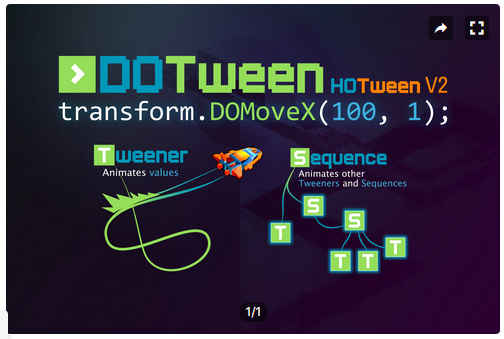
**1. Подготовка проекта**

По ссылку находится Модели (у нас их нет, поэтому реализуем модель самостоятельно через ProBilder).

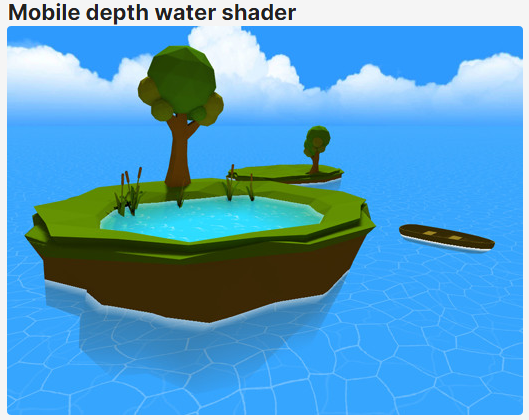
Для данной игры потребуется dotween – можно анимировать объекты, через код. (Шаблоны, Упрощение некоторых задач).

Установка дополнительного модуля dotween стандартная, сначала скачиваем ресурсы, затем импортируем.



Первое окно демонстрирует небольшую справку по поводу модуля, нажимаем Open и появляется второе окно. Во втором окне нажимаем Setup и видим модули, к которым можно присоединять компоненты данного модуля. Нажимаем Apply.

Для качественного отображения игры необходимо скачать Шейдер для Воды mobile deep.



После всех подготовок можно переключиться в мобильную (Android) платформу.

**2. Подготовка сцены**

Для подготовки сцены мы подготавливаем примитивные объекты, используя ProBuilder и получаем следующую картинку или сцену, при этом не забываем настраивать камеру и свет.

|  |  |
| --- | --- |
|  |  |

Сцена включает в себя несколько объектов + пустой игровой объект Tower, вокруг которого будет строиться геймплей игры.

Не забываем создавать функциональные папки. Для объектов рекомендуется создать материал, для их однообразия и приятного отображения.

**3. Спаунер Башни**

Наша цель построить башни и отслеживать следующий момент: «при попадании пули по блоку, вся башня спускается».

Для реализации данного этапа необходимо изначально подготовить типизацию объектов на сцене, для Block(выступающие в роли-платформ, которые нужны сбивать) и остальных элементов.

Реализация построения большого количества блоков в листинге, ниже:

|  |
| --- |
| public class TowerBuilder : MonoBehaviour  {  [SerializeField] private float towerSize;  [SerializeField] private Transform buildPoint;  [SerializeField] private Block block;  //Использование Структуры Листа  private List<Block> blocks;  private void Start() {  Build();  }  /// <summary>  /// Метод для Реализации Строительства  /// При этом инициализируентся переменная, создается начальная точка для ПервогоБлока  /// Далее создается блок, загружается в лист и точка перемещается "выше"  /// </summary>  /// <returns></returns>  public List<Block> Build() {  blocks = new List<Block>();  Transform currentpoint = buildPoint;  for (int i = 0; i < towerSize; i++)  {  Block newBlock = BuildBlock(currentpoint);  blocks.Add(newBlock);  currentpoint = newBlock.transform;  }  return blocks;  }  /// <summary>  /// Непосредстванное создание нового блока в указанной точке (с поправками)  /// </summary>  /// <param name="currentBuildPoint"></param>  /// <returns></returns>  private Block BuildBlock(Transform currentBuildPoint) {  return Instantiate(block, GetBuildPoint(currentBuildPoint), Quaternion.Euler(90,0,0), buildPoint);  }  /// <summary>  /// Определение точки создания нового блока (с поправками)  /// </summary>  /// <param name="currentSegment"></param>  /// <returns></returns>  private Vector3 GetBuildPoint(Transform currentSegment) {  return new Vector3(buildPoint.position.x + 3.5f, currentSegment.position.y+ block.transform.localScale.y / 2, buildPoint.position.z);  //return new Vector3(buildPoint.position.x+3.5f, currentSegment.position.y + currentSegment.localScale.y / 2 + block.transform.localScale.y / 2, buildPoint.position.z);  }  } |

После этого не забываем создавать Префаба и указать в публичные поля объект и точку начала спавна.

**4. Создание Танка**

Для создания танка мы используем классические блоки, представленные движком Unity. Настраиваем, размещаем и сразу же создаем пулю, как Префаба, для повторного создания объекта.

Внимание!   
 При создании пули рекомендуется задать ей координаты точки появления, для этого лучше всего сделать его изначально дочерним к объекту танка, далее задать ей позицию, согласно точке появления, таким образом он будет появляться в верном месте.

Пример изображения, полученного результата моделирования и кодинга:

|  |  |
| --- | --- |
|  |  |
|  |  |

Код программы представляет собой два основных скрипта, не забываем добавлять скрипты к объектам, а к параметрам необходимые аргументы:

|  |
| --- |
| Bullet.cs |
| //для типизации и дальнейшего использования |
| Tank.cs |
| public class Tank : MonoBehaviour{  [SerializeField] private Transform shootPoint; //точка создания  [SerializeField] private Bullet bullettemplate; //шаблон  [SerializeField] private float delayBetweenShoots; //время между выстрелами  private float timeAfterShoot; //счетчик для отслеживания частоты выстрелов  private void Update() {  timeAfterShoot += Time.deltaTime; //увеличения счетчика  if (Input.GetMouseButton(0)) {  if (timeAfterShoot > delayBetweenShoots) {  Shoot();  timeAfterShoot = 0;  } } }  //Создания объектаПули  private void Shoot() {  Instantiate(bullettemplate, shootPoint.position, Quaternion.identity);  }  } |

**5. Создание Пули**

При создании объекта Пули, необходимо добавить ей компонент Rigidbody и включить параметр isKinematic (это позволяет ей лететь строго в заданном направлении, а не по физике (баллистически))

|  |
| --- |
|  |

Напишем код, который двигает наш объект вперед:

|  |
| --- |
| Bullet.cs |
| public class Bullet : MonoBehaviour  {  [SerializeField] private float speed;  private Vector3 moveDirection;  private void Start() {  moveDirection = Vector3.forward;  }  private void Update(){  transform.Translate(moveDirection \* speed \* Time.deltaTime);  }  } |

**6. Обработка столкновения с блоком**

Для обработки столкновений нам необходимо нашему блоку добавить MeshCollider и выставить ей два параметра, такие как Convex/isTrigger.

|  |
| --- |
|  |

Пишем теперь код, для реализации столкновения:

|  |
| --- |
| Bullet.cs |
| public class Bullet : MonoBehaviour{  [SerializeField] private float speed;  private Vector3 moveDirection;  private void Start() {  moveDirection = Vector3.forward;  }  private void Update() {  transform.Translate(moveDirection \* speed \* Time.deltaTime);  }  private void OnTriggerEnter(Collider other) {  if (other.TryGetComponent(out Block block)) {  block.Break();  Destroy(gameObject);  } }} |
| Block.cs |
| public class Block : MonoBehaviour{  public void Break() {  Destroy(gameObject);  }} |

Пробуем!!!

**7. Уменьшение башни и события**

Для этого теперь переписываем определенные куски кода и используем новый способ взаимодействия с движком Unity (события)

|  |
| --- |
| Tower.cs |
| [RequireComponent(typeof(TowerBuilder))]  public class Tower : MonoBehaviour{  private TowerBuilder towerBuilder;  private List<Block> blocks;  void Start() {  towerBuilder = GetComponent<TowerBuilder>();  blocks = towerBuilder.Build();  foreach (var block in blocks) {  block.BulletHit += OnBulletHit; }  }  private void OnBulletHit(Block hitedblock) {  hitedblock.BulletHit -= OnBulletHit;  blocks.Remove(hitedblock);  foreach (var block in blocks) {  block.transform.position = new Vector3(block.transform.position.x,block.transform.position.y - 0.45f,block.transform.position.z);  }  }  } |
| В методе События стоит отметить, что при уничтожении блока, наша башня спускается. |
| TowerBuilder.cs |
| //Убираем метод Start, а именно метод Build |
| Block.cs |
| using UnityEngine.Events;  public class Block : MonoBehaviour{  public event UnityAction<Block> BulletHit;    public void Break() {  BulletHit?.Invoke(this);  Destroy(gameObject);  }  } |

Способ работы с событиями следующий:

|  |
| --- |
| public event UnityAction<Block> BulletHit; |
| Непосредственное Событие |
| BulletHit?.Invoke(this); |
| Событие случилось, при этом событие свершилось именно у определенного элемента |
| Для правильной работы Событий необходимо подписывать и отписываться, для этого используем конструкцию |
| block.BulletHit += OnBulletHit; (подписка)  hitedblock.BulletHit -= OnBulletHit; (отписка) |

Внимание!   
 Возможно при попадании пули уничтожаются сразу же несколько блоков, для исправления либо уменьшаем пули, либо увеличиваем наши блоки.

**8. Создаем препятствия и dotween**

В качестве препятствий мы используем ProBuilder.

Внимание!   
 После завершения создания объекта не забываем сделать следующие действия, а именно нажать на «Open ProBuilder» и Первое – установить CenterPivot и Второе – Export объекта, в формате obj.

Далее так же настраиваем несколько объектов и формируем некий круг. В качестве объекта, который будет крутить наши препятствия будет отдельный объект GameObject, и разместим препятствия дочерними к данному объектную.



Для кручения, мы используем bottween и пишем скрипт:

|  |
| --- |
| ObstacleRotater.cs |
| public class ObstacleRotater : MonoBehaviour{  [SerializeField] private float animatronDuration;  private void Start() {  transform.DORotate(new Vector3(0, 360, 0), animatronDuration, RotateMode.FastBeyond360).SetLoops(-1, LoopType.Yoyo);  }  } |
| Аргументы метода DORotate:  1. Конечное значение  2. Длительность  3. От 0 до 360  3.1. Повторение (-1 = бесконечность)  3.2. Принцип анимированния (Yoyo – вперед и назад) |
| Obstacle.cs |
| public class Obstacle : MonoBehaviour{  } |

Не забываем нашим препятствиям указать компонент Mesh Collider и установить параметры Convex и Is Trigger, для последующей обработки.

**9. Отскакивание от препятствий**

Переписываем небольшой код, для отскока нашей пули:

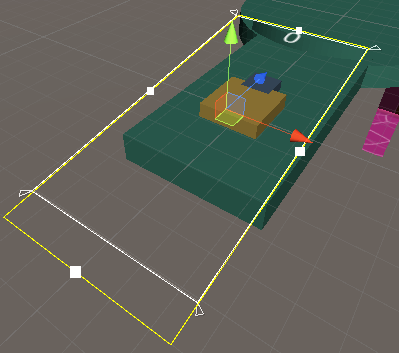
|  |
| --- |
| Bullet.cs |
| public class Bullet: MonoBehaviour{  [SerializeField] private float speed;  [SerializeField] private float bounceForce;  [SerializeField] private float bounceRadius;  private Vector3 moveDirection;  private void Start() {  moveDirection = Vector3.forward;  }  private void Update() {  transform.Translate(moveDirection \* speed \* Time.deltaTime);  }  private void OnTriggerEnter(Collider other) {  if (other.TryGetComponent(out Block block)) {  block.Break();  Destroy(gameObject);  }  if (other.TryGetComponent(out Obstacle obstacle)){  Bounce();  }  Destroy(gameObject, 5f);  }  private void Bounce() {  moveDirection = Vector3.back + Vector3.up;  Rigidbody rigidbody = GetComponent<Rigidbody>();  rigidbody.isKinematic = false;  rigidbody.AddExplosionForce(bounceForce, transform.position + new Vector3(0, -1f, 1), bounceRadius);  }  } |
| Толкание Пули rigidbody.AddExplosionForce(bounceForce, transform.position + new Vector3(0, -1f, 1), bounceRadius); |
| Аргуаменты:  1. Сила  2. Точка Взрыва  3. Дальность/Радиус |

Настраиваем параметры в префабе.

**10. Отображение размера башни**

Место расположения для текста, расположим чуть выше нашего моста. Создаем Canvas/Text (TMP) и настраиваем его как World Space и вставляем камеру.

Далее настраиваем наш Canvas на мост.



Незабываем настраивать текст, к примеру (Необходимо чтоб шифр самостоятельно в заданном диапазоне вставлял цифры):

|  |
| --- |
|  |

Пишем код, для реализации отображения количество башен, для этого снова используем систему событий:

|  |
| --- |
| TowerSizeView.cs |
| public class TowerSizeView : MonoBehaviour{  [SerializeField] private TMP\_Text sizeVies;  [SerializeField] private Tower tower;  private void OnEnable(){  tower.sizeUpdate += OnSizeUpdated;  }  private void OnDisable(){  tower.sizeUpdate -= OnSizeUpdated;  }  private void OnSizeUpdated(int size) {  sizeVies.text = size.ToString();  }  } |
| Tower.cs |
| public class Tower : MonoBehaviour{  ////  public event UnityAction<int> sizeUpdate;  ////  void Start() {  ////  sizeUpdate?.Invoke(blocks.Count);  }  private void OnBulletHit(Block hitedblock) {  /////  sizeUpdate?.Invoke(blocks.Count);  }  } |

Код работает следующим образом:

|  |
| --- |
| Если заметить, то методы называются как OnEnable и OnDisable, которые работают по принципу появления новых объектов. |
| Теперь кода у основного объекта Tower при создании или при уничтожении блока, события отправляется в наш «счетчик», который будет через метод OnSizeUpdate обновлять наш «счетчик» |

Скрипт вешаем на наш текст и заполняем аргументы.

**11. Отдача Танка**

Пишем небольшой код, который при стрельбе мы подвинем наш танк «назад».

|  |
| --- |
| Tank.cs |
| public class Tank : MonoBehaviour  {  [SerializeField] private Transform shootPoint;  [SerializeField] private Bullet bullettemplate;  [SerializeField] private float delayBetweenShoots;  [SerializeField] private float recoilDistance;  private float timeAfterShoot;  private void Update() {  timeAfterShoot += Time.deltaTime;  if (Input.GetMouseButton(0)) {  if (timeAfterShoot > delayBetweenShoots) {  Shoot();  transform.DOMoveZ(transform.position.z - recoilDistance, delayBetweenShoots / 2).SetLoops(2, LoopType.Yoyo);  timeAfterShoot = 0;  }  }  }  private void Shoot() {  Instantiate(bullettemplate, shootPoint.position, Quaternion.identity);  }  } |

Принцип работы, следующий:

|  |
| --- |
| Метод, реализующий отдачу |
| transform.DOMoveZ(transform.position.z - recoilDistance, delayBetweenShoots / 2).SetLoops(2, LoopType.Yoyo); |
| Аргументы:  1. Дистанция отдачи  2. Время отдачи, при этом танк должен успеть вернуться к изначальному состоянию  3. SetLoops (2-действия по принципу «вперед-назад», по принципу Yoyo). |

**12. Создание эффекта исчезновения**

Создание эффекта по следующим шагам:

1. Создаем объект Particle System (при этом они 2-хмерые)
2. Переходим в параметр Renderer, который отвечает за отображение. Так как нужен 3-хмерный, то выставляет RnderMode/Mesh и в качестве Mesh выставляем объект нашего Блока ( получиться, если создать объект в ProBuilder, как obj), далее указываем ей Material того же объекта.
3. Emission(излучение-сколько частиц и время), выставляем следующее RateOverTime = 0(количество), а RateOverDistance = 0, Bursts(взрыв), как count =1.
4. Для того, чтобы частица работала корректно (не смотрела на нас), необходимо выставить RenderAlegment = Local.
5. Возможно объект может быть отображен под неверным углом, поэтому параметров StartRotation выставить необходимое значение.
6. Для того, чтобы наш объект вылетал с указанной точки и летел только вверх выключаем параметр Shape, который отвечает за форму вылета наших частиц.
7. Располагаем
8. Реализуем исчезновение/прозрачность нашего объекта, при этом постепенно. Для этого есть параметр ColoroverLifeTime и выставляем его по примерно следующем примеру:

|  |
| --- |
|  |

1. Основой параметр StartLifeTime = 1 указываем время жизни нашей частицы, а Duration = 1 на длительность эффекта. StartSpeed = 3.
2. Для реализации изменения прозрачности нам нужен материал, который умеет менять прозрачность(Alpha канал). Создаем материал и указываем ей Particles/StandartSurface, выставляем параметры Metallic и Smoothness по усмотрению.   
   Главное поменять RenderindMode на Additive, который позволяет управлять прозрачностью.   
   Сменяем материал на нашу частицу
3. Включаем параметр SizeOverLifeTime и меняем начальную и конечную точку размера нашей частицы, меняем он как Transform.Scale.
4. Для того, чтобы наша частица работала не постоянно, а единожды выставляем ей параметр StopAction = Destroy, Looping = false.
5. Даем название частицы и задаем ей префаб.

Напишем небольшой код:

|  |
| --- |
| Block.cs |
| [RequireComponent(typeof(MeshRenderer))]  public class Block : MonoBehaviour{  [SerializeField] private ParticleSystem desrtoyEffects;  private MeshRenderer meshRenderer;  private void Awake() {  meshRenderer = GetComponent<MeshRenderer>();  }  public event UnityAction<Block> BulletHit;  public void SetColor(Color color) {  meshRenderer.material.color = color;    }  public void Break() {  BulletHit?.Invoke(this);  ParticleSystemRenderer renderer = Instantiate(desrtoyEffects, transform.position, desrtoyEffects.transform.rotation).GetComponent<ParticleSystemRenderer>();  renderer.material.color = meshRenderer.material.color;  Destroy(gameObject);  }  } |
| Так как в нашей игре, у каждого объекта свой цвет, то нам необходимы такие элементы как MeshRenderer и строчка кода, которая прямо нашей частицы задает цвет, через параметр renderer. |
| Непосредственное установка кода будет далее… |

Не забываем в скрипты добавлять Аргументы.

**13. Создаем возможность изменения цвета**

|  |
| --- |
| Block.cs |
| [RequireComponent(typeof(MeshRenderer))]  public class Block : MonoBehaviour{  [SerializeField] private ParticleSystem desrtoyEffects;  private MeshRenderer meshRenderer;  private void Awake() {  meshRenderer = GetComponent<MeshRenderer>();  }  public event UnityAction<Block> BulletHit;  public void SetColor(Color color) {  meshRenderer.material.color = color;    }  public void Break() {  BulletHit?.Invoke(this);  ParticleSystemRenderer renderer = Instantiate(desrtoyEffects, transform.position, desrtoyEffects.transform.rotation).GetComponent<ParticleSystemRenderer>();  renderer.material.color = meshRenderer.material.color;  Destroy(gameObject);  }  } |
| Создаем параметр MeshRenderer и инициализируем его перед запуском (Awake). |
| TowerBuilder.cs |
| public class TowerBuilder : MonoBehaviour{  ////  [SerializeField] private Color[] colors;  public List<Block> Build() {  ////  for (int i = 0; i < towerSize; i++)  {  Block newBlock = BuildBlock(currentpoint);  newBlock.SetColor(colors[Random.Range(0, colors.Length)]);  blocks.Add(newBlock);  currentpoint = newBlock.transform;  }  ///  } |
| При создании объекта ей рандомно из массива выдается цвет.  Не забываем этот цвет указать в параметрах и выставить ей Альфа канал… |

**14. Украшаем**

На сайте colors выбираем цветовую палитру и выставляем нашим объектам необходимые цвета, выставляем в Obstacle наших материалов.

Для некоторых объектов необходим Albedo. Для некоторых материалов создаем новые материалы, а не стандартные.

Для создания красивого фона – воды, необходимо сделать следующее:

1. Открыть сцену с папки MobileDepthWater/Scenes, оттуда находим нашу воду(WaterPlane) и материал воды сохраняем как отдельный Material, для этого делаем следующее:

|  |
| --- |
|  |

1. Переходим в папку MobileDepthWater\Materials\Water. И с данным объектом переходим на нашу сцену.
2. Создаем объект Plane и накидываем ей наш материал.
3. Далее настраиваем по следующим параметрам:

|  |
| --- |
|  |

1. Настраиваем на наши «ножки», как будто они в воде:

|  |
| --- |
|  |

Пример реализации воды:

|  |
| --- |
|  |

Настройка цвета Window/Rendering/LightingSettings в качестве примера:

|  |
| --- |
|  |