Création de jeux avec BabylonJS

Support des cours

Document mis à jour le 20/01/2024 18:50

Version de brouillon

Historique des versions

Date	Version	Commentaires
26/12/2023	V1	Première version
20/01/2024	V1.1	Ajout infos et liens jeux en cours

Liste de diffusion

Arguimbau Olivier	1ex

Auteur : **Olivier Arguimbau**

Table des matières

Support 1 – Principes de base

Gestion de projet, suivre UN PLAN, notions techniques et conseils.

Faire le point

Il faut évaluer votre niveau de maitrise des divers aspects du développement (de jeu mais pas que) :

- Maitrise de l'OS et des outils de développement
- Gestion de projet et travail collaboratif
- Maitrise du Javascript/Type Script (classique, objet ?)
- Maitrise du génie logiciel, de la 3D

Spécifications générales ou « Quel est mon jeu ? »

Pour réaliser un jeu ou tout autre développement informatique il faut déjà savoir ce que l'on veut faire, il faut donc déterminer les spécifications générales du jeu.

En développement les spécifications prennent la forme d'un document qui réponds souvent à la 'question' : « Le quoi ! » ou dans le cadre d'un jeu : « **Quel est mon jeu ?** »

Il faut ensuite lister les caractéristiques du jeu, ses composantes et les lister.

Exemple à développer en cours : « Mario » « Angry birds » « Pac man »

Ex : « Un jeu 3D, pas de sauvegarde, solo, avec un timer, on doit ramasser des pièces d'or en avançant en permanence, on meurt après 30s, etc. »

Gestion de projet ou « Que fait mon jeu ? »

Après les spécifications et afin de pouvoir finir le jeu (ou tout autre projet informatique) il vous faut un plan. Un plan permet de ne pas se perdre en cours de développement. Un plan n'est pas figé mais doit l'être le plus possible.

Une fois le plan déterminé il faut attribuer les délais et la répartition des taches ainsi que des réunions régulières pour suivre l'avancement et rectifier le tir au besoin.

Exemple :

1. Préparer le repo
2. Créer le squelette du jeu
3. Créer la classe principale (main loop) et les states
4. …

FSM / Boucle principale / Entities / Code the classics

Principe 1 le "Gameloop"

Le principe est assez simple, une fois que tout est initialisé le jeu va "boucler" et exécuter une liste de taches, en simplifiant cela peut donner :

- Vérifier les input (clavier, souris, etc.)
- Mettre à jour les vitesses et positions des différents composants du jeu (joueur, balle, ennemis, etc.)
- Vérifier les collisions et y réagir (changer la vitesse, la position, les points de vie, etc.)
- Afficher les composants du jeu (joueur, ennemis, score, etc.)
- Recommencer tant que le jeu n'est pas terminé

Exemple d'une boucle simple qui fait "rebondir" une balle, -le joueur n'ayant pas encore été codé- si elle descend en bas de l'écran le jeu est perdu :

```
//On initialise la taille de l'écran de jeu
let screenWidth = 640;
let screenHeight = 480;
//On place la balle au centre et en bas
let ballX = screenWidth/2;
let ballY = screenHeight - 20;
//On détermine la vitesse de départ de la balle, vers le haut et à droite
let vitesseX = 1;
let vitesseY = -2;
//On définie la variable "gameOver" à faux
let bGameOver = false;
function gameLoop() {
    //Tant que on n'a pas perdu on boucle
    while (!bGameOver) {

        //Verification des inputs
        //ICI...

        //Update joueur
        //ICI...

        //Update balle, on ajoute à sa position la vitesse
        ballX = ballX + vitesseX;
        if (ballX > screenWidth) { //La balle est en dehors à droite, on la
positionne à droite et on inverse sa vitesse en X
            ballX = screenWidth;
            vitesseX = -vitesseX;
        }
        else if (ballX < 0) {  //La balle est en dehors à gauche, on la
positionne à gauche et on inverse sa vitesse en X
```

```
            ballX = 0;
            vitesseX = -vitesseX;
        }
        //On fait de même avec Y
        ballY = ballY + vitesseY;
        if (ballY > screenHeight) {
            ballY = screenHeight;
            vitesseY = -vitesseY;
        }
        else if (ballY < 0) {
            bGameOver = true;  // Ici si la balle descend tout en bas on perd
la partie
        }

        //FIN DES UPDATES

        //Render player ICI...
        //DrawPlayer(playerX, playerY);

        //Render balle ICI...
        //DrawBall(ballX, ballY);
    } //Fin du while
    //Fin du jeu
    //Deinit, etc.

}

gameLoop();
```

Ce code n'est pas fonctionnel mais le principe est simple.
La plupart des jeux utilisent ce mécanisme et suivant le langage il va varier, on peut par exemple avec BabylonJS la trouver sous cette forme :

```
engine.runRenderLoop(function() {
    //Verification des inputs
    //ICI....

    //Update joueur
    //ICI...
    scene.render();
}
```

Ici c'est BabylonJS qui s'occupe de "faire tourner" notre boucle principale, en fait il va exécuter cette partie du code idéalement 60 fois par seconde, il ne s'agit plus vraiment d'une boucle mais le principe reste le même puisque ce code sera exécuté régulièrement.

Principe 2 : Final State machine (ou automate fini en Français)

Il s'agit d'un principe de programmation général et qui n'est pas lié aux jeux, cependant il est très souvent utilisé en programmation de jeu.

Votre programme va passer d'un état à un autre parmi une liste d'états prédéterminés et connus. Exemple en C :

```
enum States {
    INIT,
    MENU,
    START_GAME,
    PLAYING,
    PAUSE,
    WIN,
    GAME_OVER,
    EXITING
  };
```

Imaginons ce jeu simple en pseudo code :

```
void main(void) {
    enum States currentState = INIT;
    //Initialisation du jeu
    initGame();
    currentState = MENU;
    displayMenu();
    currentState = START_GAME;
    gameLoop();
    currentState = GAME_OVER;
  }
```

etc.

Si on retravaille notre jeu de balle avec ce mécanisme et en remplaçant notre variable "bGameOver" par un enum cela donne :

```
let States = Object.freeze({  //Enum like mais en JavaScript
    INIT : 0,
    PLAYING : 1,  //On joue
    PAUSE : 2,    //On a demander la pause
    GAME_OVER: 3  //On a perdu
})
let currentState = States.INIT; //Notre état de départ
Et voici le code complet en mode "State Machine" :
//On initialise la taille de l'écran de jeu
let screenWidth = 640;
let screenHeight = 480;
```

```
//On place la balle au centre et en bas
let ballX = screenWidth/2;
let ballY = screenHeight - 20;
//On détermine la vitesse de départ de la balle, vers le haut et à droite
let vitesseX = 1;
let vitesseY = -2;
//On définie la variable "gameOver" à faux
let States = Object.freeze({
    INIT : 0,
    PLAYING : 1,
    PAUSE : 2,
    GAME_OVER: 3
})
let currentState = States.INIT;
function gameLoop() {
    currentState = States.PLAYING;
    //Tant que on n'a pas perdu on boucle
    while (currentState != States.GAME_OVER) {
        if (currentState == States.PLAYING) {
            //Verification des inputs la fonction est fictive attention
            //ICI...
            if (Inputs["Space"] == true) {
                currentState = States.PAUSE;
            }

            //Update joueur
            //ICI...
            //Update balle, on ajoute à sa position la vitesse
            ballX = ballX + vitesseX;
            if (ballX > screenWidth) { //La balle est en dehors à droite, on
la positionne à droite et on inverse sa vitesse en X
                ballX = screenWidth;
                vitesseX = -vitesseX;
            }
            else if (ballX < 0) {  //La balle est en dehors à gauche, on la
positionne à gauche et on inverse sa vitesse en X
                ballX = 0;
                vitesseX = -vitesseX;
            }
            //On fait de même avec Y
            ballY = ballY + vitesseY;
            if (ballY > screenHeight) {
                ballY = screenHeight;
                vitesseY = -vitesseY;
            }
            else if (ballY < 0) {
                currentState = States.GAME_OVER;
            }
            //FIN DES UPDATES
```

```
        }
        else if (currentState == States.PAUSE) {
            if (Inputs["Space"] == true) {
                currentState = States.PLAYING;
            }
        }

        //Render player ICI...
        //DrawPlayer(playerX, playerY);
        //Render balle ICI...
        //DrawBall(ballX, ballY);
    } //Fin du while
    //Fin du jeu
    //Deinit, etc.

}
gameLoop();
```

Bien sur ce code n'est pas idéal mais il faut comprendre le mécanisme général.

En dernier lieu cette vidéo explique parfaitement le principe avec un exemple en JavaScript et fonctionnel : https://www.youtube.com/watch?v=QXYT8sEJRZk

Principe 3 : Le modèle des « entités » ou Entities

Ce modèle est très fortement lié à la programmation orientée objet mais peut être implémenté sans (avec des struct en C par exemple). Les classes sont tout de même préférables car elles permettent d'utiliser le principe de l'héritage et de segmenter le code en briques fonctionnelles distinctes.

Principe : Chaque composant « actif » de votre jeu est basé sur une classe parente dite « Entity » qui possède les caractéristiques basique de tout les composants de votre jeu, par exemple :

- Position X, Y, Z
- Vitesse X, Y, Z
- Mesh (modèle) 3D associé

Et des méthodes génériques également, par exemple :

- Init (chargement des fichiers 3D, textures, etc.)
- Update (déplacement de l'entité suivant ses vitesses)
- Render (affichage éventuel si on doit le réaliser à la main)
- CheckCollisions
- Destroy (ménage ou fonctions spécifiques)

Exemple d'implémentation en JavaScript et BabylonJS:

```
class Entity {

  x = 0;
  y = 0;
  z = 0;
  prevX = 0;
  prevY = 0;
  prevZ = 0;

  vx = 0;
  vy = 0;
  vz = 0;

  transform;
  gameObject;

  constructor(x, y, z) {
    this.x = x || 0;
    this.y = y || 0;
    this.z = z || 0;
    this.transform = new TransformNode();
    this.transform.position = new Vector3(this.x, this.y, this.z);
  }

  setMesh(gameObject) {
```

```
    this.gameObject = gameObject;
    this.gameObject.parent = this.transform;
  }

  setPosition(x, y, z) {
    this.x = x || 0;
    this.y = y || 0;
    this.z = z || 0;
    this.updatePosition();
  }

  updatePosition() {
    this.transform.position.set(this.x, this.y, this.z);
  }

  applyVelocities(factor) {
    this.prevX = this.x;
    this.prevY = this.y;
    this.prevZ = this.z;

    factor = factor || 1;

    this.x = this.x + (this.vx * factor);
    this.y = this.y + (this.vy * factor);
    this.z = this.z + (this.vz * factor);

  }

  //Called on first frame
  start() {

  }

  update(deltaTime) {

  }

  render() {

  }

}
```

Enfin nous créons une classe pour notre « Ennemy » par exemple, classe dérivée de Entity :

Elle possède des caractéristiques propres en plus de celles héritées :

- Nombre de points de vie
- Type d'ennemi
- Dégâts potentiels au joueur

Implémentation en JS :

```
class Ennemy extends Entity {

  #type;
  #life;
  #degats;

  constructor(typeEnnemi, x, y, z) {
    super(x, y, z);

    this.#type = typeEnnemi;
    this.#life = 100 + typeEnnemi*5;
    this.#degats = typeEnnemi*10;
  }

  attackOnPlayer(player) {
    player.life -= this.#degats;
  }

  attackFromPlayer(degats) {
    this.#life -= degats;
  }

}
```

Ce sont des exemples simples mais assez efficaces.

Ainsi dans la boucle principale (ou plutôt dans un « MANAGER ») nous pourrons faire un UpdateEnnemy de ce type :

```
class EnnemyManager() {

  #ennemies = [];

  constructor() {

  }

  UpdateEnnemies() {
```

```
    //On parcours le tableaux contenant nos ennemis
    for (let idx = 0; idx < this.#ennemies.length; idx++) {
      let ennemy = this.#ennemies[idx];
      ennemy.update(deltaTime);
    }
  }

}
```

Ceci sera très semblable à un update d'un autre type d'entité, par exemple des lances, des missiles ou javelots :

```
class MissilesManager() {

  #missiles = [];

  constructor() {

  }

  UpdateMissiles() {

    //On parcours le tableaux contenant nos missiles
    for (let idx = 0; idx < this.#missiles.length; idx++) {
      let missile = this.#missiles[idx];
      missile.update(deltaTime);
    }
  }

}
```

On voit qu'il n'y a quasiment pas de différence, la fonction « update » étant au départ une méthode de la classe Entity le code devient réutilisable, plus clair et plus compréhensible.

Principe 4 : Code the Classics

Il s'agit plus d'un conseil d'apprentissage que d'un principe, en effet il est bien plus facile d'apprendre à coder un jeu en imitant un jeu « simple » et classique que d'essayer de recopier un jeu complexe récent.

Quelques exemples :

- PacMan
- Pong
- Casse-briques
- Space Invaders ou autre shoot em up

A défaut de le coder totalement il faut en étudier au moins un, je vous invite à voir la déclinaison de Arkanoid que j'ai mis à disposition sur mon dépôt github : https://github.com/pigmin/Breakout

Conseils

Editeur

Vous pouvez utiliser l'éditeur de votre choix à condition de le maitriser, je conseille VScode mais libre à vous de changer.

VScode :

- Installation : https://code.visualstudio.com/
- Doc BabylonJS : https://doc.babylonjs.com/setup/support/vsCode

Gestion du travail en collaboration / versioning

Il est très fortement recommandé d'utiliser GIT, la plateforme finale (github, gitlab) n'a pas d'importance mais GIT est indispensable.

- Tutoriel : https://www.ionos.fr/digitalguide/sites-internet/developpement-web/tutoriel-git/
- Tuto avec VScode : https://www.youtube.com/watch?v=hwqgzZBsc_I&t=2s

Déploiement / Distribution

Il est fortement recommandé d'utiliser un système de déploiement de type webpack, vite ou autre. La gestion du projet et sa mise en production sera plus facile et plus rapide.

Il est également important de choisir son langage : ES6 ? Type Script ?

- Setup général : https://doc.babylonjs.com/setup
- Framework : https://doc.babylonjs.com/setup/frameworkPackages

Il faut penser à l'hébergement final du jeu (OVH, O2Switch, etc.) le plus simple à mon humble avis est d'utiliser l'hébergement de site de GitHub (heroku propose le même service).

Annexes :

Web moderne, création d'un projet BabylonJS avec NPM :

https://doc.babylonjs.com/setup/frameworkPackages/es6Support

Exemple d'un passage du Playground à la distribution sur mobile

https://doc.babylonjs.com/guidedLearning/devStories/fruitFalling