | | |
| **Méthode testée** | Camera ::deplacerXY(testDeplacerXY) | | |
|  | | | |
| **Justification** | | | |
| La position de La caméra est un paramètre très important pour la fonction Atlook qui positionne la vue de l’observateur. C’est la caméra n’est pas bien placer et déplacer, cela affecte la façon comment la table et les autres nœuds vont être vue (cercle déformé ) par exemple. | | | |
|  | | | |
| **Explication du cas de test** | | | |
| Dans ce test, on teste le bon déplacement de la caméra orthogonale selon l’axe XY. Donc, on assigne une nouvelle position à la caméra selon X et y à travers cette méthode. Ensuite, on utilise obtenirPosition() pour avoir la nouvelle. La comparaison des coordonnées nouvelles avec celles calculer à partir de sa position initiale en leur ajoutant les déplacements permet de tester si elle est bien déplacée. | | | |
| **Cas de test # 15** | | | |
| **Méthode testée** | Camera::deplacerZ(testDeplacerXY) | | |
|  | | | |
| **Justification** | | | |
| Le déplacement selon Z est aussi important comme le déplacement selon XY. Les même raisons déterminantes la sélection le choix de test de déplacement de la caméra selon l’axe XY ont incité le choix de déplacament de la caméra selon Z. | | | |
|  | | | |
| **Explication du cas de test** | | | |
| Ici on teste le bon fonctionnement de déplacement de la caméra orthogonale selon l’axe XY. Donc, on assigne une nouvelle position à la caméra selon Z à travers cette méthode. Ensuite, on utilise obtenirPosition() pour avoir la nouvelle. La comparaison de la nouvelle coordonnée Z avec celle calculer à partir de sa position initiale en leur ajoutant le déplacement en Z, permet de tester si elle est bien déplacée. | | | |
| **Cas de test # 16, 17, 18** | | | |
| **Classe testée** | Tournoi<std ::string>(TournoiTest) | **Branche** | Master |
|  | | | |
| **Justification** | | | |
| Le tournoi fait partie intégrante de l’application. C’est ce pourquoi une bonne structure sur laquelle s’appuyer est importante. Ici, un template a été utilisé afin de gérer les possibles changements d’idée au niveau de la structure de donnée qui représente le joueur | | | |

|  |  |
| --- | --- |
| **Cas de test # 16** | |
| **Méthode testée** | Tournoi<std ::string>:: obtenirProchainMatchup () |
|  | |
| **Justification** | |
| Permet de tester si on obtient le bon matchup dans un tournoi à travers la structure. Ainsi, on est certain que l’ordre d’obtention sera le bon. | |
|  | |
| **Explication du cas de test** | |
| Un tournoi est créé avec une liste de participants (ici des noms). Si on obtiens les deux derniers participants (romain, andre) dans le bon ordre, alors on devrait bien obtenir les autres matchups | |

|  |  |
| --- | --- |
| **Cas de test # 17** | |
| **Méthode testée** | Tournoi<std ::string>:: affecterScoreProchainMatchup () |
|  | |
| **Justification** | |
| Pour gagner un tournoi, il faut une progression à travers la structure. Pour se faire, ce cas de test nous permet de tester si, en affectant un score au matchup obtenu par obtenirProchainMatchup, le gagnant obtenu est le bon. | |
|  | |
| **Explication du cas de test** | |
| Un tournoi est créé avec une liste de participants (ici des noms). Si, en affectant des scores aux différents matchups, le gagnant qui en ressort est le bon. On réexécute le tout avec les deux matchups restants (l’autre demi-finale et la finale). | |

|  |  |
| --- | --- |
| **Cas de test # 18** | |
| **Méthode testée** | Tournoi<std ::string>:: obtenirMatchups () |
|  | |
| **Justification** | |
| Pour afficher, le tournoi à l’écran, il est nécessaire de connaître l’ensemble des noms et scores des participants. Pour se faire, on utilise la méthode obtenirMatchups. | |
|  | |
| **Explication du cas de test** | |
| Un tournoi est créé avec une liste de participants (ici des noms). On affecte des scores à tous les matchups et on vérifie si les noms concordent avec celui du joueur sensé y être présent et si le joueur n’est pas le gagnant, on vérifie s’il a le bon score d’associé. | |