Rapport de TER

19 mai 2018

Table des matières

Première partie

Présentation du projet

Chapitre 1

Introduction

1.1 But du projet

L'objectif de ce projet est la réalisation d'un jeu basé sur un modèle multi-agent. L'idée générale du projet est dans la continuité de celui de l'année dernière et sur le même thème. L'outil utilisé est Unity 3D, un moteur de jeu employé dans un grand nombre de réalisations de hautes qualités. Notre projet est opérationnel sur Windows et pourrait être porté sur Mac ou encore Android. Ce projet consiste de réaliser un jeu que l'on peut qualifier de programmeur et de permettre, notamment, à de jeunes personnes de se familiariser avec le monde de la programmation. L'utilisateur pourra donc créer un comportement pour des robots appelés "unité" afin de remplir des objectifs du jeu.

Metabot est un projet modeste réalisé à partir du logiciel Unity 3D par un groupe d'étudiants débutants dans l'utilisation de cet outil. Malgré le peu d'expérience dans la création pure de ce genre d'applications, le projet actuel est le fruit d'un travail important et d'une implication entière de toute l'équipe. Il a donc pour unique prétention de communiquer notre amour du jeu vidéo et de la programmation.

1.2 Cahier des charges

Notre but était de modifier le programme existant afin qu'il devienne plus générique. C'est à dire qu'il ne se limite plus au simple jeu Warbot mais qu'il permette l'implémentation de différents jeux orientées agents facilement. Pour cela il fallait donc généraliser la gestion des différentes unités et de leur comportement (action/perception/statistique). De plus, l'ajout d'actions et de moyens de perceptions doit se faire de façon intuitive de même pour les différentes exigences de jeu (règles/conditions de victoire).

1.3 Notion d'agent

1.3.1 Système multi-agents

1.3.2 Representation dans notre projet

Chapitre 2

WarBot : Le mode par défaut

2.1 Principe

Dans WarBot, deux à quatre équipes se battent sur un terrain pour les ressources afin de survivre et d'éliminer les autres équipes et d'être la dernière ne vie. Des ressources apparaissent sur la carte et peuvent être converties en unités ou en soin.

Deuxième partie

Réalisation du projet

Chapitre 3

Partie "Moteur"

3.1 État de l'art de l'ancien projet.

Mehdi

3.2 Refonte du noyau.

L'ancien projet est une adaptation du jeu Warbot crée en java en utilisant la librairie MadKit, permettant la conception et la simulation de système multi-agents. En utilisant comme base ce projet, et en utilisant la hiérarchie de classe proposé dans le code java dans un moteur de jeu relativement bien assisté comme Unity, de nombreux problèmes de conceptions peuvent apparaître.

Unity, pour rappel, est un moteur de jeu développé par Unity Technologies. Ce logiciel à la particularité d'etre "orienter assets". Les scripts associé à chaque objet (appellée GameObject) dérive de la classe "MonoBehaviour", ce qui permet d'avoir accées à un ensemble de méthodes et d'attribut nécessaire à la création de comportement et d'interaction.

Dans cette section, nous allons développer les problèmes que nous avons rencontrer dans la réalisation d'un moteur de jeu générique sur la base de l'ancien projet et de l'explication de la nécessiter de recréer un moteur à partir de zéro.

Nous avons donc remarqué des problèmes du faite de ce choix de conception. Tout abords les scripts en eux-même ne sont pas adaptés au développement d'un programme sur Unity et surtout ne sont pas générique. En effet, les unités sont codés en dur dans le code, ne permettant pas l'ajout de nouvelles unités de façon simple. Ainsi pour rajouter de nouvelles unités, il faut créer de nouvelles classes correspondant a de nouvelles unités. Cette hiérarchie des classes est un parti-pris que l'équpe de l'année passée à choisi en se basant sur le code java de WarBot. Cependant, la problématique de notre sujet de TER ne nous permet pas d'avoir ce genre de conception dans l'idée de rendre l'ajout des unités plus simple. Ainsi, il fallait reprendre la conception des unités et voir les composants que l'on peut garder de façon unique et modulable pour toute les unités.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using System;
```

```
5  using WarBotEngine.Projectiles;
6  using System.Security.Cryptography;
7
8  namespace WarBotEngine.WarBots {
9
10     public class AttackController : MonoBehaviour {
11 [...]
12         public bool Reloaded()
13         {
14             return (this.reloading + this.reload_time < Time.time);
15         }
16
17         public bool Shoot()
18         {
19             if (!this.Reloaded())
20                 return false;
21             this.Fire();
22             return true;
23         }
24 [...]
25         public void Fire() {
26             SoundManager.Actual.PlayFire(this.gameObject);
27             Instantiate(this.projectile , this.warprojectile_emitter.transform.
       position , this.warprojectile_emitter.transform.rotation , this.transform);
28             Instantiate(this.muzzle_flash , this.warprojectile_emitter.transform.
       position , this.warprojectile_emitter.transform.rotation , this.transform);
29             this.reloading = Time.time;
30         }
31     }
32 }
```

Listing 3.1 – Code du script AttackController.cs de l'ancien projet

La gestion des actions et des perceptions aussi est problématique. Le code de ces actions sont codés en dur dans des scripts correspondant a des unités d'un certain type, empêchant alors de rendre générique le fait de rajouter des actions sans devoir modifier tout les codes nécessitant le nom des actions tel que l'interpréteur. Le code ci-dessous montre le script "AttackController.cs" de l'ancien projet. Dans ce code, l'actions Shoot depend des fonction Reloaded et Fire. Reloaded peut être considérer comme une perception. Le fait que les actions et les perceptions ne sont pas clairement définie entant que tels posent des problèmes de compréhension du code et de faite le rajout de façon simple une nouvelle action et une nouvelle perception. De plus, cela pose le problème de la gestion des actions et des perceptions de l'interpréteur, car pour chaque action créer dans le script, il faut indiquer à l'interpreteur quelle fonction permet d'activer l'action du comportement en cours.

Ainsi dans le code du listing 3.2, on peut voir que pour activer la fonction Shoot() du script AttackController, il faut créer une fonction du même nom, puis renseigner toutes les unités qui peuvent effectuer l'action. Le problème est qu'il faut, pour chaque action, créer une fonction dans le fichier Unit.cs, et ainsi le fichier Unit.cs est surchargé, atteignant les 2000 lignes de codes !

```
1  using ...;
2  namespace WarBotEngine.Editeur
3  {
4      [...]
5      public class Unit
6      {
7          /// <summary>
```

```
8           /// The unit shoot a projectile if is reloading
9           /// </summary>
10          /// <returns>Return true if action success and false otherwise</returns>
11          [PrimitiveType(PRIMITVE_TYPE.ACTION)]
12          [UnitAllowed(WarBots.BotType.WarHeavy)]
13          [UnitAllowed(WarBots.BotType.WarTurret)]
14          [PrimitiveDescription("Fait tirer l'unité (termine l'action si réussi)")]
15          public bool Shoot()
16          {
17              return this.agent.GetComponent<WarBots.AttackController>().Shoot();
18          }
19      }
20  }
```

Listing 3.2 – Extrait du code du script Unit.cs

La suite de l'examen du code de l'ancien projet et avec l'accord de notre encadrant en exposant les problèmes qu'engendre la reprise de l'ancien moteur, nous avons décider de recréer un nouveau moteur de jeu pour repartir sur des bases plus générique.

3.3 Réalisation

3.3.1 Retour aux bases.

Mehdi

3.3.2 Intégration dans le projet.

La phase d'intégration.

Maxime

Problèmes liés à l'intégration.

Maxime

Résolution des problèmes.

Maxime

3.3.3 Améliorations des fonctionnalités.

Optimisation du temps de chargement

L'un des gros problèmes que l'on a rencontré est le fait que le temps de chargement lors d'une partie était beaucoup trop élevé et cela nuisait au bon déroulement de la partie, atteignant parfois deux minutes de temps de chargement lorsque l'on voulait générer 200 unités. Pour ce la nous avons étudier plusieurs pistes pour résoudre ce problème

La gestion des actions des unités. La premier piste de recherche que l'on a étudié est d'essayer d'optimiser la gestion des actions des unités. Lorsque l'on a conçu les unités, elle faisaient leurs actions dans une fonction utilisé par Unity "Update" qui est appelé à chaque calcul d'image. Ainsi pour chaque unités présente sur la scène, Unity calcule l'action que l'unité doit faire. Cependant, certaine action nécessite plus de temps en fonction de sa complexité, ce qui pouvait engendré que des unités avaient des tours de retard par rapport à d'autres unités.
Pour palier à ce problème, nous avons donc délégué à un script "TurnManagerScript.cs" (Listing 3.3) la gestion des unités.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class TurnManagerScript : MonoBehaviour
6  {
7      public float _timeTick;
8      public float _ticksPerSeconds;
9
10     // Update is called once per frame
11     void FixedUpdate ()
12     {
13         _timeTick += 0.02F * Time.timeScale;
14         _ticksPerSeconds = (1.0f / 0.02F) * Time.timeScale;
15         if (_timeTick >= 0.04f) // Est-ce que 0.4 secondes ce sont écoulées ?
16         {
17             _timeTick -= 0.04f;
18
19             foreach (GameObject unit in GameObject.FindGameObjectsWithTag("Unit"))
20             {
21                 if (unit.GetComponent<Brain>()) // Comporte un "Brain"
22                 {
23                     unit.GetComponent<Brain>().UnitTurn(); // Fait l'action de l'unité
24                 }
25             }
26         }
27     }
28 }
```

Listing 3.3 – Code du script TurnManagerScript.cs

Ce script permet donc de controler que chaque unité ne commence pas un nouveau tour si d'autre unité n'ont pas fini ou fait le leur. Cependant, cette optimisation n'a pas permis de réduire le temps de chargement d'une partie, restant toujours à prés de deux minutes pour la génération de 200 unités.

***ObjectPool*, les "bassins" d'objets.** Pour essayer de mieux localiser le problème, nous avons effectuer des tests sur le temps de chargement en fonction du nombre d'unités. Les résultats sont consignés dans le tableau ci-dessous. On peut donc voir que le nombre d'unité à générer influe directement le temps de chargement.

Nombre d'unités à générer	Temps de chargements (moyenne de 3 tests)
0 unités	0.49 secondes
1 unités	0.70 secondes
5 unités	3.13 secondes
20 unités	11.50 secondes
50 unités	27.84 secondes
100 unités	50.23 secondes
200 unités	162.20 secondes
500 unités	276.71 secondes

Pour empêcher la génération des unités pendant lors du chargement, nous décidons d'utiliser une méthode dite "du bassins d'objets" (ou *ObjectPooling*). Cette méthode consiste à créer un ensemble d'objet dans un objet appelé *Pool*, puis de les rendre inactifs pour que les scripts contenu dans ces objets ne soient pas exécutés.

```
1 public struct Pool
2 {
3     public string tag; // Label du Pool
4     public int number; // Nombre d'objets
5     public GameObject prefab; //Objet de référence
6 }
```

Listing 3.4 – Code de la structure *Pool* de ObjectPool.cs

Cette méthode permet de ne pas créer une unité directement, mais de juste prendre un objet Unit dans le bassin correspondant. Les bassins sont représentes par une structure définit au listing 3.4. elle comporte un *tag* représentant le nom du bassin, un *number* correspondant au nombre d'unités à créer dans ce bassin, et enfin un *prefab* qui est l'objet qui servira de référence pour la copie des objets. Le script gérant le script (Listing 7.1) permet la gestion de plusieurs *Pools*, et dans notre cas, nous utilisons 3 *Pools* : *Light*, *Heavy* et *Explorer*

3.3.4 Fonctionalités finales.

Mehdi

3.4 Amélioration possible

Mehdi et Maxime

3.4.1 Gestion des unités.

La gestion des unités de manière générique nous a poussé à réfléchir aux éléments minimum définissant celle-ci. Elles nécessitent donc 6 scripts de base :

- Brain
- Stats
- Percept
- Actions
- ActionNonTerminales
- Messages

Le corps

Le corps de l'unité correspond à son modèle 3D, et son GameObejct Collider associé.
Pour le jeu warbot nous avons réaliser plusieurs modèles 3D.
— Base [Photo] [Descriptif]
— Explorer [Photo] [Descriptif]
— Light [Photo] [Descriptif]
— Heavy [Photo] [Descriptif]

L'esprit

Afin de pouvoir fonctionner, les unités possede 3 parties distinctes :
— La gestion des actions possibles.
— La gestion des perceptions.
— L'identité propre de l'unité.

Perceptions. Concept de percept : qu'est ce qu'un percept ?
Un percept est condition nécessaire, dans l'architecture de subsomption, pour effectuer une action.
Création d'un percept
Gestion du percept de son appel à sa réalisation.
Prenons par exemple le percept "PerceptAlly", pour celui-ci nous ....

Actions.

Non terminales. Une action non terminale est une composante d'une action terminale, par exemple "se tourner vers un ennemi",celle-ci ne termine pas le tour de l'unité,
Création d'une action non terminale.
Gestion de l'action non terminale de son appel à sa réalisation.
Prenons par exemple l'action non terminale "ActionTurnEast", pour cellei-ci nous ....

Terminales. Une action terminale est considérée comme un ordre donné à l'unité à chaque unité de temps, à la différence de l'action non terminale, celle-ci termine le tour de l'unité,
Création d'une action terminale.
Gestion de l'action de son appel à sa réalisation.
Prenons par exemple l'action "Fire", pour cellei-ci nous ....

3.4.2 Gestion des scènes.

Une scène, comme l'unité, détient elle aussi des composants minimums, ceux-ci ont été regroupé dans un GameObject appelé "MetabotNecessary". Ils se résument en 8 scripts :
— MainCamera : la caméra principale.
— TurnManager : s'occupe de la gestion des tours.
— RessourceGenerator : un générateur de ressource.
— MinimapCamera : camera permettant l'affichage de la minimap.
— ItemManager : gère le comportement des objets.
— HUD : canvas d'affichage tête haute.
— UnitManager : gestionnaire d'unité qui comporte les 4 equipes.

3.4.3 Les différentes maps

Nous avons réaliser un ensemble de cinq maps chacune ayant son thème et ses particularités :
— Mountain [Photo] [Descriptif]
— Plain [Photo] [Descriptif]
— Desolate [Photo] [Descriptif]
— Garden [Photo] [Descriptif]
— Simple [Photo] [Descriptif]

3.5 Fonctionnalités

3.6 Amélioration possible

Malgré tous nos efforts, il persiste toujours un soucis d'optimisation .....

3.7 Phase de conception

Notre conception est basée sur notre savoir en matiere de programation orientée agent tout en gardant à l'esprit la notion de généricité qui est au coeur de notre projet. Le but de notre moteur de jeu est de permettre à des développeurs de pouvoir réaliser un mode de jeu orienté agent, de facon simple.

Chapitre 4

Partie Interface Graphique

4.1 Présentation

La partie Interface Graphique comprend principalement l'habillage visuel des éléments avec lequel l'utilisateur va interagir pour jouer au jeu. MetaBot étant un jeu pour "programmeur", le joueur interagi surtout avec le menu principal et l'éditeur de comportement afin de créer des équipes pour pouvoir les faire s'affronter.

L'interface graphique que nous avons créé se décompose en quatre parties, le menu principal, le menu de paramètre, l'éditeur de comportement et des fonctionnalités directement en jeu.

4.2 Etude de l'ancienne interface

L'ancienne interface nous est tout de suite apparu comme plutôt vide et les actions y étaient très limitées, pour l'éditeur on pouvait seulement choisir une équipe, une unité et les Contrôle/Condition/Action basiques (Il y avait aussi plusieurs bugs présents mais on ne parle ici que de l'interface).

Le menu principal contenait un bouton de nouvelle partie, un bouton vers l'éditeur, un bouton pour quitter la partie et une case à cocher pour arrêter le son.

Le premier choix a donc été de revoir totalement l'interface, ce que nous pouvions faire grâce aux nouveaux éléments apportés. En effet il fallait rajouter de nouvelles fonctionnalités basiques à cette interface qui en avait finalement très peu et ajouter un peu de couleurs à tout ça.

4.3 Nouvelle Interface

4.3.1 Menu Principal

Lancer une Partie

Ce bouton permet de lancer une partie du jeu MetaBot. Le lancement de la partie prend en compte les équipes choisies, le nombre de chaque unités en début de partie, le nombre de ressources maximum (nombre de ressource présentes en jeu au même moment), et la carte de jeu.

Bouton Editeur de Comportement

Ce bouton permet d'accéder à l'éditeur de comportement.

Bouton Paramètre

Ce bouton ouvre (révèle) le menu des Paramètres.

Choisir les équipes

Dans le carré de chaque équipe il y a un menu déroulant avec les noms de toutes les équipes présentes dans les fichiers du jeu. On peut donc en sélectionner une pour qu'elle participe à la prochaine bataille. De plus à coté du nom de l'équipe on retrouve aussi son score (ELO).

Choisir le nombre d'équipe

Au dessus de toute les équipes on peut choisir le nombre d'équipe participant à la partie. Les valeurs sont dans un menu déroulant et vont de 2 à 4.

Bouton "Reload Team"

Ce bouton permet de recharger les équipes.

Bouton des Scores

Le bouton en forme de couronne ouvre l'écran des scores. Chaque équipe a un score (ELO) qui indique son "niveau" et donc qu'elle est plus apte à gagner que les équipes avec un score plus faible.

Bouton pour quitter le jeu

Ce bouton ouvre une boite de dialogue demandant à l'utilisateur si il veut vraiment quitter le jeu. Il peut ainsi choisir de revenir sur le menu principal ou de fermer le jeu.

Choisir carte de jeu

On peut directement choisir le lieu de la partie en cliquant sur les flèches de part et d'autre de l'aperçu de la carte. Il existe pour le moment cinq cartes, Moutain, Plain, Simple, Desolate et Garden.

Choisir nombre de départ de chaque unités

En dessous de l'aperçu de la carte, il y a les noms des unités présente dans le jeu. Sous ces noms, le chiffre, indique le nombre de ce type d'unité présent au lancement de la partie. On peut incrémenter ou décrémenter ce chiffre à l'aide des boutons "+" et "-" à coté de celui-ci.

Barre de Chargement

Quand on lance une partie, une barre de chargement apparaît et indique l'avancement du chargement de la partie.

4.3.2 Menu des Paramètres

Changer le Volume de la musique

Ce slider indique le niveau sonore de la musique, on peut le changer en cliquant dessus et en déplaçant la valeur de 0 jusqu'à 100. 0 correspond à un arrêt de la musique et 100 au volume maximal. De plus le volume sonore dans le menu est le même dans l'éditeur de comportement et dans le jeu lui même.

Activer/Désactiver le tir allié

Ce bouton permet d'activer ou désactiver le fait que les unités d'une équipe peuvent infliger des dégâts aux autre unités de la même équipe en jeu.

Choisir nombre de ressource maximum dans le jeu

Dans cette case on peut entrer un chiffre entier qui indiquera le nombre maximum de ressource présente, en même temps, à l'écran en jeu.

Choisir le mode de jeu

Ce menu déroulant permet de choisir le mode de jeu de la prochaine partie. Il y a actuellement deux modes, le mode TestBot et le mode RessourceRace. Si le mode choisi est RessourceRace alors deux autres paramètres apparaissent, le temps avant que la partie ne s'arrête et la limite de ressource à atteindre pour gagner.

Choisir la langue

Les boutons en forme de drapeau indiquent les langues disponibles pour le jeu. Pour changer de langue il suffit d'appuyer sur le drapeau voulut et de faire Valider les paramètres.

Bouton Retour

Ce bouton permet de retourner à l'écran du menu principal.

Bouton Valider

Ce bouton valide les paramètres définit au dessus et revient au menu principal en ayant appliquer ces paramètres.

4.3.3 Editeur de Comportement

Bouton Nouveau Comportement

Ce bouton permet de vider le comportement de l'unité de l'équipe courante. Si on veut tout effacer sur l'unité où on est on appuie dessus au lieu de tout supprimer à la main.

Bouton Chargement Comportement

Ce bouton permet de charger un comportement pour l'unité de l'équipe courante.

Bouton Sauvegarde du Comportement

Ce bouton sauvegarde le comportement de l'unité de l'équipe courante. Si on change d'unité dans la même équipe mais sans sauvegarder, alors le comportement de l'unité précédente sera perdu.

Bouton "Undo"

Ce bouton permet de ramener la dernière pièce supprimée là où elle était. C'est le bouton d'annulation de changement.

Bouton "Redo"

Ce bouton permet d'annuler le dernier "Undo" en annulant son action et ramenant l'état du comportement.

Bouton de retour au menu principal

Ce bouton ramène l'utilisateur au menu principal. Avant de cliquer dessus il faut penser à bien sauvegarder le comportement en cours pour ne pas le perdre.

Choix de l'équipe

Ce menu déroulant permet de choisir l'équipe sur laquelle on veut travailler.

Choix de l'unité

Ce menu déroulant permet de choisir l'unité de l'équipe sur laquelle on va opérer les changements de son comportement.

Bouton Création d'équipe

Juste à droite des équipes se trouve un bouton en forme de croix, il permet de créer un nouvelle équipe. Si on appuie une boite de dialogue s'ouvre et nous demande le nom de la nouvelle équipe. Après avoir validé le nom, la boite de dialogue se ferme et la nouvelle équipe est présente dans le menu déroulant des équipes.

Bouton Suppression d'équipe

Ce bouton permet de supprimer définitivement l'équipe courante. Une boite de dialogue demandera confirmation.

Affichage des statistiques de l'unité courante

Quand on choisit une unité, sur sa droite apparaît ses valeurs dans le cadre "Propriétés", cela correspond à ses statistiques.

Affichage du modèle 3D de l'unité courante

Dans le même cadre "Propriétés" apparaît aussi le modèle 3D de l'unité courante. C'est l'apparence qu'aura l'unité en jeu.

Boutons de choix de la catégorie de la pièce

Sur la gauche, de manière verticale, se trouve cinq noms de catégories qui sont les Contrôles, les Conditions, les Actions, les Messages et les Actions non terminale. En cliquant sur une de ses catégories on affiche la liste des éléments de cette catégorie dans la zone directement à droite.

Zone de sélection de la pièce suivant la catégorie

C'est dans cette zone qu'apparaît la liste des éléments des catégories de pièces de l'éditeur. Les éléments sont présents sous forme de case avec leur nom à l'intérieur (certain possède même des menus déroulant pour choisir la valeur voulue une fois dans la zone d'édition). La couleur des éléments dépend de leur catégorie. Les éléments peuvent être sélectionnés et déplacés dans la zone d'édition du comportement grâce au glissé/déposé (Drag & Drop).

Zone éditeur de comportement de l'unité et de l'équipe courante

Dans cette zone arrivent les pièces venant de la zone de sélection. Elles ce placent à l'aide du curseur de la souris sur un grille invisible. Si un pièce est valide alors elle est colorée sinon elle est grise. Les pièces "IF" possèdent un cadenas dans le coin supérieur droit, il permet de déplacer, en plus de la pièce IF, tout les éléments valides qui lui sont rattachés (hors IF). Les pièces de contrôle sont valide en dessous d'autre pièce du même type ou en dessous de la pièce "Start". Les autres pièces doivent se rattacher à des pièces de contrôle, en haut à droite pour les pièces Condition et en bas à droite pour les autres. Les pièces hors Contrôle sont ainsi valide lorsqu'elles sont au bon endroit, sur leur ligne collé au contrôle ou collé à une autre pièce valide.

4.3.4 Élément dans le jeu

Réglage du volume du son

En bas à droite de la fenêtre de jeu il y a un icône de son et un slider. Le slider permet, comme dans le menu des paramètres, de changer le volume du son. L'icône lui sert de bouton, si on le presse le son passe à 0 et l'icône devient barré. Si on appuie de nouveau il devient normal et le son revient comme avant.

Troisième partie

L'avenir du projet

Chapitre 5

Amélioration possible

Chapitre 6

Bugs

Quatrième partie

Annexe

Chapitre 7

Scripts Remarquables

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ObjectPool : MonoBehaviour
{
    static public Dictionary<string, Queue<GameObject>> dictPools = new Dictionary<
    string, Queue<GameObject>>();
    public Pool[] pools;

    static bool created = false;

    void Awake()
    {
        if (!created)
        {
            DontDestroyOnLoad(this.gameObject);
            created = true;
        }
        else
        {
            Destroy(this.gameObject);
            return;
        }

        foreach(Pool pool in pools)
        {
            pool.prefab.SetActive(false);
            Queue<GameObject> queue = new Queue<GameObject>();
            for (int i = 0; i < pool.number; i ++)
            {
                GameObject instance = Instantiate(pool.prefab);
                instance.SetActive(false);
                instance.transform.parent = transform;
                queue.Enqueue(instance);
            }
            dictPools.Add(pool.tag, queue);
        }

    }

    static public GameObject Pick(string tag, Vector3 position, Quaternion rotation)
```

```
42      {
43          GameObject obj = dictPools[tag].Dequeue();
44          if (obj)
45          {
46              obj.transform.position = position;
47              obj.transform.rotation = rotation;
48              obj.SetActive(true);
49
50          }
51          return obj;
52      }
53
54      static public GameObject Pick(string tag, Vector3 position)
55      {
56          return Pick(tag, position, Quaternion.identity);
57      }
58
59      static public GameObject Pick(string tag)
60      {
61          return Pick(tag, Vector3.zero, Quaternion.identity);
62      }
63  }
```

Listing 7.1 – Script ObjectPool.cs