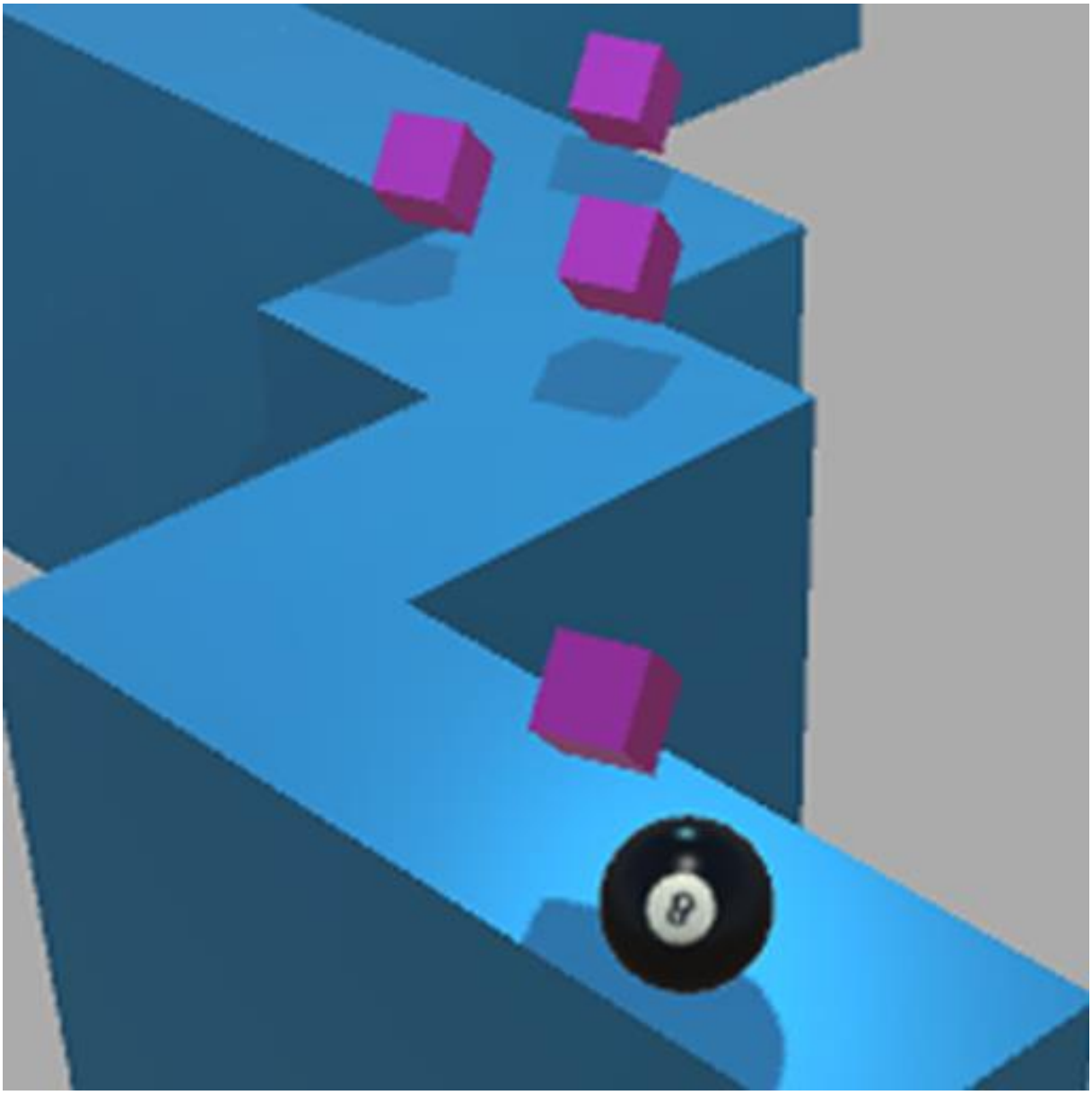
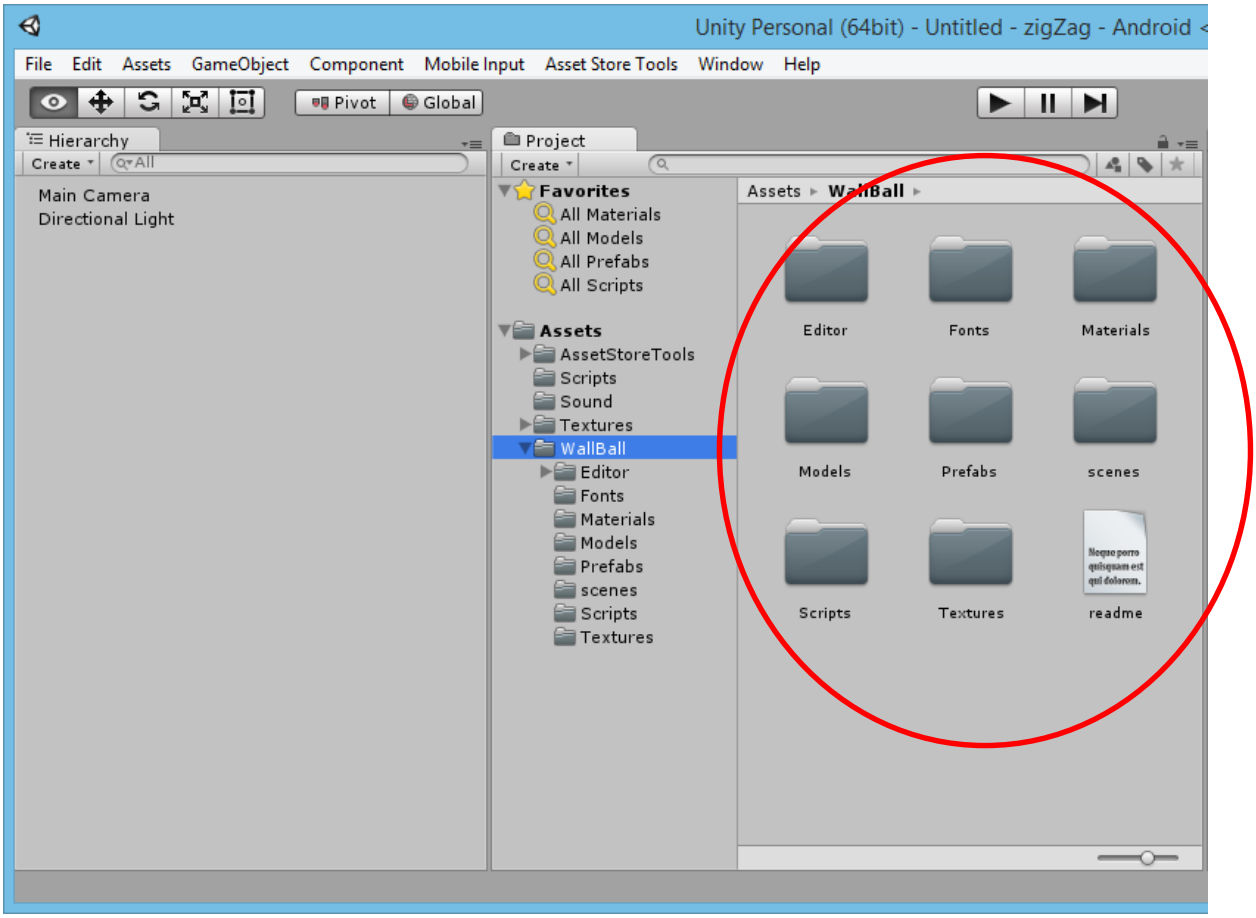


WALL BALL

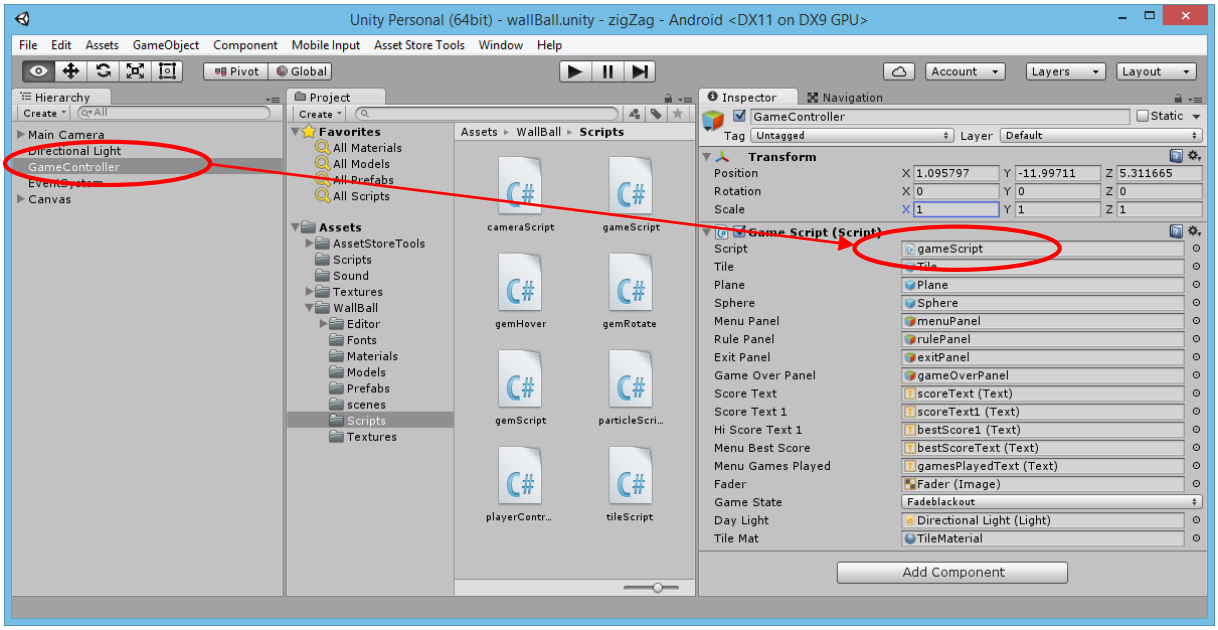
Hello and welcome to the documentation of Project WALL BALL. This document should help you getting started with the project. Opening the project you will find the folder *WallBall*. This folder contains the project files.



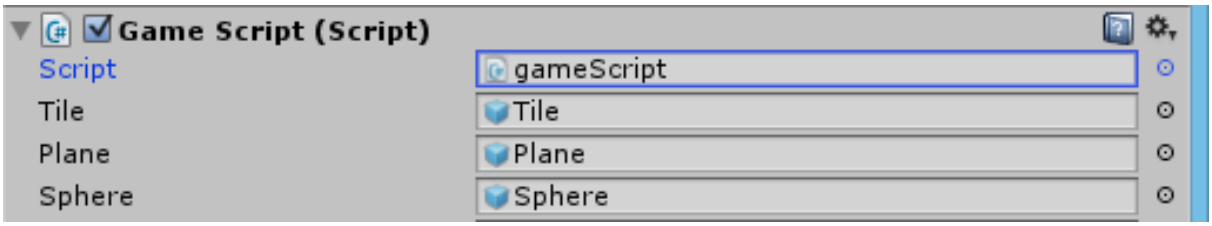
Most important in this structure is the folder *scenes* which contains the game scene.

wallBall

The most important object in scene *wallball* is the GameObject *GameController*. Attached to this object is the script *gameScript*. Made visible to the unity inspector, you can see some fields, of which the most are UI controls.

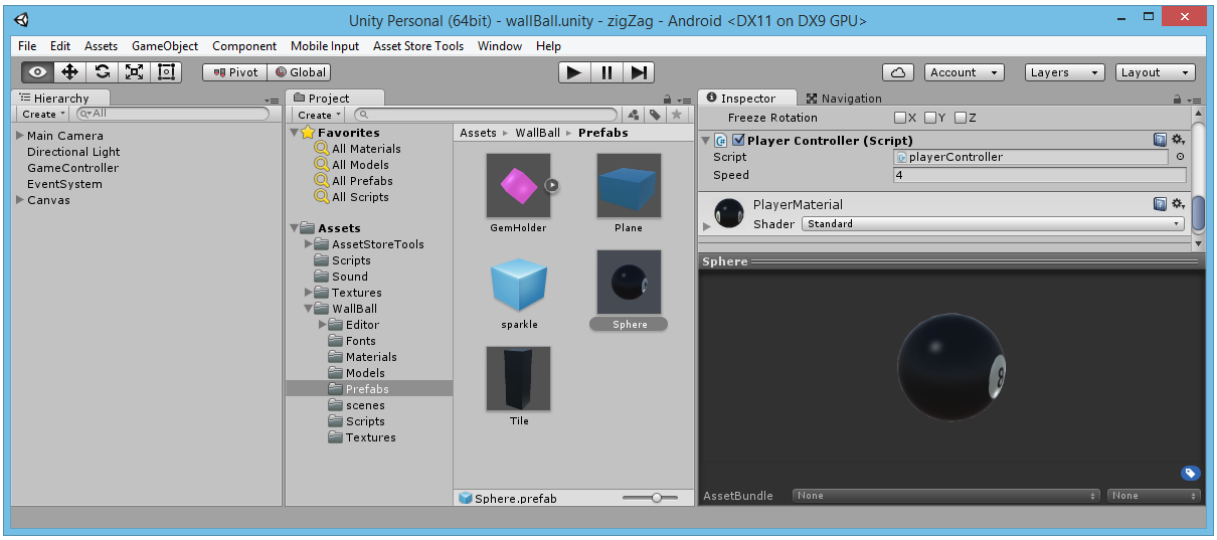


The really important fields for the game are the prefabs *Tile*, *Plane* and *Sphere*. The Game comprises of these prefabs, which will be discussed in detail later on.



Sphere

Sphere is the Player Object and contains the Player Controller Script. Via Inspector you can set the the ball's speed to increase difficulty.



The attached script itself is short and self explanatory.

In essence, the *Start* method tries to get a reference to the main gaming controller attached to the *GameController* object:

```
using UnityEngine;
using System.Collections;

//
// This script controls the player
//

public class playerController : MonoBehaviour {

    // minimal physics and collision detection
    // for the plaxer object
    Rigidbody rigidBody;
    // player can roll in two directions
    bool rollingLeft;
    // player speed.
    // faster means more difficult
    public float speed=2;

    // Reference to the main gaming controller script
    gameScript gameScriptReference;

    // Use this for initialization
    void Start () {
        // get the rigidbody reference
        rigidBody = GetComponent<Rigidbody> ();
        // set starting direction to left
        rollingLeft = true;
        // get reference to the main controller script
        gameScriptReference = GameObject.Find ("GameController").GetComponent<gameScript> ();
    }
```

The *Update* method controls the steering – every time the Space bar is pressed, the balls changes its direction in the X- and Z-Plane. If the player's Y-Position is below 10, the Player dropped and the game is over.

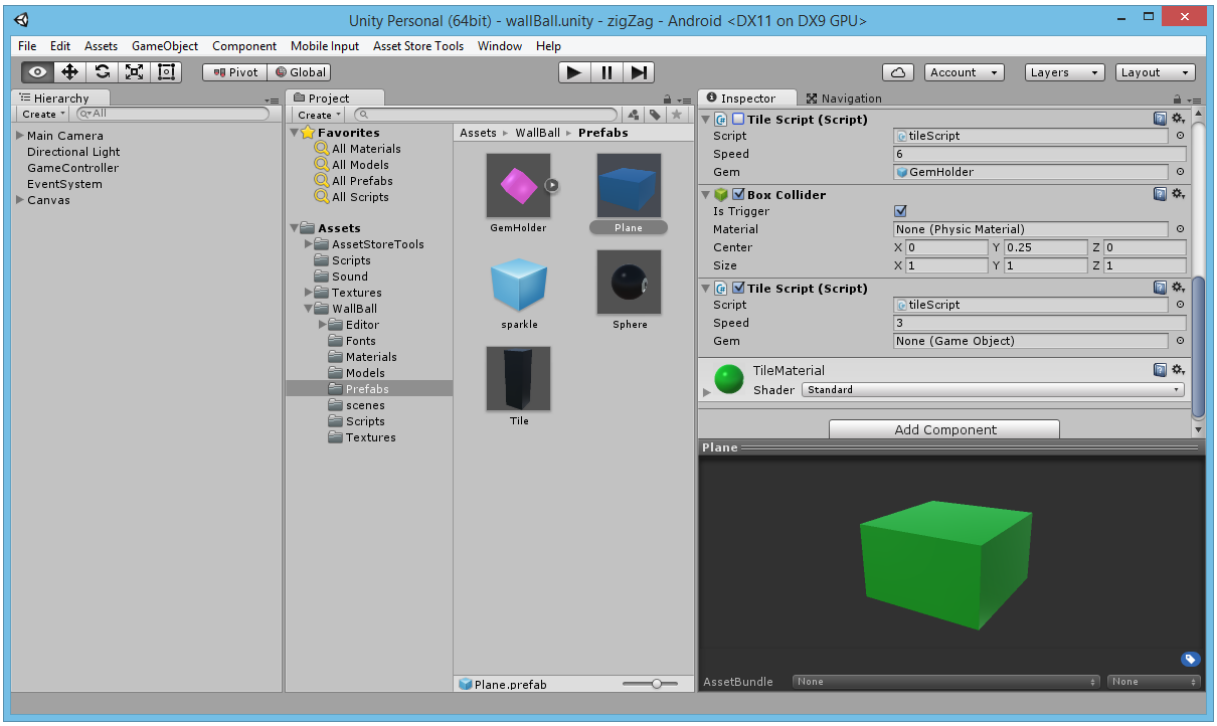
```
    // Update is called once per frame
    void Update () {
        // only update the player if game is in game mode
        if (gameScriptReference.inGame()) {
            // if player presses space key or
            // taps on the screen,
            // change the direction
            if (Input.GetKeyDown (KeyCode.Space) ||
                Input.GetMouseButtonDown(0)) {
                    gameScriptReference.addScore(1);
                rollingLeft = !rollingLeft;
            }
            // player is falling down
            // that means, the game is over
            if (transform.position.y < -10) {
                gameScriptReference.gameOver();
                Destroy(gameObject);
            }
        }
    }
```

Every periodical interaction with the physics engine should be placed in the Method FixedUpdate (according to unity's documentation). In this case, the *FixedUpdate* method controls the player physics by setting the speed to the correct direction. The call to method inGame() makes sure the game is not in menu or pause mode.

```
    void FixedUpdate() {
        // only update the player if game is in game mode
        // ball physics
        if (gameScriptReference.inGame()) {
            if (rollingLeft)
                rigidBody.velocity = new Vector3 (-speed, Physics.gravity.y, 0);
            else
                rigidBody.velocity = new Vector3 (0, Physics.gravity.y, speed);
        }
    }
}
```

Plane

Plane is the second important object. It is the starting point of the Player – a big tile. Attached to it is the script call *TileScript*.



TileScript handles the lifecycle of a tile. Basically, tiles are generated by the GameController script, which we will have a closer look later on. Each Tile comprises of a collider which has the trigger property set. When a player leaves the tile, method *OnTriggerExit* is called. This will set a state variable which causes the tile to fall down smoothly. To do this animation without importing a tweening engine, I needed some variables, which are declared before the methods.

```
using UnityEngine;
using System.Collections;

// This script handles the lifecycle of a tile
// (the blue cubes)

public class tileScript : MonoBehaviour {

    // tiles can fall down
    // start position and end position of fall animation
    Vector3 startMarker;
    Vector3 endMarker;
    // fall speed, may be adjusted at will
    public float speed = 3.0F;
    private float startTime;
    private float journeyLength;
    // I wanted to make the falling tile transparent
    Color startCol, endCol;

    // a tile can have a gem
    public GameObject gem;

    bool triggerLeft;

    float timer;
```

```
    float timerInterval = 0.2f;
```

Method *Start* initializes the plane, setting its start and end position after animation. Despite that, every tile may contain a gem (which the player is supposed to collect).

The generation of a gem is done via a random number generator. With a probability of 40%, a gem is placed every time a new tile is generated (we will have a look at the gem later on). You may change this value at will, to make the game easier or more difficult for the player to collect gems.

Hint: Maybe the player can buy something in the app? A better ball? A bigger ball? A slower ball? A magnetic ball which collects gems automatically?

```
// Use this for initialization
void Start () {
   // Initialize the tile
    triggerLeft = false;
    startMarker = transform.position;
    endMarker = startMarker - new Vector3 (0, 10, 0);

    // generate a random value,
    // with a probability of 40% we will place a gem on the tile
    int myRand = Random.Range (0, 101);
    if (myRand < 40 && gameObject.name != "Plane") {
        GameObject gemGO=Instantiate(gem,transform.position,Quaternion.identity) as
            GameObject;
        gemGO.transform.position += new Vector3(0,2,0);
        gemGO.transform.SetParent(this.transform);
    }
}
```

The code in method *Update* will only be called, if the player set the trigger *triggerleft* in method *OnTriggerExit*. The tile will then gently fall down and be destroyed after 98% of its journey (meaning reaching its y-position at -10).

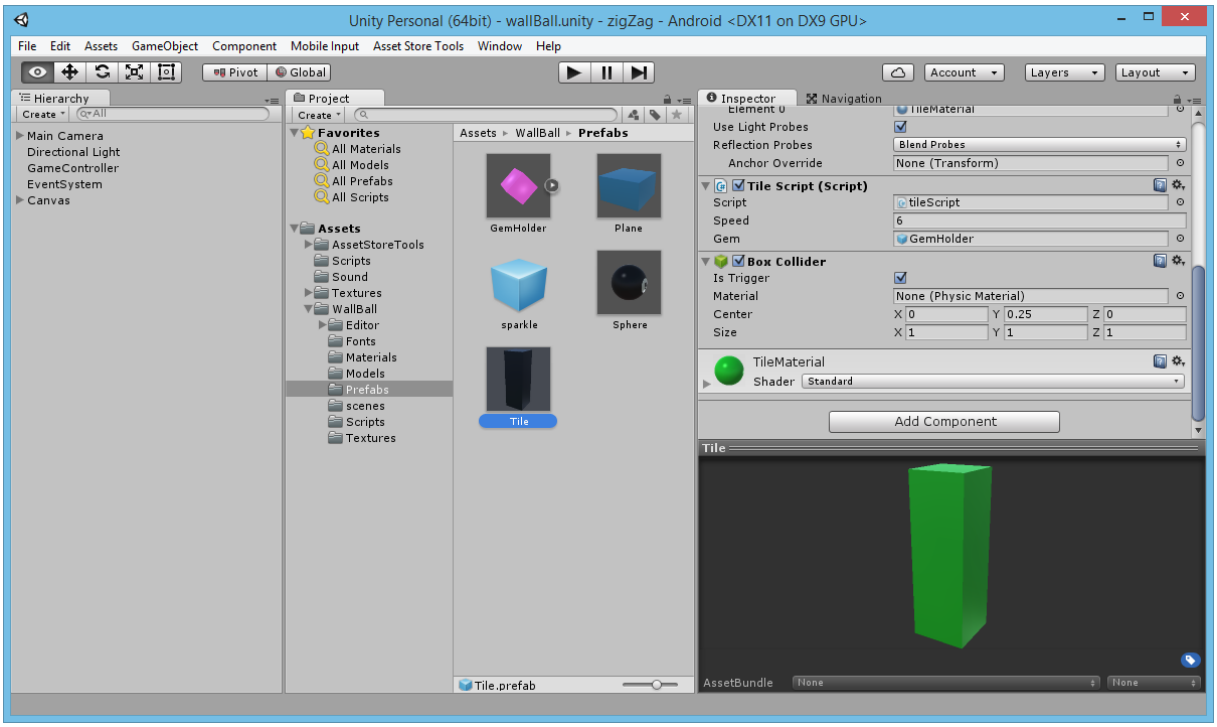
```
// Update is called once per frame
void Update () {
    // only update the falling animation,
    // when player left the tile
    if (triggerLeft) {
        timer += Time.deltaTime;
        float distCovered = (timer-timerInterval)*speed;
        float fracJourney = distCovered / journeyLength;
        transform.position = Vector3.Lerp (startMarker, endMarker, fracJourney);
        if (fracJourney > 0.98f)
            Destroy(this.gameObject);
        }
    }
}
```

When a player leaves the tile, method OnTriggerExit is called. This will set a state variable which causes the tile to fall down smoothly.

```
    /// <summary>
    /// Raises the trigger exit event.
    /// </summary>
    /// <param name="other">Other.</param>
    void OnTriggerExit(Collider other) {
        // The player left a tile
       // now it may fall down
       if (other.name == "Sphere") {
           startTime = Time.time;
           journeyLength = Vector3.Distance(startMarker, endMarker);
           triggerLeft = true;

       }
   }
}
```

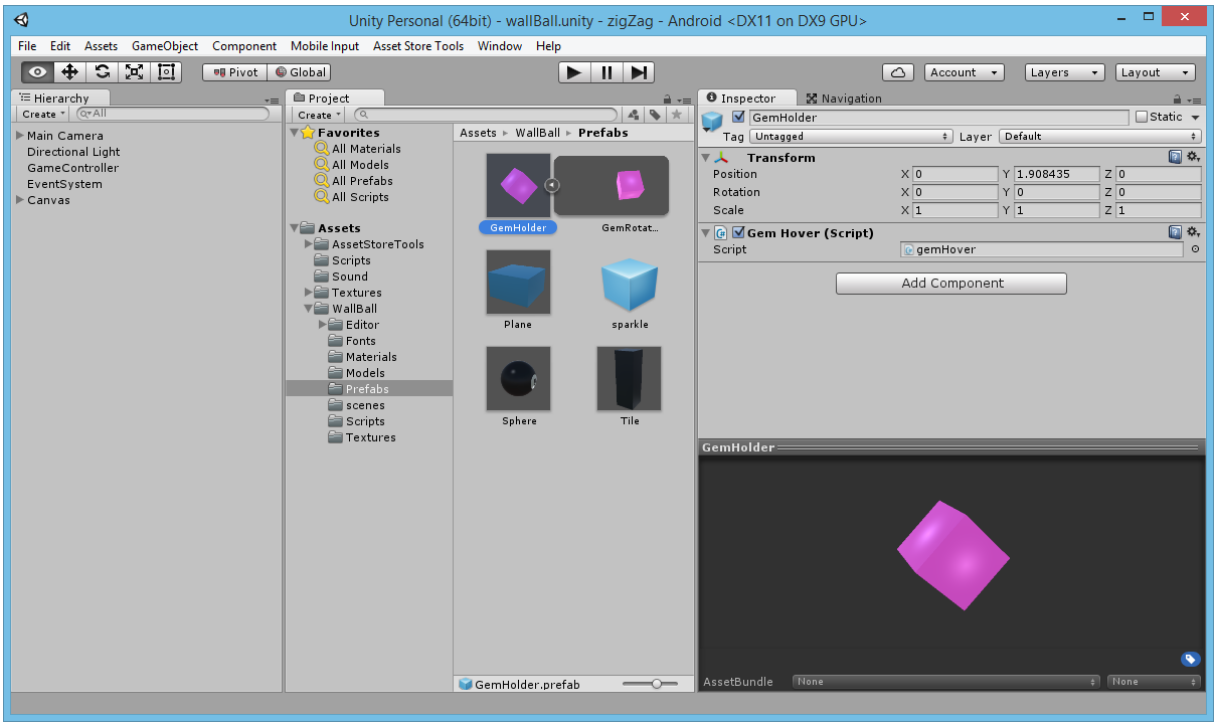
Tile

Tile is the little brother of Plane. It also has the script tileScript attached and embodies the smallest game component.



Gem

The last prefabs used in the game is the *Gem*. To modularize the animation and script design, Gem is composed of nested GameObjects. The parent GameObject is called *GemHolder*.



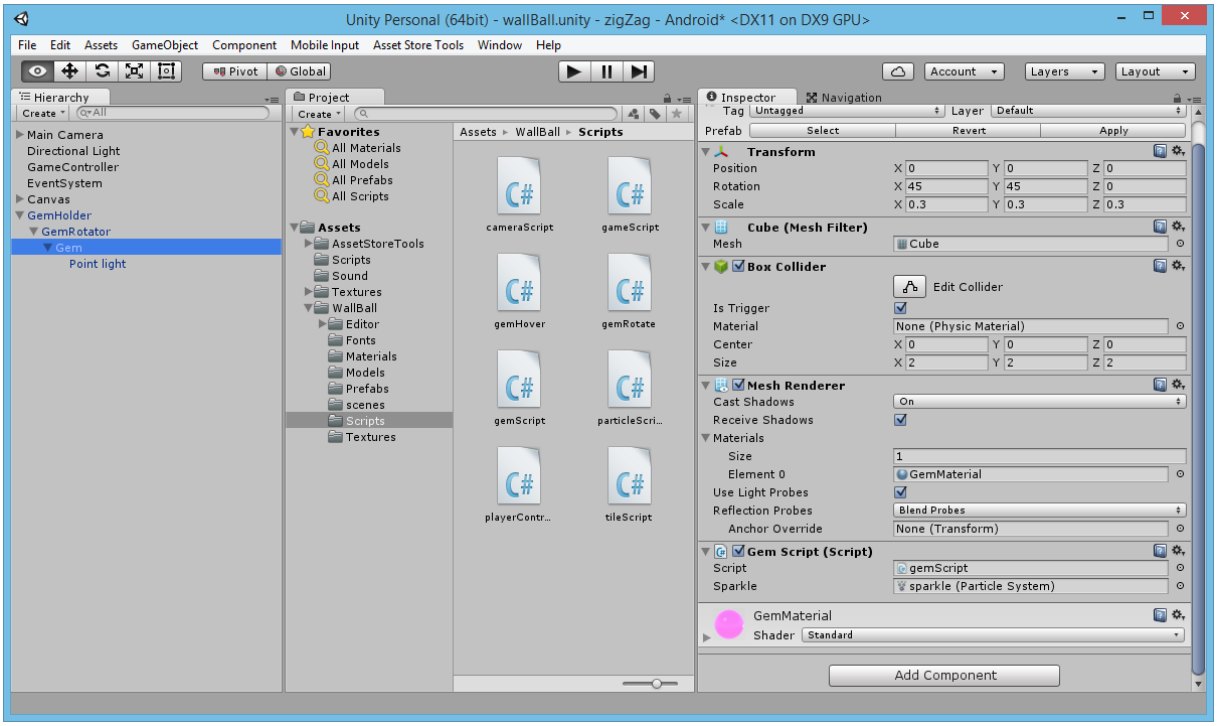
GemHolder has the Script *gemHover* attached to it. This script realizes the gem's hovering up and down. The script itself is quite short and consists only of the method *Update* which moves the gem according to a sine function.

```
using UnityEngine;
using System.Collections;

//
// The gem consists of nested objects
// first of all, the gem is hovering
// this is achieved by a simple sine function

public class gemHover : MonoBehaviour {

    float startY;
    float hover = 0.1f;
    float speed = 2f;

    // Use this for initialization
    void Start () {
        startY = transform.position.y;
    }

    // Update is called once per frame
    void Update () {
        Vector3 position = transform.position;
        position = new Vector3 (position.x, startY + Mathf.Sin(Time.time*speed)*0.1f,
           position.z);
        transform.position = position;
    }
}
```

Nested inside the GemHolder is the *GemRotator*. *GemRotator* contains the script gemRotate. As previously mentioned, this distribution was done to accomplish the separation of degress of freedom. The only task of this script is the rotation around one of the gem's axes.

```
using UnityEngine;
using System.Collections;

//
// this script rotates the gem,
// which is nested in gemHover
//

public class gemRotate : MonoBehaviour {

    float speed = 1f;
    float angle;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        angle = (angle + speed) % 360f;
        transform.localRotation = Quaternion.Euler(new Vector3(0,angle,0));
    }
}
```

Inside the *GemRotator*-GameObject, you finally find the gem object itself. This is a gameobject with the script gemScript and a point light attached to it. Also part of it is a particle system which will sparkle if the gem is collected.



```
using UnityEngine;
using System.Collections;

//
// This script handles the collision detection
// between gem and player
//

public class gemScript : MonoBehaviour {

    // a reference to the main gaming controller script
    gameScript gameScriptReference;

    // sparkle a particel system
    // when player collects a gem
    public ParticleSystem sparkle;

    // Use this for initialization
    void Start () {
        // get the reference to the main gaming script
        gameScriptReference = GameObject.Find ("GameController").GetComponent<gameScript> ();
    }

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

    }

    /// <summary>
    /// Raises the trigger enter event.
    /// </summary>
    /// <param name="other">Other.</param>
    void OnTriggerEnter(Collider other) {
       // gem collided with a plaer object
       if (other.name == "Sphere") {
           // give the player 5 points
           gameScriptReference.addScore(5);
           // instantiate the particle system
           // you can find it in the folder "prefabs"
           GameObject particleSystem = Instantiate(sparkle, transform.position,
             Quaternion.identity) as GameObject;
           // deactivate the gem
           // it will be deleted by the tile which it is parented to
           this.gameObject.SetActive(false);
       }
    }
}
```

As you can see, the script tries to get a reference to the main GameController script. This is done in the *Start* method. To make the gem collectable, its BoxCollider is set to be a trigger.

OnTriggerEvent ist he method which does the magic here. If the player enters the collider/trigger, 5 points are sent to the main GameController script. The particle system is fired and the gem is disabled. It is ot destroyed – that would stop the particle system.

Camera and cameraScript

During the game, the camera follows the player in a certain distance. To realize this behavior, a script called cameraScript is attached to the camera. The script tries to find the player GameObject and gets a refrerence to it. After that, the camera updates its position acoording to the player's actual position. This is done (according to unity's documentation) in the method *LateUpdate* to avoid flickering.

```
using UnityEngine;
using System.Collections;

//
// This Camera script follows the Player Object ("Sphere")

public class cameraScript : MonoBehaviour {

    // Player object's Transform
    public Transform target;

    Vector3 oldPosition;

    // Use this for initialization
    void Start () {
        // set up the camera
        setup ();
    }


    /// <summary>
    /// Sets up the camera
    /// </summary>
    ///
    public void setup() {
       // do we follow a player object, actually?
       // if not, set the values
       if (target == null)
           target = GameObject.Find ("Sphere").transform;
       oldPosition = target.position;
    }


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

    }


    /// <summary>
    /// Updates the camera
    /// Always do camera updates in LateUpdate
    /// </summary>
    ///
    void LateUpdate() {
        if (target != null) {
           Vector3 position = target.position;
           Vector3 delta = oldPosition - position;
           delta.y = 0;
           transform.position = transform.position - delta;
           oldPosition = position;
       }
    }
}
```

GameScript

The Script gameScript is attached to the GameController GameObject and acts as the master controlling game script.

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

// This is the main script attached to the GameObject Gamecontroller

public class gameScript : MonoBehaviour {

    // startposition of the game
    Vector3 startPosition;
    Vector3 actualPosition;
    int myRandom;

    // References to the prefabs
    public GameObject tile;
    public GameObject plane;
    public GameObject sphere;

    // references to the ui panels
    public GameObject menuPanel;
    public GameObject rulePanel;
    public GameObject exitPanel;
    public GameObject gameOverPanel;

    // score display on the ui
    public Text scoreText;
    public Text scoreText1;
    public Text hiScoreText1;

    public Text menuBestScore;
    public Text menuGamesPlayed;

    float timer;
    float timerInterval = 0.3f;

    int score;

    public Image fader;

    // Game may be in various states
    public enum GameState {
       fadeblackout,
       menu,
       game,
       over,
       exit,
    };

    public GameState gameState;

    float fadeTimer;
    float fadeInterval;

    Color col;

    Color camColor;
    Color camColorBlack;
    float camLerpTimer;
    float camLerpInterval = 1f;
    bool camColorLerp = false;
    int direction = 1;

    Color[] tileMatDay;
```

```
Color tileMatNight;
int actTileColor;

public Light dayLight;
public Material tileMat;

float daytimeTimer;
float dayTimeInterval = 15f;
```

Method Awake is called at initialization. This method sets all values which have to be set once before starting the game. The game uses different materials for the tiles and for the day-night-cycle. The color for these materials are fixed in the code. You may change this to some inspector slots to make the game more customizable.

After doing the coloring stuff, the UI Panels are hidden from view except for the menu panel.

```
/// <summary>
/// Awake this instance.
/// </summary>
void Awake() {

    camColor = new Color (0.75f, 0.75f, 0.75f);
    camColorBlack = new Color (0, 0, 0);

    actTileColor = 0;
    tileMatDay = new Color[3];
    tileMatDay[0] = new Color (10/256f, 139/256f, 203/256f);
    tileMatDay[1] = new Color (10/256f, 200/256f, 20/256f);
    tileMatDay[2] = new Color (220/256f, 170/256f, 45/256f);
    tileMatNight = new Color (0, 8/256f, 11/256f);
    tileMat.color = tileMatDay [0];

    // set up the game elements
    setupGame ();
    // set the game state
    gameState = GameState.fadeblackout;
    // hide all panels except for menu panel
    menuPanel.SetActive (true);
    rulePanel.SetActive (false);
    exitPanel.SetActive (false);
    gameOverPanel.SetActive (false);
}
```

The game uses PlayerPrefs to store the high score and the number of games the player already played. To make sure these PlayerPrefs are present, they are queried and set in the start method.

```
// Use this for initialization
void Start () {
    // set the hiscore value
    if (!PlayerPrefs.HasKey ("HiScore"))
        PlayerPrefs.SetInt ("HiScore", 0);
    if (!PlayerPrefs.HasKey ("GamesPlayed"))
        PlayerPrefs.SetInt ("GamesPlayed", 0);
}
```

Update mainly control the keyboard input and the different gamestates. Independent of the gamestate the application is in, a press on „Escape“ (Home key on smartphones) will cause the app to exit.

```
// Update is called once per frame
void Update () {
    // Escape key for android smartphones
    if (Input.GetKeyDown (KeyCode.Escape)) {
        if (gameState == GameState.game)
            Time.timeScale = 0;
        gameState = GameState.exit;
        exitGame();
    }
```

After that, the general gamestate handling is done. The game start in the gamestate *fadeblackout.* This meand, that the game starts by fading a black image out.

```
    switch (gameState) {
        case GameState.fadeblackout:
            col = fader.color;
            col.a -= 0.01f;
            if (col.a <= 0) {
                col.a = 0;
                fader.gameObject.SetActive(false);
                gameState = GameState.menu;
            }
            fader.color = col;
            break;
```

If the fading is done, the app changes to the gamestate *menu*. Here, the game waits for button presses. Due to UI handling in unity 4.6 and above, the actions performed are outsourced in special methods.

```
        case GameState.menu:
            break;
```

In the gamestate game, the application does three things. First, it builds new tiles after a fix interval. The interval time may be adjusted by changing the value of varriable timerInterval at the top of the script. The new tiles are built by calling *buildTile()*.

```
        case GameState.game:
            // generate further tiles the player may roll on
            timer += Time.deltaTime;
            if (timer >= timerInterval) {
                timer -= timerInterval;
                buildTile ();
            }
```

Second, a day-night-cycle is implemented to make the game more interesting. This is done by color interpolation.

```
            daytimeTimer += Time.deltaTime;
            if (daytimeTimer > dayTimeInterval) {
                daytimeTimer -= daytimeTimer;
                camColorLerp = true;
                camLerpTimer = 0;
            }
```

```
            if (camColorLerp) {
                camLerpTimer += Time.deltaTime;
                float percent = camLerpTimer / camLerpInterval;
                if (direction == 1) {
                    Camera.main.backgroundColor =
                       Color.Lerp(camColor,camColorBlack,percent);
                    tileMat.color = Color.Lerp(tileMatDay[actTileColor],
                       tileMatNight,percent);
                    dayLight.intensity = 1-percent;
                }
                else {
                    Camera.main.backgroundColor =
                       Color.Lerp(camColorBlack,camColor,percent);
                    tileMat.color = Color.Lerp(tileMatNight,
                       tileMatDay[actTileColor],percent);
                    dayLight.intensity = percent;
                }
```

If the night cycle is over, the game chooses a new tile color and changes the material. This is done by a random number generator.

```
                if (percent > 0.98f) {
                    camLerpTimer = 1;
                    direction *= -1;
                    camColorLerp = false;
                    if (direction == -1)
                        actTileColor = Random.Range(0,tileMatDay.Length);
                }
            }

            break;
        case GameState.over:
            break;

    }
}
```

Since the game itself consists of only one scene, some GameObjects have tob e reset after a player loses or quits a game. This work is done by *setupGame()*. I don't want to talk about this method in detail – the code is commented and speaks for itself.

```
/// <summary>
/// Setups the game.
/// </summary>
void setupGame() {
    // clear the score
    score = 0;
    // set the ui
    scoreText.text = score.ToString ("D5");
    scoreText1.text = score.ToString ("D5");

    menuBestScore.text = "BEST SCORE: " + PlayerPrefs.GetInt ("HiScore").ToString();
    menuGamesPlayed.text = "GAMES PLAYED: " + PlayerPrefs.GetInt
       ("GamesPlayed").ToString();

    // clean up old tile from potential previous game
    GameObject[] tiles = GameObject.FindGameObjectsWithTag ("Tile");
    foreach (GameObject t in tiles)
        Destroy (t);

    // generate new starting plane
    GameObject planeGO = Instantiate (plane, new Vector3 (0, -2, 0), Quaternion.identity)
        as GameObject;
    planeGO.name = "Plane";
```

```
    // generate the player object
    GameObject sphereGO = Instantiate (sphere, new Vector3 (0, -0.1f, 0),
        Quaternion.Euler(45,63,23)) as GameObject;
    sphereGO.name = "Sphere";

    // set the camera object
    Camera.main.transform.position = new Vector3 (5, 5, -5);
    Camera.main.GetComponent<cameraScript> ().setup ();

    // set the starting position for the blue tiles
    startPosition = new Vector3(-2,-2,3);
    GameObject newTile=Instantiate(tile,startPosition,Quaternion.identity) as GameObject;
    newTile.name = "Tile";
    actualPosition = startPosition;

   // now generate 20 tile in advance
    for (int i = 0; i < 20; i++) {
        buildTile ();
    }
}
```

BuildTile() is a core method in this application. Here a random number generator is used to generate a new tile the player can roll on. With a probability of 50% the new tile is set tot he left or the right. Instantiating a new tile will call the script tileScript attached to the tile. We already discussed that script – it will handle the creation oft he gems, if neccessary.

```
/// <summary>
/// Builds the tile.
/// </summary>
void buildTile() {
    Vector3 newPosition = actualPosition;
    myRandom = Random.Range(0,101);

    // with a probability of 50% set the new tile left or right
    if (myRandom < 50) {
        newPosition.x -= 1;
    }
    else {
        newPosition.z += 1;
    }
    actualPosition = newPosition;
    GameObject newTile=Instantiate(tile,actualPosition,Quaternion.identity) as GameObject;
    newTile.name = "Tile";
}
```

Methods in subordinate scripts make use of the method *inGame()*. inGame returns the current state: is the application currently in game mode or not.

```
// for other instances
// check, if game is still in game mode
public bool inGame() {
    if (gameState == GameState.game)
        return true;
    else
        return false;
}
```

AddScore() is called in script gemScript, when a player enters a gem's trigger. To handle the scoring intelligence in the main script, the already discussed references to the main GameController are neccessary.

```
// add points to the player score
public void addScore(int amount) {
   score += amount;
    scoreText.text = score.ToString ("D5");
    scoreText1.text = score.ToString ("D5");
}
```

GameOver() is called when the player leaves or loses the game. The game logic checks if the actual score exceeds the highscore (saved in a PlayerPref) and changes it, if neccessary.

```
// player lost the game
// display the gameover panel
// check the score and the actual hiScore
public void gameOver() {
    gameState = GameState.over;
    gameOverPanel.SetActive (true);
    int hiScore = PlayerPrefs.GetInt ("HiScore");
    if (score > hiScore)
        PlayerPrefs.SetInt ("HiScore", score);

    scoreText1.text = score.ToString ("D5");
    hiScore = PlayerPrefs.GetInt ("HiScore");
    hiScoreText1.text = hiScore.ToString ("D5");

    int gamesPlayed = PlayerPrefs.GetInt ("GamesPlayed");
    gamesPlayed++;
    PlayerPrefs.SetInt ("GamesPlayed", gamesPlayed);
}
```

Last but not least, there are event methods which are triggered when the player presses a button on the UI-Panels. These methods are straightforward and will not be discussed in detail.

```
/// <summary>
/// Now following the ui button events.
/// </summary>

public void playGame() {
   menuPanel.SetActive (false);
    gameState = GameState.game;
}

public void showRules() {
    menuPanel.SetActive (false);
    rulePanel.SetActive (true);
}

public void exitGame() {
    menuPanel.SetActive (false);
    gameOverPanel.SetActive (false);
    exitPanel.SetActive (true);
}

public void exitRules() {
    menuPanel.SetActive (true);
    rulePanel.SetActive (false);
    gameState = GameState.menu;
}
```

```
    public void resumeGame() {
        exitPanel.SetActive (false);
        if (Time.timeScale == 0) {
            Time.timeScale = 1;
            gameState = GameState.game;
        }
        else {
            if (GameObject.Find("Plane") == null)
                setupGame();
            menuPanel.SetActive (true);
        gameState = GameState.menu;
        }
    }

    public void retryGame() {
        setupGame ();
        menuPanel.SetActive (true);
        gameOverPanel.SetActive (false);
        gameState = GameState.menu;
    }

    public void quitGame() {
        Application.Quit ();
    }
}
```

I hope you find this documentation useful. Please let me know if something is missing or unclear. You should now be able to customize the project at your will.