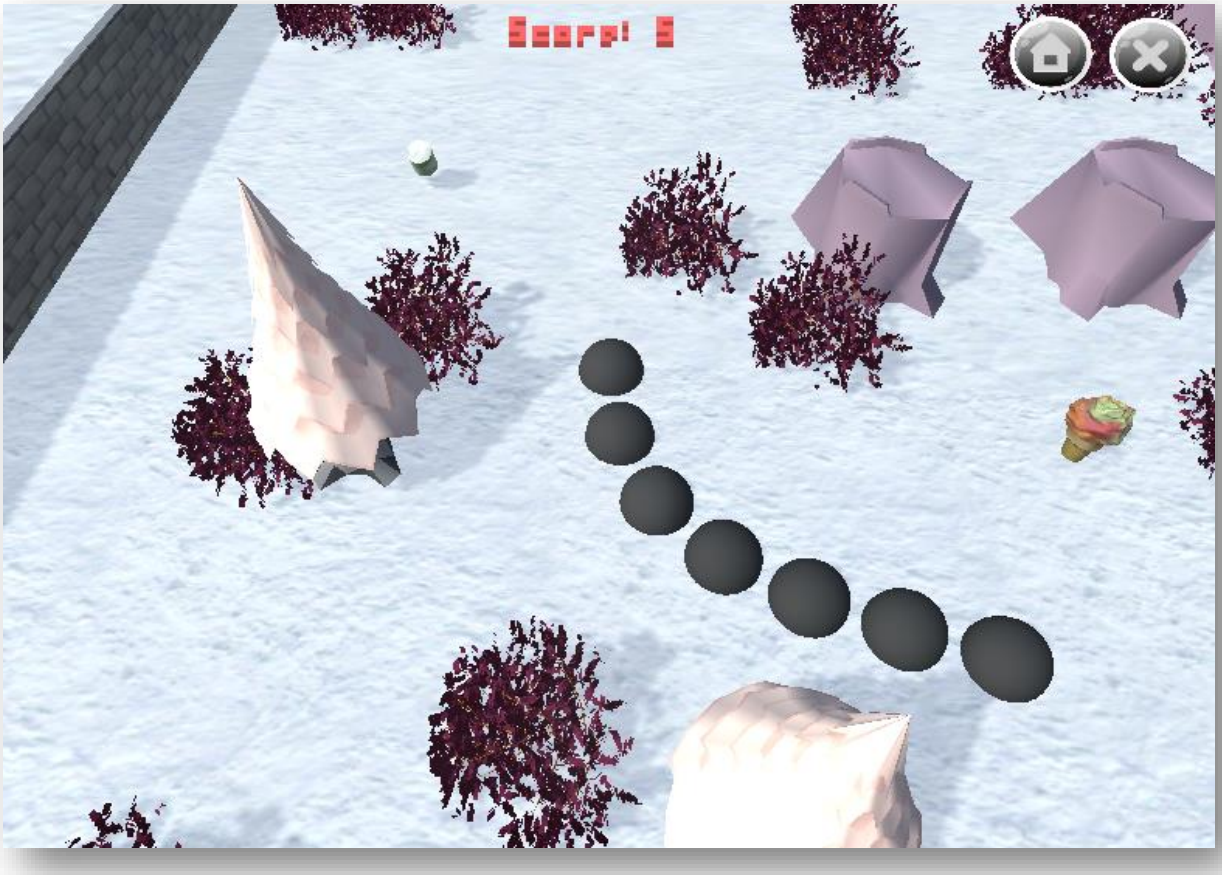
SNAKES IN DREAM

REALIZACJA PROSTEJ GRY TYPU „SNAKE” NA PLATFORMĘ ANDROID W JĘZYKU C#

Spis treści

WSTĘP

Gra działa na platformie Android i nosi nazwę „Snake in Dream”.



Gra polega na zjadaniu różnych smakołyków, które pojawiają się w losowym miejscu na mapie. Wąż Bob (bo tak ma na imię), może poruszać się w dwóch kierunkach tzn. lewo i prawo, a zwierzęcy instynkt nakazuje mu cały czas ruszać na przód w poszukiwaniu jedzenia i nawet jeśli stoi przed nim przeszkoda nie zatrzyma się.

Omijanie przeszkód i zjadanie smakołyków – to główne zadania gracza sterującego Bobem.

Za każdy zjedzony smakołyk gracz dostaje punkty, jednak gra byłaby zbyt łatwa gdyby wąż nic więcej nie robił.

Podczas gdy liczba punktów rośnie, wąż przyspiesza o małą wartość, tym samym dając graczowi wyzwanie.

Oprócz pojawiającego się jedzenia, podczas uruchomienia planszy na mapie rozstawiają się przeszkody w postaci pni drzew, choinek i drzew.

Gdy wąż napotka przed sobą przeszkodę i uderzy w nią, gracz zostanie poinformowany o tym stosowną informacją.

Cała aplikacja została stworzona na silniku Unity3D, z wykorzystaniem języka C#.

Proces tworzenia całej aplikacji zajął około dwa tygodnie, głównie z powodu średniej znajomości języka C#.

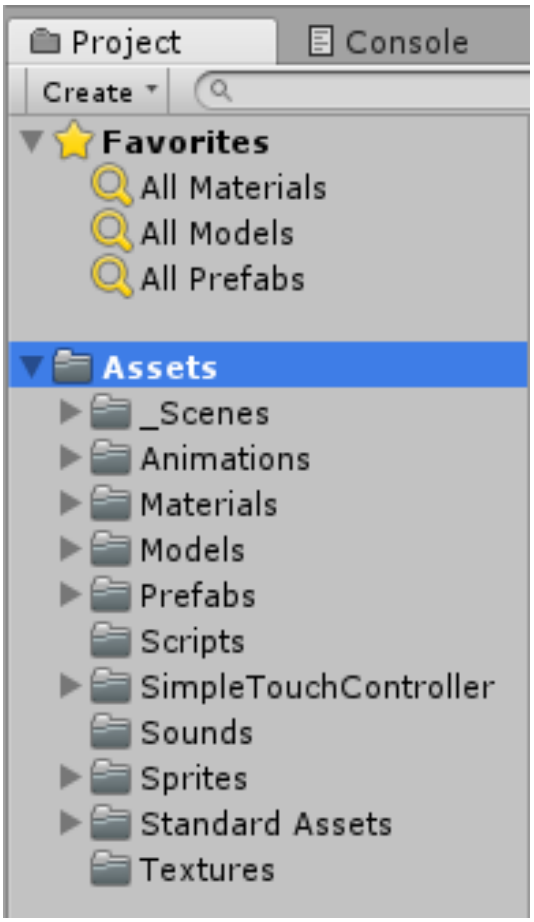
Dzięki temu projektowi wgłębiłem się w ten język, stopniowo poznając jego zalety oraz wady.

Doświadczenie to, dało mi również porównanie między procesem tworzenia gry w Unity3D wykorzystując C#, a językiem Java z frameworkiem libGDX, który ostatnio zrodził we mnie zaciekawienie.

Dla ciekawskich informacje na temat wyżej wymienionej biblioteki poniżej:

http://libgdx.badlogicgames.com/

ZAWARTOŚĆ PROJEKTU



Obrazek przedstawiony obok prezentuje zawartość projektu.
Folder *Scenes* – jak sama nazwa wskazuje zawiera sceny projektu.
Folder *Animations* – zawiera animacje – w tym przypadku poruszający się napis w menu gry.
Folder *Materials* – Przechowuje wszystkie pliki z rozszerzeniem .mat.
Folder *Models* – Przechowuje modele. Między innymi modele jedzenia dla węża i przeszkody.
Folder *Prefabs* – Przechowuje pliki z rozszerzeniem *prefab*.
Folder *Script* – Przechowuje skrypty wykorzystywane w projekcie.
Folder *SimpleTouchController* – Jest to Asset importowany ze strony Unity3D, który obsłuży sterowanie dotykiem.

Folder *Sounds* – Przechowuje dźwięki gry.

Folder *Sprites* – Przechowuje między innymi ikony służące do przemieszczania się między scenami.

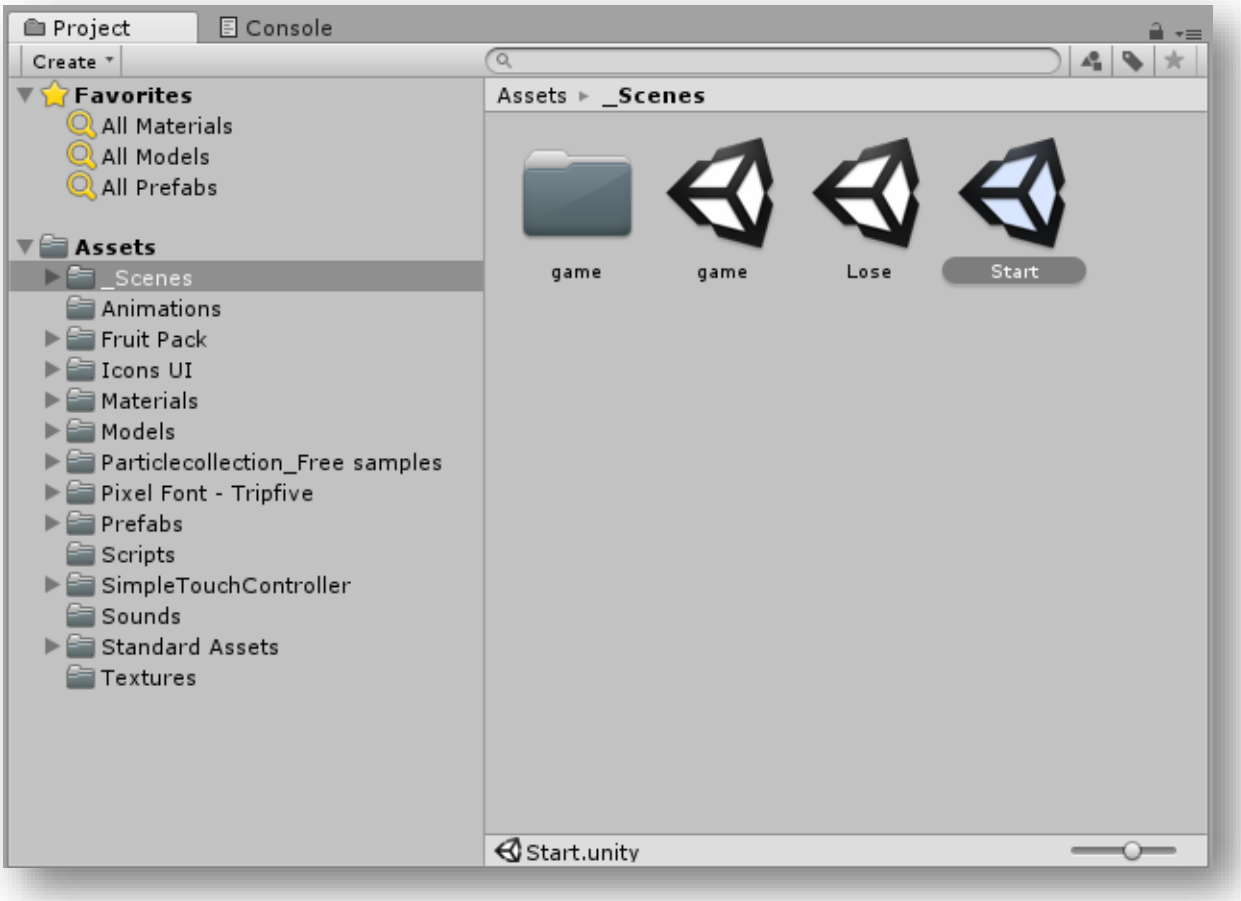
Folder *Standard Assets* – Przechowuje domyślną zawartość - między innymi czcionki itp.

Folder *Textures* – Przechowuje tekstury wykorzystywane w projekcie.

PROJEKTOWANIE SCEN

W tym rozdziale zostanie opisany proces projektowania scen w całej aplikacji.

Sceny znajdują się w folderze „_Scenes”, znajdującym się w głównym folderze z wszystkimi plikami.



Pierwsza Scena jak nazwa wskazuje, jest to scena pt. „Start”.

Jest to scena, która zawiera całą konstrukcję menu gry.

Menu gry jest bardzo proste i intuicyjne, co przykuwa uwagę gracza.

Poniższe zdjęcie prezentuje widok po włączeniu gry.

MENU GRY

Za menu gry odpowiedzialny jest obiekt Canvas, w którym umieszczone są przyciski.

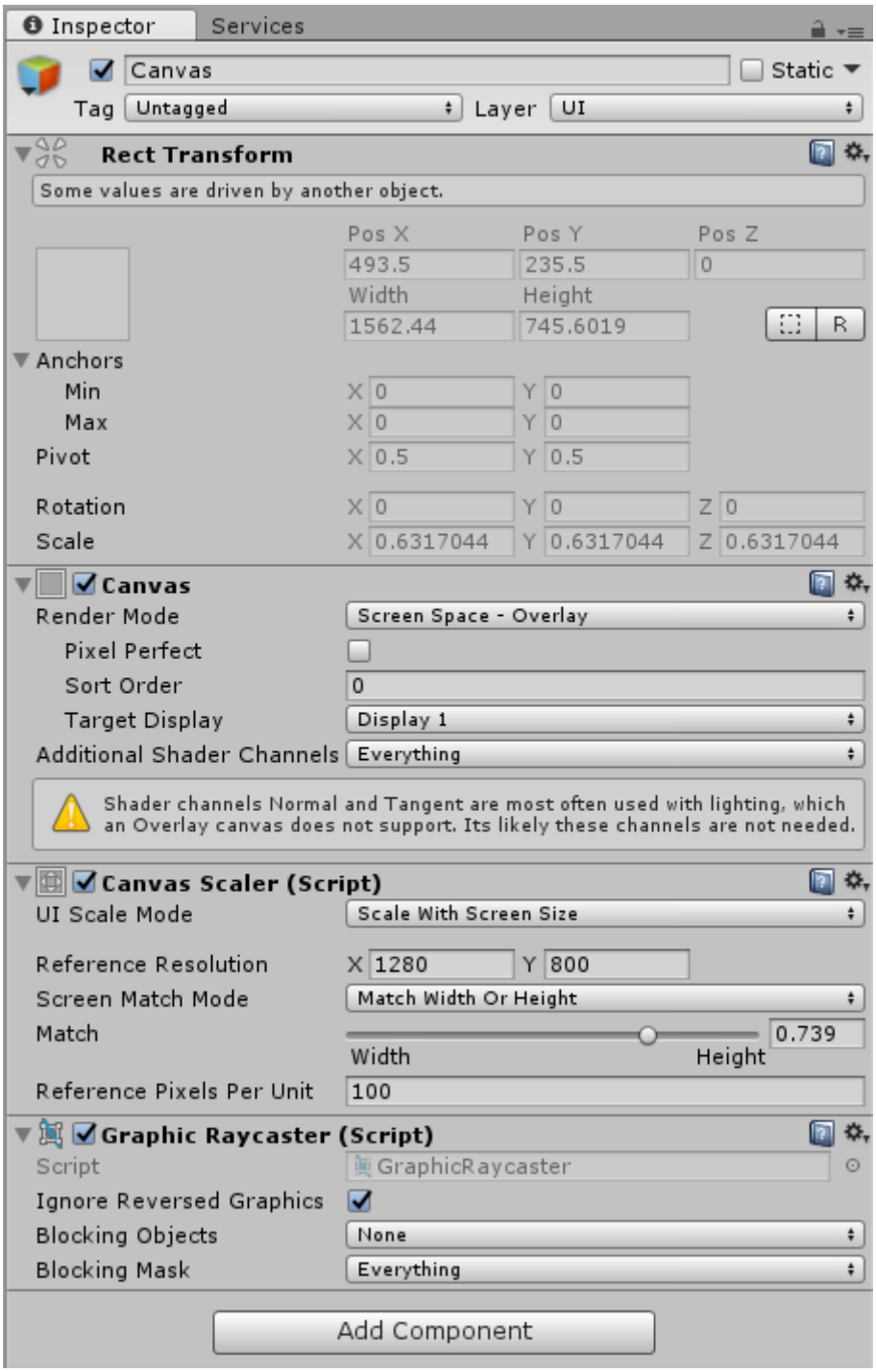
W obiekcie Canvas znajdują się 2 przyciski – „*Tap to start a game*”, „*Exit*”

oraz napis tytułowy – „*Snake 3D*” z dołączoną animacją pt. „*Menu*”.

Animacja ta porusza napis w górę i w dół co sprawia wrażenie, że gra jest warta każdych pieniędzy.

Prezentowana czcionka ma nazwę – „*Pixel Font – Tripfive*”.

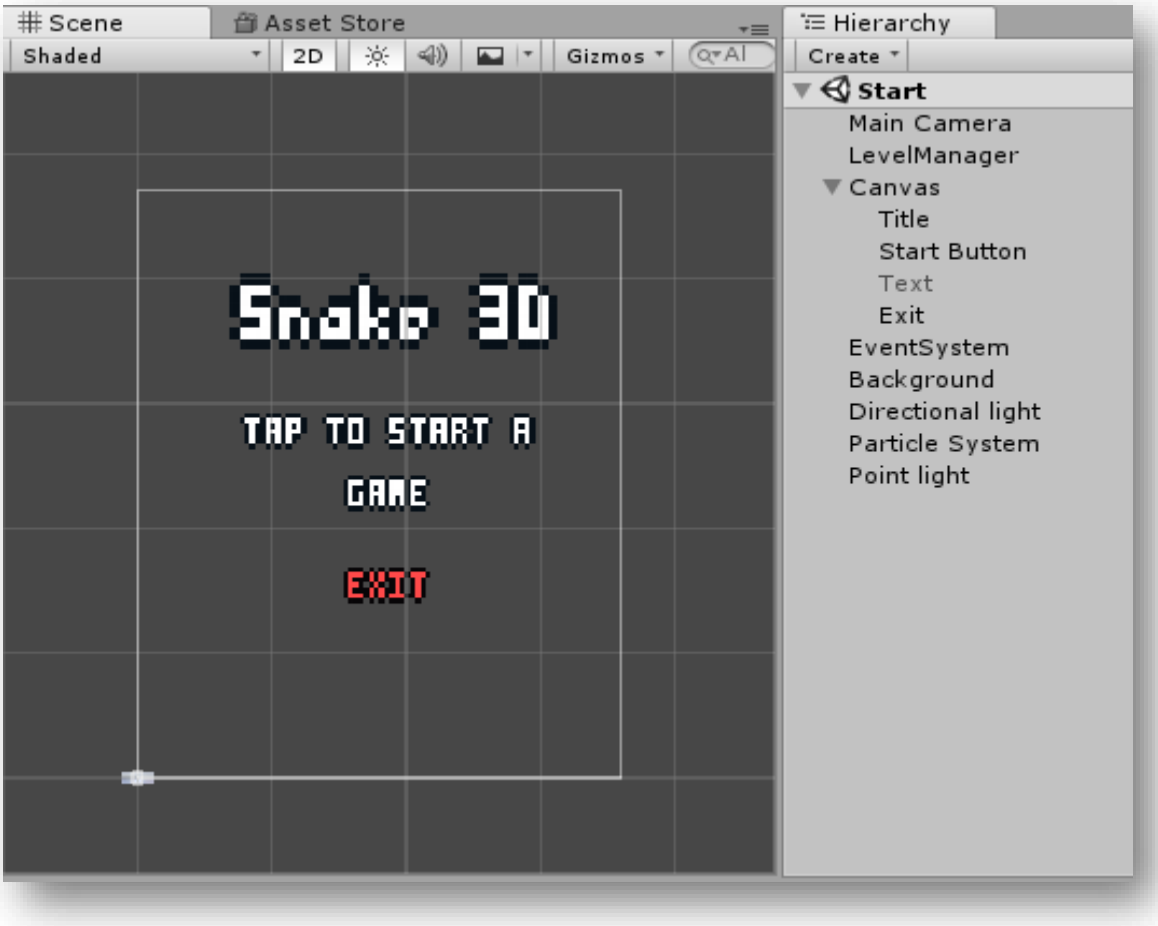
Konfiguracja obiektu „Canvas” wygląda następująco:



Ważną opcją tego obiektu jest „*Scale With Screen Size*”.

Dzięki takiej konfiguracji, menu doskonale dostosowuje się do rozdzielczości dowolnego urządzenia mobilnego z systemem Android.

Widok sceny wygląda w następujący sposób:



Jak widać na powyższym zdjęciu przedstawiającym zawartość sceny,

w zakładce „Hierarchy” są następujące elementy:

- „Main Camera”

Jest to główna kamera – widok z perspektywy gracza

- „Level Manager”

Jest to obiekt z dołączonym skryptem pt. „LevelManager”, który działa na zasadzie przechwycenia momentu kliknięcia przez gracza „Tap to start a game”.

Oprócz przeniesienia gracza do drugiej sceny pt. „game”, wyświetla nieaktywny jeszcze napis „Loading”, który jest zawarty w obiekcie Canvas pt. „Text”.

Typ sposobem powstaje swego rodzaju loading screen, który będzie widoczny aż do zmiany sceny.

- „Canvas”

Jest to obiekt należący do głównej kategorii dostępnych w Unity obiektów „UI”. Umożliwia on nałożenie w wolnym polu sceny, charakterystycznego obszaru wydzielonego na poszczególne komponenty uzupełniające menu.

- „EventSystem”

Odpowiedzialny za każdy click w grze.

- „Background”

Jest to zwyczajny obiekt „Cube”, z dołączoną teksturą tła menu.

- „Directional light”

Jest to główne światło kierunkowe, które widoczne jest w obrębie całej sceny.

- „Particle System”

Particles – tzn. cząsteczki bądź też innymi słowy efekt w grze. W tym przypadku imitacja śniegu.

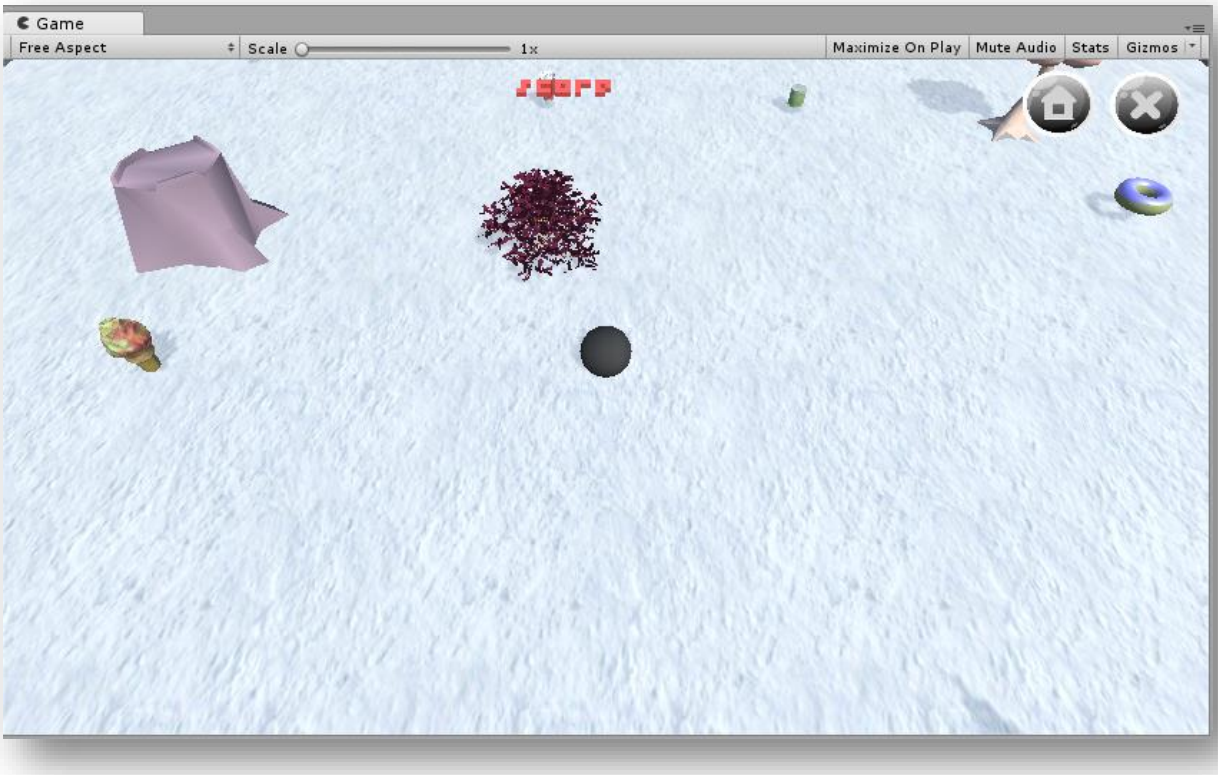
- „Point light”

Światło punktowe, odpowiadające za podświetlenie odrębnej części menu.

PLANSZA

Scena odpowiedzialna za działanie planszy, jak nazwa wskazuje, zawarta jest w scenie pt. „game”. Jest to scena, która zawiera całą konstrukcję głównego levelu gry.

Poniższe zdjęcie prezentuje widok gry.



Cała mapa stworzona jest na podstawie obiektu „Plane” z teksturą śniegu, a dalsza część mapy będąca jednocześnie niedostępna dla gracza, to obiekt „Terrain” wygenerowany głównie dla poprawienia estetyki.

W tym przypadku, wykorzystywany obiekt „Plane”, nosi nazwę Ground, a dalsza jego specyfikacja będzie opisana w późniejszym etapie.

Jest on widoczny dopiero po przybliżeniu się gracza do ściany.

Drugą rzeczą, która znacząco przykuwa uwagę, jest napis „Score” na górnej - środowej części ekranu.

Jest to nic innego jak punktacja inkrementowana wraz ze zjedzonymi smakołykami.

Oprócz punktacji, to po prawej stronie widnieją dwa przyciski, które odpowiadają za przemieszczanie się między scenami.

Pierwszy od lewej to przeniesienie do menu gry, a drugi to zamknięcie aplikacji.

Po długiej analizie dostępnych sposobów na przedłużanie węża, najlepszym sposobem okazało się stworzenie go z obiektu „Sphere”, czyli szarej kuli i dodawanie do niego kolejnych tego typu obiektów za każdym zjedzeniem przysmaku.

Każdy nowy element węża, jest obiektem znajdującym się w folderze pt. „Prefabs” i nosi nazwę „BodyPart”.

Na zdjęciu poniżej widnieje zawartość sceny, wraz z listą elementów w zakładce „Hierarchy”.

Oprócz przedstawionych już we wcześniejszej scenie obiektów, pojawiły się tutaj nowe, takie jak:

- Default

Gumowy misiek dodający charakteru grze

- Tree/Pintree

Drzewa choinki wygenerowane w losowym miejscu na mapie.

- Ground

Obiekt „Plane” służący jako podłoga dla Bobka.

- Walls

Ściany, służące jednocześnie jako klatka dla Bobka.

- Camera Controller

Obiekt przechowujący skrypt odpowiadający za zachowanie kamery podczas zmiany kierunku poruszania się węża.

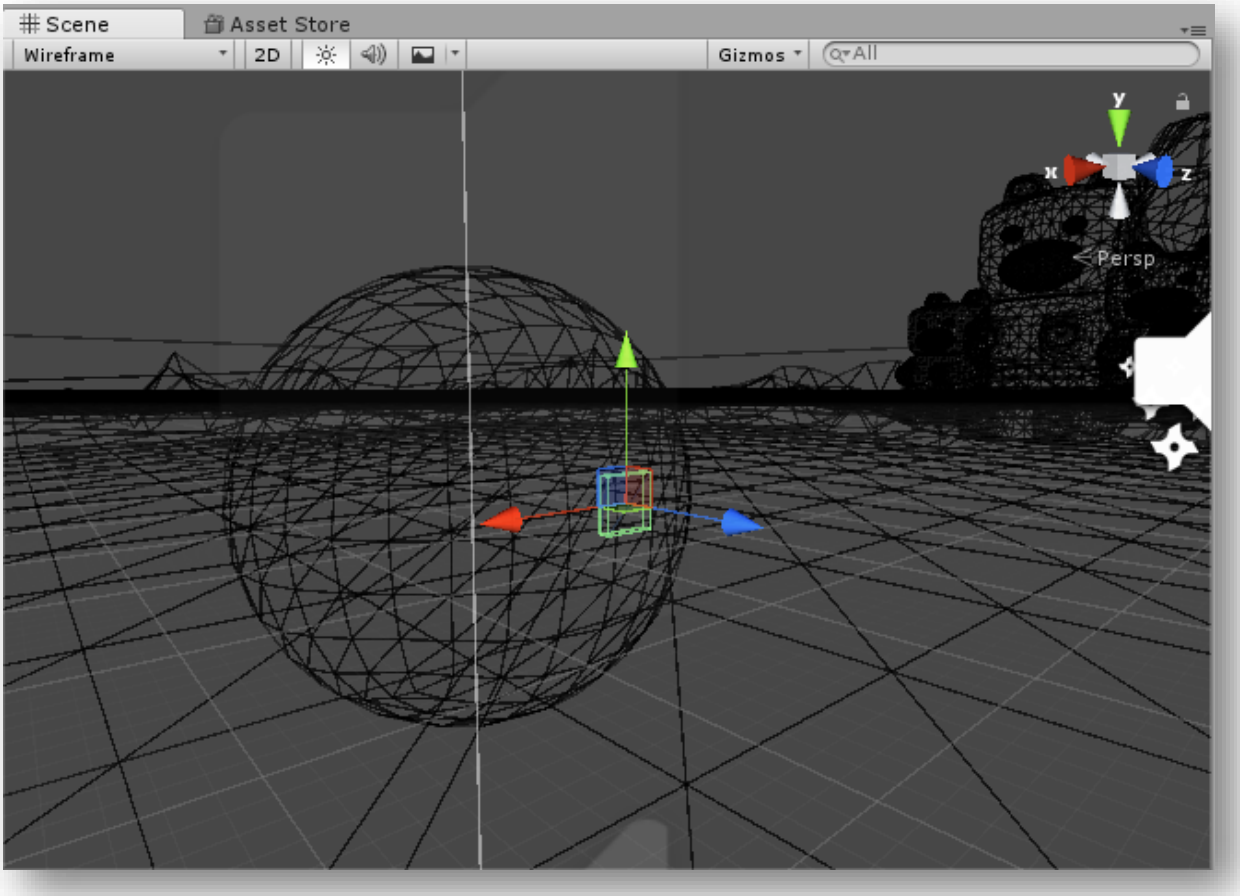
(W nowszej wersji usunięty)

- Snake

Cała kategoria Snake.

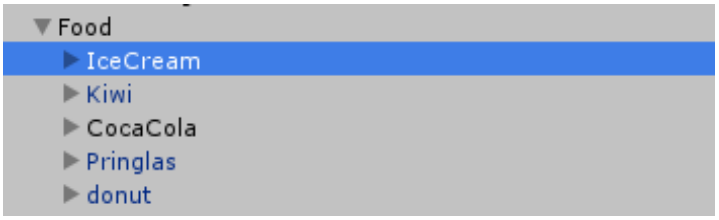
Przechowuje obiekty takie jak głowa (obiekt kuli), kamera i czoło (obiekt „Forhead” umieszczony na przodzie obiektu węża).

Jest to mała kostka odpowiedzialna za wykrywanie kolizji węża.



- Food

Kategoria przechowująca obiekty jedzenia i efekty particles.

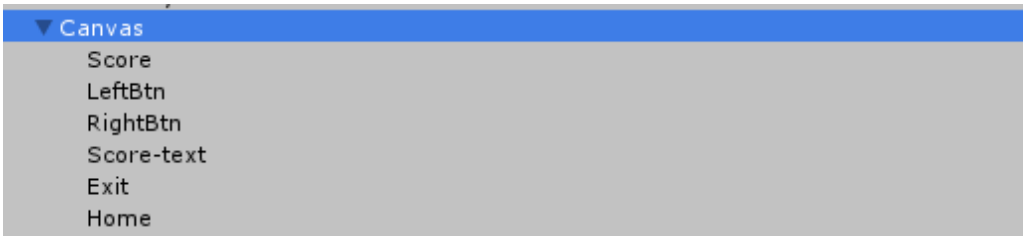


(Lody, Kiwi, CocaCola, Pringlas, Donuty)

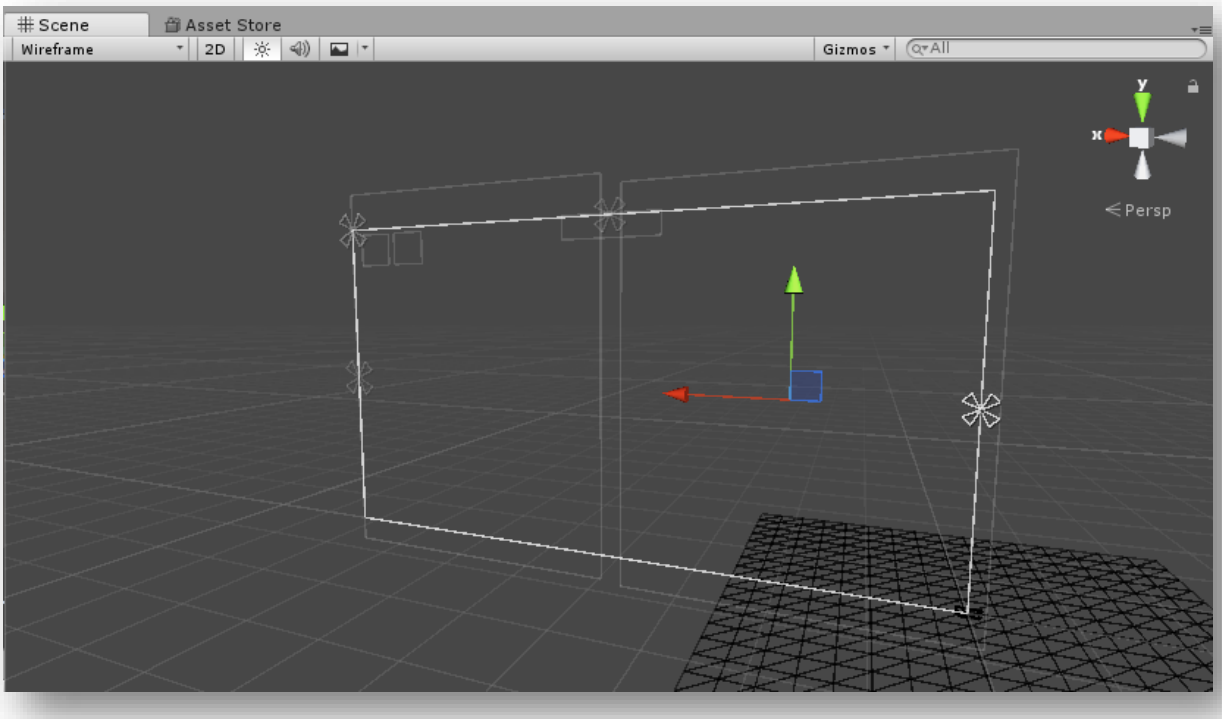
- Canvas

Płótno Canvas w scenie „Game”, przechowuje dodatkowo

*Przyciski **lewo „LeftBtn”/prawo „RightBtn”**, odpowiadające za sterowanie Bobem.*



Scena prezentująca rozmieszczenie elementów Canvas:



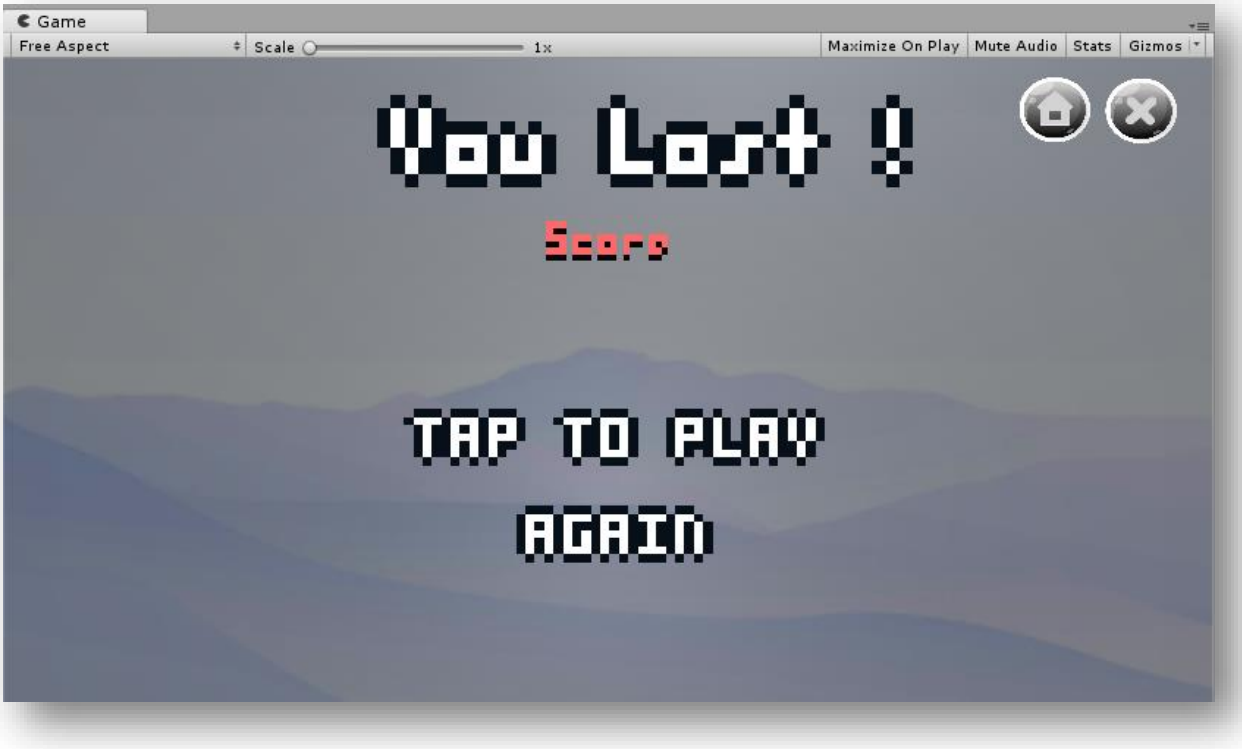
- Cube

Ogrzewane siedziska dla gumowych misiów.

SCENA PRZEGRANEJ GRY

Scena przegranej gry pt. „Lose”, pojawia się przed oczami gracza po kolizji z obiektami, które odpowiadają za przeszkody.

Gracz przekierowany do takie sceny, zobaczy widok przedstawiony poniżej:



Oczywiście, napis „Score”, zostaje automatycznie uzupełniony ilością zebranych punktów przez gracza.

W tym przypadku jest to tylko podgląd nieuruchomionej jeszcze gry.

Podgląd sceny i lista elementów wygląda w następujący sposób:

Jak widać, nie jest to skomplikowany układ, a działa w sposób wystarczający do swobodnej i przyjemnej gry.

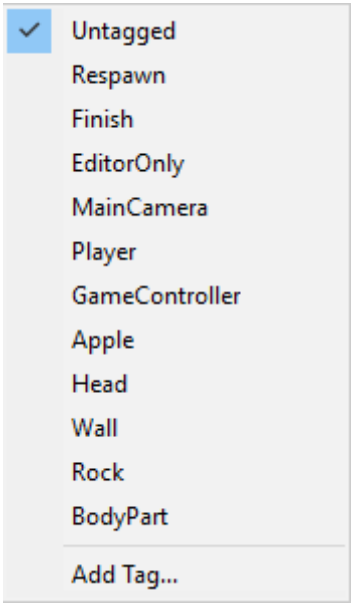
Jak łatwo się domyślić, kliknięcie „Tap to play again” spowoduje, iż gracz zostanie przeniesiony do planszy, gdzie kolejny raz spróbuje swoich sił.

Scena zawiera również obiekt „Cube”, który odpowiedzialny jest za pokazanie graczowi tekstury tła.

Lewy dolny róg (bardzo mała przestrzeń), jest miejscem obiektu, gdzie skierowana jest również kamera i oświetlenie punktowe.

DZIAŁANIE SKRYPTÓW

Po opisie ważnych obiektów w scenie, również ważną częścią którą warto opisać są tagi przedstawione są niżej:



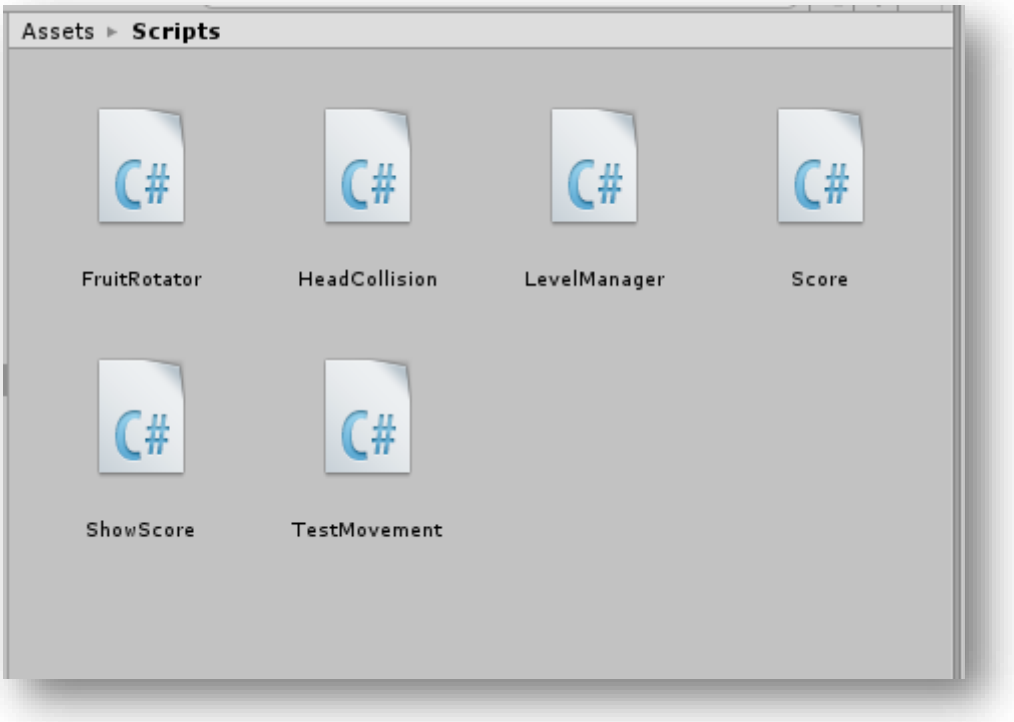
Najprościej tłumacząc, są to etykiety grupujące poszczególne obiekty.

Dzięki przypisaniu takiej etykiety, skrypty mogą w bardzo prosty sposób komunikować się między obiektami.

Teraz, po objaśnieniu podstawowej konfiguracji, opisany zostanie kolejny etap – tzn. część odpowiadająca za działanie.

W folderze ze skryptami można znaleźć wszystkie skrypty, które zostały wykorzystane w grze.

Zdjęcie poniżej prezentuje wykorzystane skrypty.



- FruitRotator

Odpowiedzialny za kręcenie się obiektów do zjedzenia wokół osi Y.

- HeadColision

Odpowiedzialny za wykrywanie kolizji między głową węża, a obiektami z etykietami - wall, rock i bodyPart.

- LevelManager

Odpowiedzialny za przemieszczanie się między scenami.

- Score

Licznik zdobytych punktów podczas gry.

- ShowScore

Pokazuje wszystkie zdobyte punkty w scenie przegranej gry.

- TestMovement

Sterowanie Bobem.

Metody klasy FruitRotator:

Metoda **Start**:

```
16    void Start () {
17        count = 0;
18
19        audioSrc = GetComponent<AudioSource>();
20
21        particles.GetComponent<ParticleSystem>().enableEmission = false;
22
23        newPosition = new Vector3(Random.Range(minDistance, maxDistance), 0.5f, Random.Range(minDistance, maxDistance));
24    }
25
```

Linia 17:

Odpowiada za inicjalizację zmiennej odpowiedzialnej za liczenie punktów.

Linia 19:

Odpowiada za inicjalizację źródła dźwięku

Linia 23:

Odpowiada za generowanie nowej pozycji obiektu, gdzie

```
private int minDistance = -8;
private int maxDistance = 8;
private Vector3 newPosition;
```

Co oznacza, że obiekty generują w sposób losowy w przestrzeni ograniczonej.

Metoda **Update**:

```
29
30    void Update () {
31        transform.Rotate(new Vector3(0, 120, 0) * Time.deltaTime);
32    }
33
```

Odświeżana co klatkę na sekundę.

W tym przypadku rotacja obiektu.

Metoda **OnTriggerEnter**:

```
36      void OnTriggerEnter(Collider other){
37          if (other.gameObject.CompareTag("Head")){
38              snakeObject.GetComponent<TestMovement>().addBodyPart();
39
40              particles.GetComponent<ParticleSystem>().enableEmission = false;
41              StartCoroutine(stopParticles());
42
43              audioSrc.Play();
44
45              StartCoroutine(getNewPosition());
46
47              this.gameObject.SetActive(false);
48              Debug.Log("Triggred");
49
50              this.gameObject.transform.position = newPosition;
51              this.gameObject.SetActive(true);
52              count++;
53          }
54      }
```

Odpowiada za obsługę zdarzenia wynikającego z kolizji z głową Boba.

Linia 40,41:
Uruchamia emisję cząsteczek (efekt zebrania jedzenia).

Linia 43:
Uruchamia źródło dźwięki (dźwięk jedzenia).

Linia 45:
Bierze pozycję spawnu obiektu.

Linia 47:
Ukrywa obiekt po zjedzeniu.

Linia 50:
Ustawia nową pozycję do spawnu obiektu.

Linia 51,52:
Aktywuje widoczność obiektu w nowej pozycji i inkrementuje punkty.

Metoda **getNewPosition:**

```
46
47      IEnumerator getNewPosition(){
48          while(Physics.CheckSphere(newPosition, 1.0f)){
49              newPosition = new Vector3(Random.Range(minDistance, maxDistance), 0.8f, Random.Range(minDistance, maxDistance));
50              yield return null;
51          }
52      }
53
```

Bierze nową pozycję na spawn obiektu.

Metoda **stopParticles**:

```
53
54      IEnumerator stopParticles(){
55          yield return new WaitForSeconds(0.4f);
56          particles.GetComponent<ParticleSystem>().enableEmission = false;
57      }
58  }
```

Zatrzymuje emisję cząsteczek po kolizji z obiektem.

@HEADCOLLISION

```
4  public class HeadCollision : MonoBehaviour {
5
6      private LevelManager levelManager;
7
8      void Start () {
9          levelManager = GameObject.Find("LevelManager").GetComponent<LevelManager>();
10     }
11
12     void OnTriggerEnter(Collider other){
13         if(other.gameObject.CompareTag("Wall") || other.gameObject.CompareTag("Rock") || other.gameObject.CompareTag("BodyPart")){
14             levelManager.LoadLevel("Lose");
15         }
16     }
17
18 }
19
```

Metoda **Start**:

Zawiera referencję do obiektu LevelManager.

```
levelManager = GameObject.Find("LevelManager").GetComponent<LevelManager>();
```

Metoda **OnTriggerEnter**:

Działanie skryptu polega głównie na wykrywaniu kolizji obiektu z obiektami z tagami:

```
(other.gameObject.CompareTag("Wall") || other.gameObject.CompareTag("Rock") || other.gameObject.CompareTag("BodyPart"))
```

Oznacza to, że jeśli obiekt wywoła kolizję między obiektami z etykietami „Wall”, „Rock”, „BodyPart”, to wywoła metodę na obiekcie LevelManager, tzn.:

```
levelManager.LoadLevel("Lose");
```

Załadowanie levelu o nzawie „Lose”.

@SCORE

Skrypt Score – używany w scenie „game”.

```
5 public class Score : MonoBehaviour {
6
7     public static int score = 0;
8     public Text scoreText;
9     void Start () {
10        GameObject.DontDestroyOnLoad(gameObject);
11    }
12
13    void Update () {
14        score = FruitRotator.count;
15        scoreText.text = "Score: " + score;
16    }
17 }
18
19
```

Prosty licznik zdobytych punktów podczas sterowania Bobem, w metodzie odświeżanej co klatkę na sekundę.

@SHOWSCORE

Skrypt pokazujący ilość zdobytych punktów w scenie przegranej gry.

Działanie polega na zwyczajnej zmianie tekstu, wraz z operatorem konkatenacji i dodaniu ilości punktów.

```
5 public class ShowScore : MonoBehaviour {
6
7     public Text scoreText;
8
9     void Start () {
10        scoreText.text = "Score: " + Score.score;
11    }
12
13    void Update () {
14
15    }
16 }
17
```

@LEVELMANAGER

Jest to skrypt odpowiadający za przekierowanie gracza na odpowiednią scenę.

Metoda **LoadLevel**:

Przekierowuje gracza na level podany wcześniej w zakładce „Inspector”.

Pozwala to na łatwą zmianę nazwy sceny, bez konieczności ingerowania programisty w kodzie.

Metoda **QuitLevel**:

Zamka aplikację.

Metoda **LoadNextLevel**:

Przekierowuje gracza na wcześniej ustawioną scenę.

Zmienna globalna z nazwą sceny wygląda następująco:

```
public string levelName;
```

Metoda **LoadingGame**:

Po kliknięciu “Tap to start a game”, ukrywa napis i uaktywnia widok napisu „Loading…”, jednocześnie dając na to czas metodą **LoadGameWithProgress**

```
13      public void LoadLevel (string levelName){
14          Debug.Log("Loaded Level: "+levelName);
15          SceneManager.LoadScene(levelName);
16      }
17
18      public void QuitLevel (){
19          Debug.Log("Quit");
20          Application.Quit();
21      }
22
23      public void LoadNextLevel(){
24          Application.LoadLevel(levelName);
25      }
26
27      public void loadingGame(){
28          startButton.gameObject.SetActive(false);
29
30          loadingText.gameObject.SetActive(true);
31          StartCoroutine(loadGameWithProgress());
32      }
33
34
```

Metoda **LoadGameWithProgress**:

```
35      IEnumerator loadGameWithProgress(){
36          yield return new WaitForSeconds(1);
37
38          ao = SceneManager.LoadSceneAsync(1);
39
40          ao.allowSceneActivation = false;
41
42          while(!ao.isDone){
43              if(ao.progress == 0.9f){
44                  ao.allowSceneActivation = true;
45              }
46
47              Debug.Log(ao.progress);
48              yield return null;
49          }
50      }
51  }
52
```

Odpowiedzialna jest za załadowanie levelu z określonym progress timem, czyli ładuje level przed przełączeniem go.

@TESTMOVEMENT

Skrypt odpowiada za poruszanie się / ustawianie elementów na mapie.
Zmienne globalne dzielą się na te odpowiadające za:

- *Właściwości Snake'a*
- *Rozstawienie elementów na mapie*
- *Inicjalizację ilości generowanych elementów na mapie*
- *Właściwości odpowiadające za movement*
- *Właściwości odpowiadające za rotację*
- *właściwości odpowiadające za dystans obiektów od krawędzi mapy*

Metoda **Start:**

```
45
46      void Start(){
47          audioSource = GetComponent<AudioSource>();
48
49          if(bodyParts.Count != 0)
50              startingRotation = bodyParts[0].rotation;
51
52          for(int i = 0; i < startBodySize - 1; i++){
53              addBodyPart();
54          }
55
```

Odpowiada za rotację początkową i długość początkową węża.

Dodawanie obiektów takich jak kamienie, krzaki wygląda w ten sposób:

```
58      for(int j=0; j<ILOŚĆ_KAMIENI; j++){
59          Vector3 newPosition = new Vector3(Random.Range(boardMinDistance, boardMaxDistance), 0.3f, Random.Range(boardMinDistance, boardMaxDistance));
60          if(OBIEKT_KAMIENIA != null) {
61              GameObject go = Instantiate(OBIEKT_KAMIENIA, newPosition, Quaternion.identity) as GameObject;
62          }
63      }
64
```

```
for(int j=0; j<ILOŚĆ OBIEKTÓW; j++){
Vector3 newPosition = new Vector3(Random.Range(boardMinDistance,
boardMaxDistance), 0.3f, Random.Range(boardMinDistance, boardMaxDist
ance));

if(OBIEKT != null) {
        GameObject go = Instantiate(OBIEKT, newPosition, Quaternion.i
dentity) as GameObject;
        }
}
```

Metoda **RightTouched**:

```
public void RightTouched()
{

    audioSource.PlayOneShot(moveSound, 0.4F);
    bodyParts [0].position = new Vector3 (bodyParts [0].position.x, 0.5f, bodyParts [0].position.z);

    StopAllCoroutines ();
    StartCoroutine (Rotate (angel));
    angel += 90;
    bodyParts [0].Rotate (-Vector3.forward * fTurnRate * Time.deltaTime);

}
```

Odpowiada za obsługę każdego prawego clicku.

Każde dotknięcie powoduje odtworzenie dźwięku i zmianę kątu poruszania się węża.

Dodatkowo ustawia węża na ustalonej wysokości, co zabezpiecza węża przed zmianą położenia.

Metoda **LeftTouched**:

```
public void LeftTouched()
{
    audioSource.PlayOneShot(moveSound, 0.4F);
    bodyParts[0].position = new Vector3(bodyParts[0].position.x, 0.5f, bodyParts[0].position.z);

    StopAllCoroutines();
    StartCoroutine(Rotate(angel-180));
    angel-=90;
    bodyParts[0].Rotate (Vector3.forward * fTurnRate * Time.deltaTime);

}
```

Metoda działa analogicznie jak metoda „RightTouched”, tylko że kąt zmienia się na -90.

Metoda **Move**:

```
130     public void move(){
131
132         float currentSpeed = speed;
133
134         if(Input.GetKey(KeyCode.UpArrow)) currentSpeed *= 2;
135
136         if(bodyParts.Count != 0)
137             bodyParts[0].Translate(bodyParts[0].forward * currentSpeed * Time.smoothDeltaTime, Space.World);
138
139         if(bodyParts.Count != 0)
140             bodyParts[0].position = bodyParts[0].position + bodyParts[0].forward * 2.0f * Time.deltaTime;
141
142         for(int i = 1; i<bodyParts.Count; i++){
143             currentBodyPart = bodyParts[i];
144             prevBodyPart = bodyParts[i-1];
145
146             distance = Vector3.Distance(prevBodyPart.position, currentBodyPart.position);
147
148             Vector3 newPosition = prevBodyPart.position;
149
150             newPosition.y = bodyParts[0].position.y;
151
152             float dTime = Time.deltaTime * (distance/minDistance) * currentSpeed;
153
154             if(dTime > 0.5f) dTime = 0.5f;
155
156             currentBodyPart.position = Vector3.Slerp(currentBodyPart.position, newPosition, dTime);
157             currentBodyPart.rotation = Quaternion.Slerp(currentBodyPart.rotation, prevBodyPart.rotation, dTime);
158         }
159     }
```

Linia 146:

Oblicza dystans między bieżącym, a wcześniejszym obiektem ciała węża.

Linia 148:

Otrzymanie wcześniejszej pozycji węża.

Linia 150:

Blokuje oś Y.

Linia 152-154:

Zapobiega tzw. Frame dropy.

Linia 156:

Ruch między aktualną pozycją, a nową pozycją.

Linia 157:

Ruch obiektu węża z aktualnej pozycji.

Metoda **AddBodyPart**:

```
161     public void addBodyPart(){
162         if(bodyParts.Count != 0){
163             speed += 0.03f;
164             Transform newPart = (Instantiate(bodyPartPrefab, bodyParts[bodyParts.Count - 1].position, bodyParts[bodyParts.Count -1].rotation) as GameObject).transform;
165             newPart.SetParent(transform);
166             bodyParts.Add(newPart);
167         }
168     }
169
```

Jeśli liczba dodanych kul do węża jest różna od 0, to wąż przyspiesza o wartość float równą 0.03f.

```
speed += 0.03f;
```

Oprócz tego tworzy instancje ostatniej kuli w liście.

```
Transform newPart = (Instantiate(bodyPartPrefab, bodyParts[bodyParts
.Count - 1].position, bodyParts[bodyParts.Count -
1].rotation) as GameObject).transform;
```

Następne linie dodają obiekt ostatniej kuli do listy.

Metoda **Rotate**:

```
169
170     IEnumerator Rotate(float rotationAmount){
171         Quaternion finalRotation = Quaternion.Euler( 0, rotationAmount, 0 ) * startingRotation;
172         while(bodyParts[0].rotation != finalRotation){
173             bodyParts[0].rotation = Quaternion.Lerp(bodyParts[0].rotation, finalRotation, Time.deltaTime*rotationSpeed);
174             yield return null;
175         }
176     }
```

Służy do powolnej rotacji węża.

MATERIAŁY

#1 Tutorial

[Unity 5] Tutorial: How to create snake 3D in Unity - introduction

https://www.youtube.com/watch?v=7SB1IQN3MtE

2# Unity Manual

Unity User Manual (2017.3)

https://docs.unity3d.com/Manual/index.html

3# Kevin MacLeod

Muzyka w tle

https://www.youtube.com/user/kmmusic/videos

4# Models

Free 3D Models

https://free3d.com

5# Czcionki

Pixel Font – Tripfive

https://www.dafont.com/5x5-pixel.font

6# Dodatkowe informacje

Forum informatyczne

https://forum.pasja-informatyki.pl/