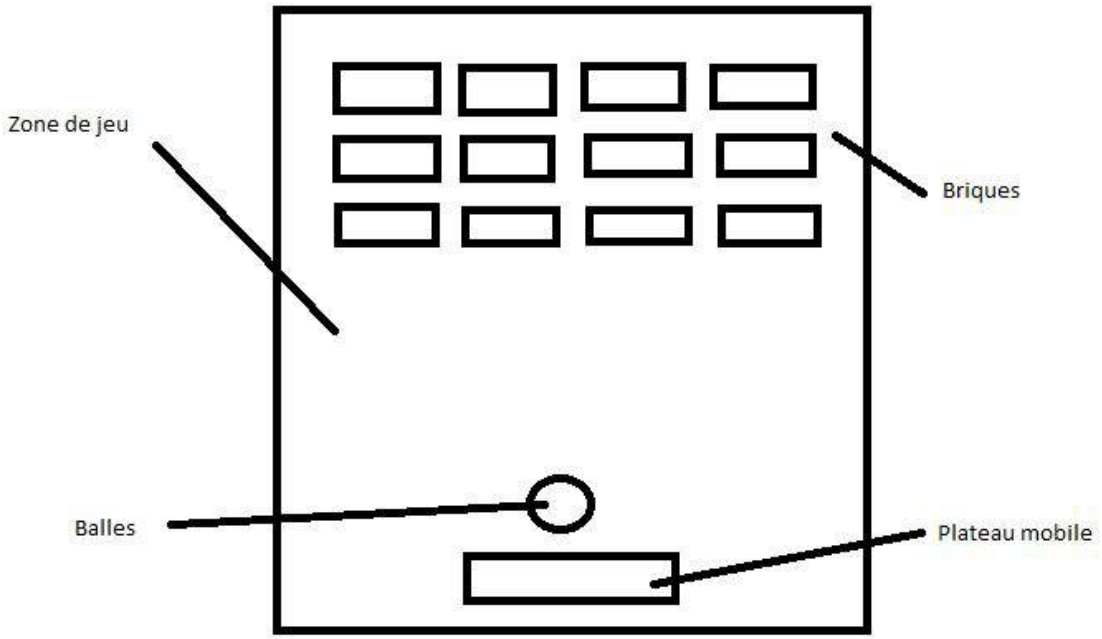
Tutoriel : Le casse-brique

I - Introduction

Ce tutoriel à pour objectif de montrer les différentes étapes de la création d'un jeu de casse brique avec deux technologie différentes. Nous avons choisi Unity et React pour édutier les différentes technologies et les adaptations d'interaction qu'elle propose.

Pour étudier efficacement les interactions sur ces différents terminaux nous avons décidé de développer une interface minimal et d'implémenter plusieurs interaction différentes. Voici une maquette de l'interface, et une liste des interactions choisi :



Lancer la balle :

- Barre espace sur clavier d'ordinateur
- Relâcher le clique de la sourie d'ordinateur
- Relâcher la pression du doigt sur l'écran du téléphone

Déplacer le plateau :

- Flèches directionnelles du clavier d'ordinateur
- Touches "W" "A" "S" "D" du clavier d'ordinateur
- Maintient du clique et déplacement de la sourie d'ordinateur
- Pression à un endroit de l'écran du téléphone
- Maintenir la pression du doigt et le déplacer sur l'écran du téléphone

II - React

II.1 - Réalisation

Nous avons choisi ici de vous montrer comment faire un casse-brique en JS puis en Natif Android et ensuite de l'adapter à React afin de mieux vous montrer à quoi sert React et comment il fonctionne. Nous allons essayer de rester le plus fidèle possible entre les deux implémentations (Javascript et Andoid), afin que le code React (JSX) soit identique pour les deux versions.

II.1.a - Javascript

Le casse brique a été réalisé en JavaScript pure, permettant de définir les objets qui le compose. On trouvera notamment les briques, la balle et le paddle, pour les objets physique et le canvas pour les objets virtuels.

Pour afficher tout cela, nous avons décidé de faire le code dans un canvas JS.

```
<canvas id="myCanvas"></canvas>
```

On pourra désormais utiliser le contexte du canvas grâce à la fonction suivante :

```
var ctx = document.getElementById("myCanvas").getContext("2d")
```

Et le canvas :

```
var canvas = document.getElementById("myCanvas")
```

Que l'on placera avant les fonctions que l'on va définir, se seront des variables dites globales.
Le contexte nous permettra plus tard de créer et de consulter les éléments présent dans le canvas.

Nous avons donc un canvas prêt de recevoir des "dessins".

II.1.a.1 - Plateau

On va d'abord réalisé le plateau afin de tester les réactions (avec le navigateur internet, souris et flèche directionelle).
On ajoute donc le plateau dans une fonction drawPaddle

```
ctx.beginPath(); // on ouvre la possibilité d'ajouter du contenu
ctx.rect(paddleX,canvas.height-paddleHeight,paddleWidth,paddleHeight); // On créer le rectangle grâce à des variables globale
ctx.fillStyle = "#0095DD"; // Pour la couleur
ctx.fill(); // on applique le plateau
ctx.closePath(); // On ferme la possibilité d'ajouter du contenu.
```

On ajoute donc les variables paddleX, paddleHeight et paddleWidth
Afin de pouvoir recevoir les événements, il faut demander à javascript d'écouter les activités du clavier et de la souris.

Pour le clavier :

```
document.addEventListener("keydown",keyDownHandler, false);
document.addEventListener("keyup", keyUpHandler, false);
```

keydown pour l'appuie d'une touche et keydown lorsqu'on lâche une touche. Il est nécessaire de faire les deux cas, car il faut déplacer le plateau uniquement lorsque l'on est appuyé sur la touche.

On bind les fonctions qui vont être appelées lors de l'appuie des touches.
l'intérieur des fonctions ressembleront à cela:

```
if(e.keyCode == 39) {rightPressed = true;}
else if(e.keyCode == 37) {leftPressed = true;}
```

On va mettre dans une variable le fait que les touches sont enfoncées (leftPressed = true) ou relachées (leftPressed = false).

Pour la souris :

```
document.addEventListener("mousemove",mouseMoveHandler,
false);
document.addEventListener("mousedown",mouseClickHandler, false);
document.addEventListener("mouseup", mouseClickRelease, false);
```

mousemove lorsque la souris se déplace, mousedown lorsque le click et préssé et mouseup lorsque le click est relaché.

On a donc lié les listeners à des fonctions, on peut donc maintenant les définir:
Dans mouseClickHandler, on va juste prévenir que le bouton a été enfoncé à l'aide d'une variable globale qu'il faut ajouter

```
click = true;
```

Le mouseClickRelease va prévenir que l'on a lâcher le bouton et donc modifier la variable click

```
click = false;
```

Pour mouseMoveHandler, on recupère la position de la souris par rapport au canvas.
puis on vérifie que l'utilisateur click bien sur la souris grâce à la variable click.

```
var relativeX = e.clientX -canvas.offsetLeft;
if(relativeX > 0 && relativeX <canvas.width && click){paddleX =
relativeX -paddleWidth/2;}
```

On a donc créée la possibilité d'afficher un plateau et de le bouger !!!.

On va créer une fonction globale qui va permettre de dessiner les différents éléments.
Dans la suite, on appellera cette fonction draw.
Il faudra d'abord libérer le canvas de son précédant contexte

```
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

puis dessiner le plateau :

```
drawPaddle();
```

Il faut donc consulter les variables précédemment définies (rightPressed/leftPressed) ainsi que les limites du canvas, afin de ne pas dépasser les bords.

```
if(rightPressed &&paddleX <canvas.width-paddleWidth){paddleX += 7;}
else if(leftPressed &&paddleX > 0) {paddleX -= 7;}
```

Enfin il faut ajouter cette ligne dans la fonction draw:

```
requestAnimationFrame(draw);
```

Cette fonction va permettre d'animer le canvas et donc de mettre à jour les données en temps réel.

Pour finir le plateau, il faut lancer la fonction draw dans un script classique javascript sur la page html.

A ce stade, nous pouvons lancer le fichier html dans un navigateur afin de voir le paddle (fonctionnel, il réagit à la souris et aux touches directionelles).

II.1.a.2 - Les briques

Dans la même idée que le plateau, nous allons créer une fonction pour afficher les briques. Tout d'abord, il faut mettre une variable contenant les briques dans les variables globales. React ne supportant pas les matrices, nous avons fait le choix de faire des tableaux imbriqués.

```
bricks: [
  [{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 }],
  [{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 }],
  [{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 },{ x: 0, y: 0, status: 1 }]
]
```

x et y corresponde à la position de chaque brique, le status correspond au fait qu'elle soit détruite (0) ou non (1).

Une fois les briques instanciées, on peut les remplir à l'aide d'une fonction qui les dessinera. La fonction contiendra à peut près cela
Il ne faut pas oublié d'ajouter les variables globales suivantes, brickWidth, brickHeight, brickColumnCount et brickRowCount.

```
for(var c=0; c<brickColumnCount; c++) {
```

```
for(var r=0; r<brickRowCount; r++) {
    if(bricks[c][r].status == 1) {
        var brickX = (r*(brickWidth+brickPadding))+brickOffsetLeft;
        var brickY = (c*(brickHeight+brickPadding))+brickOffsetTop;
        bricks[c][r].x = brickX;
       bricks[c][r].y = brickY;
       ctx.beginPath();
       ctx.rect(brickX, brickY,brickWidth,brickHeight);
       ctx.fillStyle = "#0095DD";
       ctx.fill();
       ctx.closePath();
    }
}}
```

On ajoute la fonction dans la fonction draw, cela devrait ajouter les briques dans le canvas.

II.1.a.3 - Les textes

Nous allons ajouter deux nouvelles fonctions, permettant respectivement d'afficher le score et le nombre de vie.

Pour le score : (on ajoute la variable score)

```
ctx.font = "16px Arial";
ctx.fillStyle = "#0095DD";
ctx.fillText("Score: "+score, 8, 20);
```

Pour le nombre de vie : (on ajoute la variables lives)

```
ctx.font = "16px Arial";
ctx.fillStyle = "#0095DD";
ctx.fillText("Lives: "+lives, canvas.width-65, 20);
```

II.1.a.4 - La balle

Nous allons maintenant faire la dernière partie du jeu en JS avant de l'adapter en React.
On fait donc une fonction qui permet de dessiner la balle

```
ctx.beginPath();
ctx.arc(x,y,ballRadius, 0, Math.PI*2);
ctx.fillStyle = "#0095DD";
ctx.fill();
ctx.closePath();
```

On doit donc ajouter les variables globales suivantes: x et y pour la position de la balle dans le canvas, ballRadius pour le rayon de la balle.

On ajoute cette fonction dans la fonction draw.

II.1.a.4.a - Le mouvement

Maintenant on veut lancer la balle, pour cela on doit définir une fonction qui permet de la faire bouger

```
if(start) {
  x += dx;
  y += dy;
} else {
  x = paddleX + 37;
  y = canvas.height-40;
}
```

start est une variable globale qui permet de lancer la balle. On l'initialise à false au départ.

On doit définir les fonctions "listeners" pour passer la variable à "true", on a choisi l'espace et le click de souris pour lancer la balle (on vas donc rajouter les lignes nécessaires dans les fonctions mouseClickHandler et keyUpHandler).

Maintenant il faut vérifier que la balle rebondi bien sur les murs, pour cette partie, nous avons choisi des réactions de la balle simple, elle rebondira toujours avec un angle de 90°. (On lui donne deux valeurs (dx et dy), permettant de modéliser un vecteur afin de donner un mouvement fixe à la balle). Elles seront incrémentées au fur et à mesure, comme défini dans la fonction ci-dessous.
Ce code doit être mis dans la fonction draw

```
if(x + dx > canvas.width-ballRadius || x + dx < ballRadius) {dx = -dx;}

if(y + dy < ballRadius) {dy = -dy;}
```

```
else if(y + dy > canvas.height-ballRadius) {
    if(x > paddleX && x < paddleX + paddleWidth) {dy = -dy;}
    else {
      this.decrementLife;
      if(!lives) {
        alert("GAME OVER");
        document.location.reload();
      } else {
        start = false;
        x = canvas.width/2;
        y = canvas.height - 30;
        dx = 3;
        dy = -3;
        paddleX = (canvas.width - paddleWidth)/2;
      }
    }
}
```

Plusieurs chose dans ce bout de code. Tout d'abord la première partie

```
if(x + dx > canvas.width-ballRadius || x + dx < ballRadius) {dx = -dx;}
```

Elle permet de vérifier que la balle ne sort pas par les côtés du canvas, et de la faire rebondir lorsque c'est le cas.

On notera les nouvelles variables à ajouter : dx et dy pour le déplacement de la balle.

On vérifie ensuite que la balle ne touche pas le haut du canvas grâce à cette condition:

```
if(y + dy < ballRadius) {dy = -dy;}
```

Une fois que les vérification pour les bords du canvas faites, on vérifie que la balle ne soit pas en dessous du plateau.

Lorsque que la balle est en dessous du plateau, on décrémente la vie grâce à la fonction decrementLife, qui modifie la variable globale life.

Si il nous reste des vies, on instancie de nouveau le jeu:

```
start = false;
x = canvas.width/2;
y = canvas.height - 30;
dx = 3;
dy = -3;
paddleX = (canvas.width - paddleWidth)/2;
```

II.1.a.4.b - Les collisions

Maintenant la balle est lancée et rebondit contre les murs, il faut la faire rebondir et casser les briques.

Il va falloir détecter les collisions grâce à cette fonction

```
for(var c=0; c<this.state.brickColumnCount; c++) {
  for(var r=0; r<brickRowCount; r++) {
    var b =bricks[c][r];
    if(b.status == 1) {
      if(x+ballRadius > b.x
          && (x-ballRadius < b.x+brickWidth ||x < b.x+brickWidth)
          && y+ballRadius > b.y
          && (y-ballRadius < b.y+brickHeight ||y < b.y+brickHeight)) {
        dy = -dy;
         b.status = 0;
         this.incrementScore;
         if(score ==brickRowCount*brickColumnCount) {
           alert("YOU WIN!");
           document.location.reload();
         } } } } }
```

Ici on fait des vérifications en comparant la position de toutes les briques avec celle de la balle. On prend en compte la largeur de la balle.

On ajoute la fonction dans draw et l'application est terminée.

II.1.b - Adaptation React

Maintenant que nous avons une application fonctionnelle en JavaScript, nous pouvons nous attaquer à l'adaptation react.

React utilise un système de component et de classe. Nous allons donc définir une classe de la manière suivante

```
var Breakout = React.createClass({
render: function(){
    return (
      <div className="counter">
        <button type="button" onClick={this.draw}>Play</button>
      </div>
    );
 }
```

```
});
React.renderComponent(<Breakout/>,document.getElementById('mount-point'));
```

Le bouton Play, permet de lancer le jeu pour la première fois.

RenderComponent permet d'afficher le component dans une div.

Et le html ressemblera a cela

```
<!DOCTYPE html>
<html>
 <head>
   <script src="build/react.js"></script>
   <script src="build/JSXTransformer.js"></script>
   <link rel="stylesheet" type="text/css" href="css/style.css" media="all"/>
 </head>
 <body>
   <div class="contains">
     <canvas id="myCanvas" width="480" height="320"></canvas>
     <div id="mount-point"></div>
     <script type="text/jsx" src="basic.js"></script>
   </div>
 </body>
</html>
```

Les deux lignes import du javascript au début sont là pour les librairies react.

La div "#mount-point" est la pour contenir les éléments react.

Le canvas contiendra le jeu

II.1.b.1 - Les variables

Les variables globales doivent être instanciés dans le "state" du component de la manière suivante

```
getInitialState: function(){
  return {
      canvas: document.getElementById("myCanvas"),
      ctx: document.getElementById("myCanvas").getContext("2d"),
      ballRadius: 10
  }
},
```

Ces variables devront être appelés à l'aide de ce "state", on devra donc les appeler comme cela

```
this.state.ballRadius
```

II.1.b.2 - Les évenements

Les évènements doivent être liés dans un contexte particuliers, il faudra que React les monte et les démontes.

```
componentWillMount: function(){
  document.addEventListener("keydown", this.keyDownHandler,
false);},
componentWillUnmount: function(){
  document.removeEventListener("keydown", this.keyDownHandler,
false);},
```

On ajoute les listeners dans componentWillMount et on les retire dans componentWillUnmount

II.1.b.3 - Les fonctions

Toutes les fonctions que nous avons précédemment définies doivent être instanciés comme suit

```
drawLives: function() {
    this.state.ctx.font = "16px Arial";
    this.state.ctx.fillStyle = "#0095DD";
    this.state.ctx.fillText("Lives: "+this.state.lives,
this.state.canvas.width-65, 20);
},
```

II.1.c - Piste d'amélioration

Suite à des complications dans les choix technologiques, nous n'avons pas pu terminer la versions Android, il serait donc intéressant de montrer le fonctionnement de React Native, qui permet d'adapter les vues sur des smartphones.

II.1.d - Outils de développement

Pour le développement de cette application, nous avons choisit d'utiliser l'éditeur WebStorm, car il nous propose une coloration du JSX.

II.2 - Déploiement

II.2.a - ReactJS

Afin de pouvoir lancer le jeu, il vous faudra installer nodeJS, npm, bower, react, babelify et reactify.
sous linux, il suffit de rentrer la commande suivante pour installer nodejs et npm:

```
sudo apt-get install nodejs npm
```

Ensuite on installe bower:

```
npm install -g bower
```

Et enfin react:

```
npm install --save react react-dom
```

Il arrive de temps à autre que babelify et reactify ne se soit pas installés sur la machine tout seul. Il faut donc les installer de la manière suivante:

```
npm install --save reactify
npm install -g babelify
```

Une fois que tout est installé, il faut aller dans le dossier ou ce trouve le .js que vous avez créé et lancer les commandes suivantes:

```
bower init
bower install
browserify --debug --transform reactify basic.js > bundle.js
```

Pour la première fois il sera nécessaire d'entrer toutes les commandes. Par la suite, il sera nécessaire d'entrer la commande suivante uniquement lorsque l'on ajoute un composant ou le supprime.

```
browserify --debug --transform reactify basic.js > bundle.js
```

Ensuite, il suffit simplement de lancer le fichier html principal.

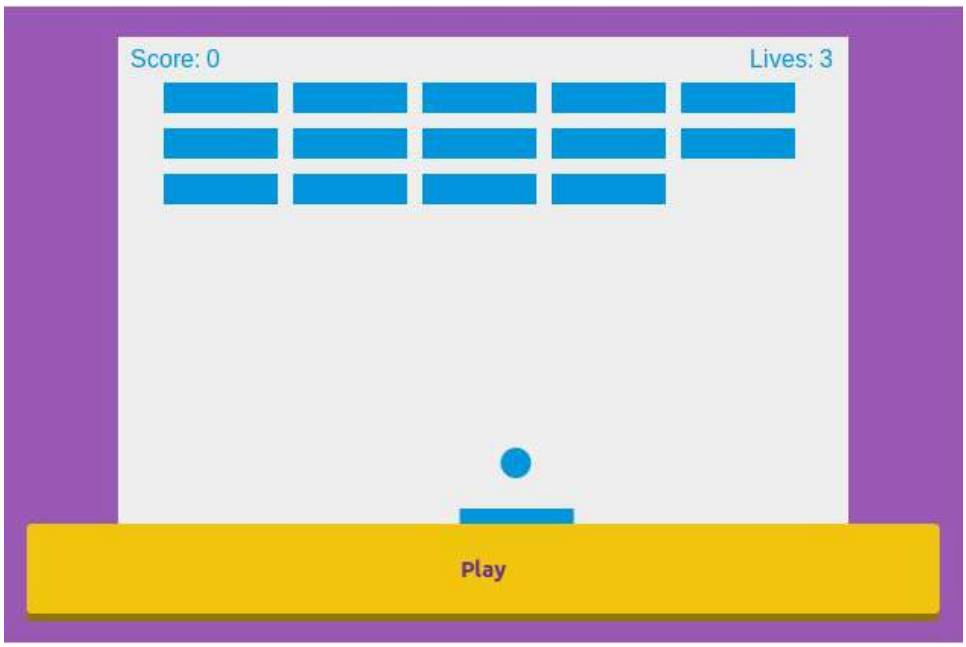
II.2.b - Capacité d'adaptation

Bien que nous n'ayons pas eu le temps de finir, nous pouvons dire que ReactJS s'adapte bien au commande ordinateur et React Native s'adapte bien aux commandes d'Android car les fonctions principales sont codées en JavaScript ou en Natif Android.

React n’est pas un framework, il gère uniquement l’affiche et la gestion de l’affichage via des composants.
Le code JSX (React) ne va cependant pas réellement changer lorsque l’on change de technologies. Il reste logique dans sa conception.
Nous n’avons donc juste à changer le code à l'intérieur.

II.3 - Rendu

On peut voir ici le Breakout. Le canvas est en blanc, le bouton play en jaune et la div react est en violette.



III - Unity3D

III.1 - Réalisation

Cette version du casse-brique, développée dans un éditeur orienté jeux vidéo, a été réalisé en deux phases:

- la conception graphique
- le développement de scripts C# définissant le comportement des éléments graphiques.

III.1.a - Interface graphique

A la création d'un nouveau projet, l'éditeur propose un environnement 2 dimensions ou 3 dimensions, il génère alors une caméra qui représente ce que voit l'utilisateur final et un élément lumière qui gère la luminosité générale de la scène.

Une fois le projet créé nous avons inséré une sphère et des cubes que nous avons dimensionnés et positionnés pour former des mur, un toit, un sol, un plateau et des briques. Ces dernières sont créées individuellement, il est donc recommandé, pour un soucis d'organisation, de créer un objet vide "bricks" et de toutes les mettre dedans.
On pourra ainsi les modifier en même temps.

Un objet physique peut également contenir d'autres objets physiques, ces derniers seront dépendant du premier. Nous nous somme servi de ce principe pour lier la balle au plateau, ainsi la sphère se déplacera en même temps que le plateau si le joueur bouge avant de tirer. Nous avons ensuite personnalisé chaque élément pour donner des comportements physique, des textures, des couleurs différentes.

Les murs, le plafond, les bricks et le plateau doivent être rebondissant pour permettre à la boule de ne pas perdre de vitesse dans l'environnement de jeu, alors que le sol, lui, doit la laisser passer, mais détecter quand même la collision pour savoir que la balle est sortie du jeu. Tous ces réglages sont facilement modifiables dans l'éditeur.

La dernière chose à faire avant de passer à la programmation du jeux est de créer un objet vide que l'on nommera "Game Manager" et des objets texte, par exemple "You won !". Nous verrons plus tard comment les utiliser.

III.1.b - Comportement

Afin de pouvoir diriger le plateau, lancer la balle, compter le nombres de briques restantes, réinitialiser la partie si elle est gagnée ou perdue, etc.
Il faut développer des scripts en C# qui seront liés aux objets que nous avons créé dans la partie précédentes.

Pour cela il faut utiliser la ligne suivante au début du fichier :

```
using UnityEngine;
```

III.1.b.1 - Le gestionnaire de jeu

Nous avons vu que les scripts doivent être attachés à des objets de la scène.
Or la mécanique général du jeu (compter le nombre de brique, réinitialiser le plateau si la balle sort, réinitialiser la partie si elle est perdue ou gagnée) n'est pas propre à un objet en particulier, c'est pour cela que nous avions créé un "Game Manager" vide, pour pouvoir lui attacher ce script.

Dans ce fichier, nous avons défini des valeurs, tel que le nombre de brique en jeu, le temps avant de réinitialiser une partie et une instance du Game Manager qui est un singleton, créé dans la fonction start.
Il faut également définir les autres objets du jeu: le texte, les briques, le plateau…
Notez que nous avions lié la balle au plateau, il nous faudra donc créer un clone de ce dernier afin de pouvoir le déplacer indépendamment de la balle.

```
public int bricks = 15;
public float resetDelay = 1f;
public GameObject youWon;
public GameObject bricksPrefab;
public GameObject paddle;
public GameObject deathParticules;
public static GM instance = null;

private GameObject clonePaddle;
```

Les fonctions de base sont start (appelé au lancement du script) et update (appelé à chaque frame du jeu).
Dans notre cas, cette dernière vérifie si l'utilisateur appui sur la touche echap pendant le jeu, pour pouvoir le quitter dans la version PC.

```
void Update()
{
    if (Input.GetKey(KeyCode.Escape))
    {
        Application.Quit();
    }
}
```

La fonction start, comme nous l'avons précisé, sert à créer notre objet selon le pattern singleton. Elle appelle ensuite la fonction setup qui instancie les briques et une copie du plateau en fonction du véritable plateau dans l'espace de jeu.

```
public void Setup()
{
    clonePaddle = Instantiate(paddle, new Vector3(0, -9f, 0), Quaternion.identity) as GameObject;
    Instantiate(bricksPrefab, transform.position, Quaternion.identity);
}
```

Ensuite le scripts suit le déroulement de la partie; à chaque collision avec une brique on décrémente le compteur et on vérifie si la partie est terminé.
Tant qu'il reste des briques il ne se passe rien, des qu'il n'y en à plus on affiche le texte "You won !", on modifie le timeScale pour donner une impression de ralenti sur l'action et on appelle la méthode reset avec un délai qui permet au joueur de voir le texte de félicitation. Cette méthode arrête le ralenti et recharge le niveau, dans le cas d'un jeu à plusieurs niveau il faut préciser le niveau à charger, sinon, par défaut, c'est le niveau courant qui sera rechargé.

Enfin la méthode loseLife sera appelée à chaque fois que la balle sort du jeu, elle détruit l'instance du clonePaddle et appelle la fonction SetupPaddle qui le ré-instancie.
Dans notre cas nous faisons tourner le jeu tant que la partie n'est pas gagnée pour ne pas être dérangée pendant les tests, mais pour implémenter un système de vies il suffit de définir un compteur comme celui des briques, de le décrémenter à chaque fois que la méthode loseLife est appelé et de le tester dans le checkWon.
S'il est à zéro mais qu'il reste des briques, on fait apparaître un texte "You loose" de la même manière que lors d'une victoire.

```
void checkWon()
{
    if(bricks < 1)
    {
        youWon.SetActive(true);
        Time.timeScale = .25f;
        Invoke("Reset", resetDelay);
    }
}

void Reset()
{
    Time.timeScale = 1f;
    Application.LoadLevel(Application.loadedLevel);
}

public void loseLife()
{
    Instantiate(deathParticules, clonePaddle.transform.position, Quaternion.identity);
    Destroy(clonePaddle);
    Invoke("SetupPaddle", resetDelay);
}

void SetupPaddle()
{
    clonePaddle = Instantiate(paddle, new Vector3(0, -9f, 0), Quaternion.identity) as GameObject;
}

public void DestroyBrick()
{
    bricks--;
    checkWon();
}
```

III.1.b.2 - Les briques

Dans le script des briques nous n'avons pas besoin de fonction start et update, uniquement une fonction qui détecte les collisions avec l'objet, qui appelle la fonction DestroyBrick du Game Manager dont nous venons de parler et qui fait disparaître la brique touchée. Notez que ce script est attaché à brique, donc à chacune des briques.

```
void OnCollisionEnter(Collision other)
{
    Instantiate(brickParticule, transform.position, Quaternion.identity);
    GM.instance.DestroyBrick();
    Destroy(gameObject);
}
```

III.1.b.3 - La balle

Pour rappel: au lancement d'une partie la balle est liée au plateau mobile, il va donc falloir la détacher de son élément parent, la rendre solide et lui appliquer une force.
C'est la fonction Update qui attend un événement de type clique ou barre espace si la balle n'est pas en jeu, pour appliquer ces modification.
Voici la classe correspondante :

```
public float ballInitialVelocity = 600f;

private Rigidbody rb;
private bool ballInPlay;

void Awake()
{
    rb = GetComponent<Rigidbody>();
}

void Update()
{
    if ((Input.GetKey(KeyCode.Space) || Input.GetMouseButtonUp(0)) && ballInPlay == false)
    {
        transform.parent = null;
        ballInPlay = true;
        rb.isKinematic = false;
        rb.AddForce(new Vector3(ballInitialVelocity, ballInitialVelocity, 0));
    }
}
```

III.1.b.4 - Le plateau mobile

Le plateau mobile est l'élément le plus important de notre application (juste après le gestionnaire de jeu) car c'est lui que l'on veux déplacer avec différentes interactions.
La position peux être modifiée par les flèches directionnelles du clavier, ou par la position de la souris ou du doigt, grâce au principe de collision de rayon.

Pour cela nous avons ajouté un élément transparent devant le jeu pour récupérer cette position. Cette classe étant plus complexe, nous l'avons commenté pour la rendre plus compréhensible :

```
// Update is called once per frame
void Update() {

    //check if the screen is touched / clicked
    if ((Input.touchCount > 0 && Input.GetTouch(0).phase == TouchPhase.Moved) || (Input.GetMouseButton(0)))
    {
        //declare a variable of RaycastHit struct
        RaycastHit hit;
        //Create a Ray on the tapped / clicked position
        Ray ray = new Ray();
        //for unity editor
        #if UNITY_EDITOR
        ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        //for touch device
        #elif (UNITY_ANDROID || UNITY_IPHONE || UNITY_WP8)
         ray = Camera.main.ScreenPointToRay(Input.GetTouch(0).position);
        #endif

        //Check if the ray hits any collider
        if (Physics.Raycast(ray, out hit))
        {
            //set a flag to indicate to move the gameobject
            flag = true;
            //save the click / tap position
            playerPos = hit.point;
            //as we do not want to change the y axis value based on touch position, reset it to original y axis value
            playerPos.y = xAxis;
            playerPos.z = 0;
            Debug.Log(playerPos);
        }


}
else
{
    float xPos = transform.position.x + (Input.GetAxis("Horizontal") * paddleSpeed);
    playerPos = new Vector3(Mathf.Clamp(xPos, -8f, 8f), -9f, 0);
    transform.position = playerPos;
}

//check if the flag for movement is true and the current gameobject position is not same as the clicked / tapped position
if (flag && !Mathf.Approximately(gameObject.transform.position.magnitude, playerPos.magnitude))
{ //&& !(V3Equal(transform.position, endPoint))){
  //move the gameobject to the desired position
    playerPos.x = Mathf.Clamp(playerPos.x, -8f, 8f);
    //gameObject.transform.position.x = Mathf.Clamp(gameObject.transform.position.x, -8f, 8f);
    gameObject.transform.position = Vector3.Lerp(gameObject.transform.position, playerPos, 1 / (paddleSpeed * (Vector3.Distance(
}
//set the movement indicator flag to false if the endPoint and current gameobject position are equal
else if (flag && Mathf.Approximately(gameObject.transform.position.magnitude, playerPos.magnitude))
{
    flag = false;
    Debug.Log("I am here");
}
```

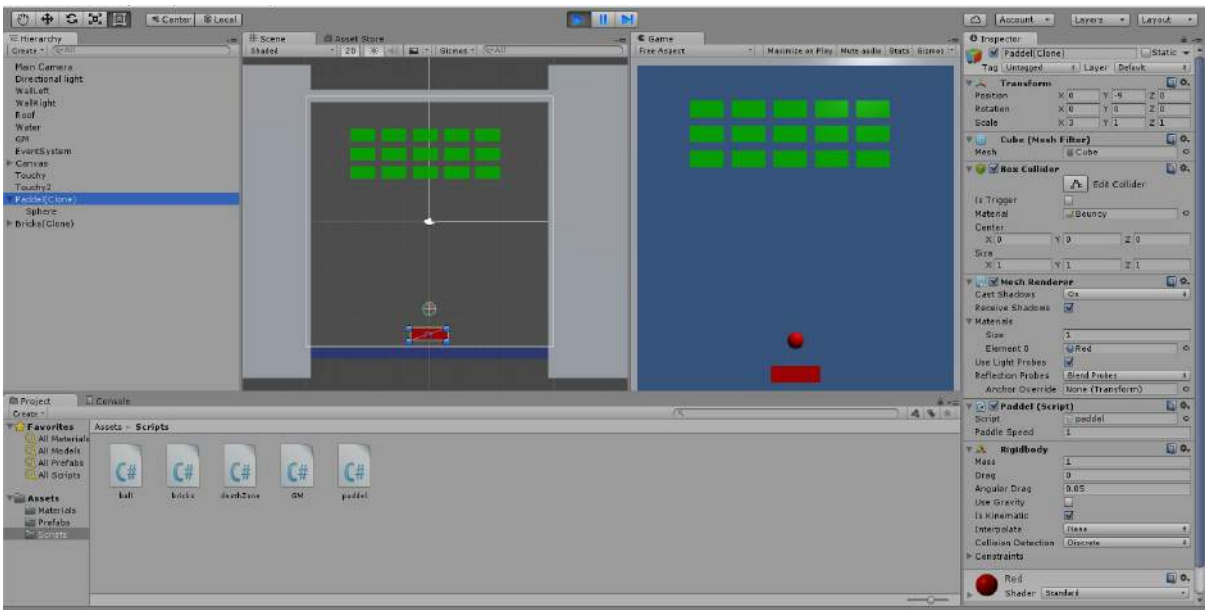
III.1.b.5 - La sortie de jeu

Un script DeathZone est attaché à l'élément sol et de la même manière que les briques il appelle la fonction looseLife du Game Manager à chaque fois qu'il est traversé (par la balle).

```
void OnTriggerEnter(Collider col)
{
    GM.instance.loseLife();
}
```

III.1.c - Environnement et test

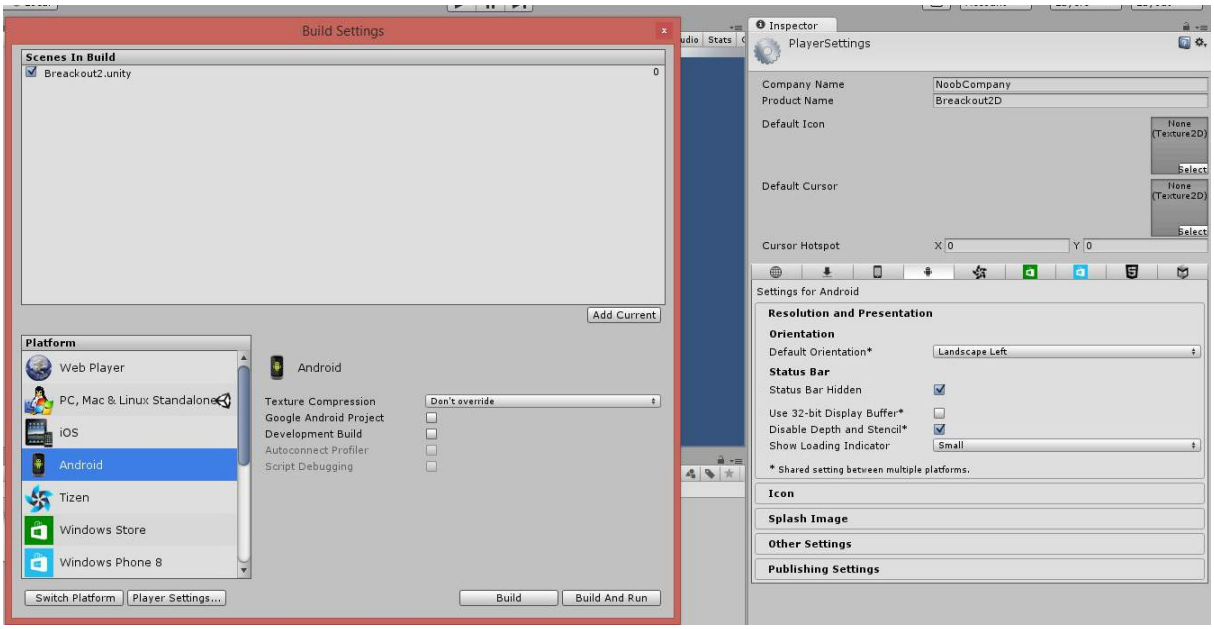
Unity permet de visualiser le jeu avant de l'exporter sur les différentes plates-formes. Cependant il existe quelques différences entre cette version et la version finale au niveau des Listeners. Notez que lorsque l'on est dans ce mode, toutes modifications seront supprimées lors du retour au mode normal.



Pour l'édition des scripts Unity ouvre Visual Studio, mais vous pouvez choisir un éditeur de votre choix adapté au C#.

III.2 - Déploiement

L'éditeur Unity propose des options de construction de projet très poussé en fonction du terminal pour lequel on veux effectuer une application. Nous avons par exemple paramétré le jeux pour qu'il s'exécute en paysage sur les appareilles mobile. Il suffit ensuite de lancer le .exe ou le .apk généré, sur le terminal correspondant.



III.3 - Adaptation

Les adaptations sont faite en fonction des plates-formes sur lesquelles on va développer, c'est à dire qu'il faut faire des Listeners sur chaque interactions possibles avec la plate-forme finale. Si l'on veut exporter le jeu sur plusieurs plate-forme, on fait des Listeners de plusieurs plate-forme.

III.4 - Rendu

Voici donc le rendu final de l'application généré par Unity qui fonctionne aussi bien sur mobile que sur ordinateur.