**ВОПРОСЫ К КОНТРОЛЬНОЙ РАБОТЕ «ПРОГРАММИРОВАНИЕ В UNITY»**

1. Программирование динамического изменения координат и масштаба объекта по осям координат, назначение методов Start() и Update().

Функция

**Start() {…}**

определяет код, который выполняется при запуске приложения (выполняется только один раз для инициализации переменных)

Функция

**Update() {…}**

определяет код, перерисовывающий экран 50 раз в секунду (по умолчанию), т. е. создает программную анимацию на экране, если в этой функции задан шаг изменения какого-либо параметра объекта на сцене.

Запись обращения к свойству позиции ***position*** объекта в пространстве:

**gameObject.GetComponent<Transform>().position**

Для сокращения используется свойство **transform** для игрового объекта:

**transform.position**

Для хранения координат объекта в 3D-пространстве используется тип данных **Vector3**, имеющий 3 параметра, соответствующих значению координат по осям **X, Y, Z**

Например, определив переменную **pos**  типа **Vector3**

**Vector3 pos = transform.position;**

получаем доступ к координатам объекта по осям:

**pos.x, pos.y, pos.z**

соответственно **Х, Y** и **Z**

Для программного масштабирования объекта используется свойство **localScale**

Например, для непрерывного увеличения размеров объекта по оси X можно использовать следующий код:

**void Update ()**

**{**

**transform.localScale += new Vector3(0.1f, 0.0f, 0.0f);**

**}**

1. Программирование динамического вращения объекта с помощью функций Rotate, углов Эйлера и кватернионов, особенности использования кватернионов.

Для поворота в 3D-пространстве используются ***углы Эйлера*** в свойстве **eulerAngles** для компоненты **transform**:

**float angl;**

**void Update () {**

**angl+=5.0f;**

**transform.eulerAngles=new Vector3(angl,0,0);**

**}**

Более просто для вращения можно использовать функцию **Rotate()**,котораяавтоматизирует процесс приращения угла поворота, заданного значением угла для соответственно осей **X, Y, Z**:

**void Update () { transform.Rotate(3, 0, 0);}**

*Для измерения угла поворота в* ***Unity*** *используются* ***градусы***

В поворотах с помощью функции **Rotate** или свойства **eulerAngles** необходимо задавать в общем случае ***три угла поворота*** вокруг осей **X, Y** и **Z**, при этом обратное вращение для них будет некоммутативной (неперестановочной) операцией.

При задании вращения с помощью **кватерниона** **(Quaternion)** задается ***один угол поворота*** и ***ось вращения*** в трехмерном пространстве, что позволяет полностью восстановить первоначальное положение объекта после его поворота.

Кроме того, помощью кватернионов можно осуществить более гладкую интерполяцию между двумя разными положениями объекта, чем это делают углы **Эйлера** и функция **Rotate**.

**Кватернион** – это гиперкомплексное число, представляемое четверкой чисел **q=(w,x,y,z)** или в виде **q=[w,v]**, где: **w** – число, а **v=(x,y,z)** – вектор в трехмерном пространстве.

**Quaternion orig;**

**float angl;**

**void Start()**

**{ orig = transform.rotation; }**

**void Update()**

**{ angl += 3.0f;**

**Quaternion rotY = Quaternion.AngleAxis(angl, Vector3.up);**

**transform.rotation = orig \* rotY;**

**}**

1. Управление движением объекта клавишами клавиатуры, назначение конструкции Time.deltaTime.

**void Update(){**

**if (Input.GetKey(KeyCode.D))**

**{transform.position += new Vector3(0.1f, 0.0f, 0.0f); }  
if (Input.GetKey(KeyCode.W))  
 {transform.position += new Vector3(0.0f, 0.1f, 0.0f); }  
if (Input.GetKey(KeyCode.E))**

**{transform.position += new Vector3(0.0f, 0.0f, 0.1f); }  
if (Input.GetKey(KeyCode.A))**

**{transform.position -= new Vector3(0.1f, 0.0f, 0.0f); }  
if (Input.GetKey(KeyCode.S))**

**{transform.position -= new Vector3(0.0f, 0.1f, 0.0f); }  
if (Input.GetKey(KeyCode.Q))**

**{transform.position -= new Vector3(0.0f, 0.0f, 0.1f); }**

**}**

Класс **Time** обладает рядом свойств и методов, позволяющих регулировать время. Его свойство **deltaTime** задает величину изменения во времени — это время между кадрами на любом компьютере по его таймеру. Например, при заданной частоте 50 кадров в секунду значение переменной **deltaTime** составляет 1/50 секунды.

Умножение скорости передвижения на эту переменную приведет к масштабированию и уравниванию скорости движения на различных компьютерах. Например:

**transform.position += transform.forward\*Time.deltaTime** **;**

Умножение на переменную **Time.deltaTime** уменьшает скорость воспроизведения, поэтому для ее регулировки необходимо домножить ее на нужное число. Например:

**transform.position += transform.forward\*Time.deltaTime\*30;**

1. Использование конструкции Input.GetAxis для управления объектом с помощью клавиатуры и движений курсора мыши с использованием ключевых слов.

Конструкция **Input.GetAxis** возвращает значение перемещения объекта по горизонтали и вертикали клавишами клавиатуры и вращения объекта движениями курсора мыши по экрану, связанные соответственно с именными переменными: **Horizontal, Vertical** и **Mouse X, Mouse Y,** настроить значения которых можно в окне, открывающемся командой меню:

**Edit/Project Settings/Input**

**void Update()  
{  
float dX = Input.GetAxis ("Horizontal");** //клавиши: A, D (стрелки: <, >)  
**float dZ = Input.GetAxis ("Vertical");** //клавиши: W, S (стрелки: ^, вниз)  **transform.Translate (dX, 0, dZ);** //как функция **Rotate()** для перемещ.

**float dXm = Input.GetAxis ("Mouse Y");** //движение курсора по вертик. **float dYm = Input.GetAxis ("Mouse X");** //движение курсора по гориз. **transform.Rotate (dXm, dYm, 0);**

**}**

1. Программная генерация объектов из примитивов Unity со свойствами твердого тела в заданной позиции.

**public class Script\_cubic : MonoBehaviour {**

//объявление переменной типа **MeshRenderer** для каркасной сетки плоскости

**MeshRenderer rend=gameObject.GetComponent<MeshRenderer>();**

//объявление переменных для определения границ плоскости по осям **X-Z**

**public float minX; public float maxX;**

**public float minZ; public float maxZ;**

//объявление переменных для координат **X-Y-Z** генерируемого объекта на сцене

**public float nX; public float nY; public float nZ;**

**void Start () {**

//задание значений для границ плоскости с использование конструкций

//**bounds.min.x** и т. д.**,** где **bounds –** это свойство границы объекта на 3D-сцене

**minX=rend.bounds.min.x; maxX=rend.bounds.max.x;**

**minZ=rend.bounds.min.z; maxZ=rend.bounds.max.z;**

//задание значения координаты генерируемого объекта по высоте – оси **Y**

**nY=gameObject.transform.position.y+5; }**

**void Update () {**

**nX=Random.Range(minX,maxX);**

**nZ=Random.Range(minZ,maxZ);**

**if(Input.GetKeyDown(KeyCode.Q))**

//генерация объекта **cub** из примитива типа **Cube**

**{GameObject cub=GameObject.CreatePrimitive(PrimitiveType.Cube);**

**cub.transform.position=new Vector3(nX,nY,nZ);**

//добавление к объекту **cub** компоненты **Rigidbody** с гравитацией

**cub.AddComponent<Rigidbody>();**

**}**

//поворот плоскости для «сбрасывания» оставшихся на ней кубиков

**if(Input.GetKeyDown(KeyCode.W))**

**{Quaternion rotZ = Quaternion.AngleAxis(-1, new Vector3(0, 0, 1));**

**GameObject.transform.rotation \*= rotZ; }}**

**}**

1. Программная генерация объектов из префаба со свойствами прыгучести и скриптом на удаление, структура функции Instantiate().

**Prefab** – это контейнер (шаблон) для создания клонированных, т. е. идентичных по свойствам и содержанию объектов.

Для создания контейнера **Prefab** необходимо «перетащить» мышью объект из окна **Hierarchy** в область **Assets** в нижней части окна среды. При этом в шаблоне сохраняются все присвоенные исходному объекту компоненты и установленные для них свойства, включая поведение объекта на основе добавленного ему программного кода в компоненте **Script**.

Для создания клонированных объектов на сцене в редакторе среды **Unity** необходимо перетянуть из области **Assets** созданный **Prefab** на сцену и затем повторить операцию столько раз, сколько потребуется клонированных объектов.

Определение границ плоскости:

**public float minX;**

**public float maxX;**

**public float minZ;**

**public float maxZ;**

**public float nX;**

**public float nY;**

**public float nZ;**

**void Start () {**

**MeshRenderer.rend=gameObject GetComponent<MeshRenderer>();**

**minX=rend.bounds.min.x;**

**maxX=rend.bounds.max.x;**

**minZ=rend.bounds.min.z;**

**maxZ=rend.bounds.max.z;**

**}**

Генерация кубиков:

**void Update () {**

**nX=Random.Range(minX,maxX);**

**nZ=Random.Range(minZ,maxZ);**

**if(Input.GetKeyDown(KeyCode.Space))**

**{GameObject cub=GameObject.CreatePrimitive(PrimitiveType.Cube);**

**cub.transform.position=new Vector3(nX,5,nZ);**

**cub AddComponent<Rigidbody>();**

**}**

**}**

* Свойство **Rigidbody** для упругости и гравитации
* Физический материал прыгучесть **Bouncy** для компоненты **Sphere Collider**
* Скрипт для удаления экземпляра через 5 секунд после появления:

**float lifeTime = 5f;  
    void Update()  
    {  
        if (lifeTime > 0)  
        {  
            lifeTime -= Time.deltaTime;  
            if (lifeTime <= 0)  Destroy(this.gameObject);      *//удалить экземпляр*  
        }  
    }**

Генерация объекта:

Метод **Instantiate(a,b,c)** имеет 3 аргумента:

**a** – имя переменной для связи с шаблоном **Prefabs** объекта;

**b** –позиция объекта на сцене;

**c** – поворот объекта относительно осей координат на сцене.

Генерация объектов методом **Instantiate():**

**public GameObject prefub1;** //публичная переменная **prefub1  
 void Update()  
 {  
   if (Input.GetKeyDown(KeyCode.Space))  
    {  
 Vector3 position = new Vector3(0,0,0);** // позиция в центре сцены **Instantiate(prefub1, position, Quaternion.identity);  
    }  
 }**

1. Программная обработка щелчков мыши по экрану с использованием функции OnPointerClick() для изменения цвета объекта, дополнительная настройка сцены и скрипта для обработки щелчков мыши.
2. Для обработки события «щелчок мышью по объекту» необходимо добавить на сцену новый объект **Create/UI/EventSystem**
3. В сценарий для объекта необходимо добавить модуль

**using UnityEngine.EventSystems**

3. К базовому классу **MonoBehaviour** добавить через запятую новый класс для интерфейса системы событий **IPointerClickHandler**

**using UnityEngine;**

**using System.Collections;**

**using UnityEngine.EventSystems;**

**public class Script11 : MonoBehaviour, IPointerClickHandler {**

**…**

**}**

Функция **OnPointerClick()** имеет аргументы, которые указывают на обработку события щелчка мыши при выборе курсором мыши объекта на экране в заданной точке.

Использование функции **OnPointerClick()** для изменения цвета объекта:

**public void OnPointerClick(PointerEventData eventData)**

**{**

**float red = Random.Range(.0f, 1.0f);**

**float green = Random.Range(.0f, 1.0f);**

**float blue = Random.Range(.0f, 1.0f);**

**Color col1 = new Color(red, green, blue);**

**GetComponent<Renderer>().material.color = col1;**

**}**

1. Программная реализация метода «бросания» луча Raycasting и обработка его попадания в объект.

**private Camera cam1;**//инициализация переменной cam1  
**void Start() {  cam1 = GetComponent<Camera>();  } void Update() {    if (Input.GetMouseButtonDown(0)) {**//нажатие ЛКМ

//вектор точки в центре экрана

**Vector3 point = new Vector3(cam1.pixelWidth/2,cam1.pixelHeight/2,0);**//создание луча, испущенного из центра экрана

**Ray ray = \_camera.ScreenPointToRay(point);**//переменная, хранящая информацию об испущенном луче **RaycastHit hit;**

**if (Physics.Raycast(ray, out hit)) {**

//обработка действий для испущенного луча  
**GameObject sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere);** //создание сферы//определение точки курсора мыши на экране при нажатии на ЛКМ

**Vector3 pos = hit.point;**

//позиционирование созданной сферы на экране

**sphere.transform.position = pos; }  }  }**

1. Программная обработка источников света с помощью триггеров на основе их функций, особенности выполнения и использования функции OnTriggerStay().

Триггеры – это объекты, которым назначается свойство *Is Trigger* в инспекторе, коллайдеры которых не взаимодействуют с физикой, не имеют твердой оболочки и которые в итоге делают невидимыми на сцене, выключив режим *Mesh Renderer*.

Триггеры используются для управлением другими объектами при попадании движущегося объекта в область их коллайдера.

Скрипт можно добавлять на один из объектов – триггер или движущийся объект.

**Триггеры обрабатываются тремя функциями:**

**1) void OnTriggerEnter(Collider col)**

**{ if(col.name=="Cube “) {…обработка входа} }**

**2) void OnTriggerExit(Collider col)**

**{if (col.name == "Cube “) {…обработка выхода} }**

**3) void OnTriggerStay(Collider col)**

**{ if (col.name == "Cube “) {…обработка нахождения} }**

**Скрипт** нужно добавить на триггер.

Переменную обрабатываемого триггером объекта нужно объявить публичной **public** и связать в окне **Inspector** с объектом на сцене.

Например, для обработки источника света:

**public Light light1;**

**private void OnTriggerEnter(Collider col)** //при входе в триггер

**{ if(col.gameObject.name==«player») light1.intensity=100; }**

**private void OnTriggerExit(Collider col)** //при выходе из триггера

**{ if(col.gameObject.name==«player») light1.intensity=0; }**

**}**

***light1*** *– переменная типа* ***Light****, которую нужно связать с источником света.*

***col*** *- столкнувшийся с триггером коллайдер объекта.*

***player*** *– имя попавшего в триггер объекта*

**Задание цвета источнику света L1 при входе в триггер объекта player:**

**public Light L1;**

**void OnTriggerEnter(Collider col) {**

**if (col. gameObject.name == “player")**

**{ L1.color = Color.red;} }**

Функция OnTriggerStay() выполняется аналогично функции Update() в каждом кадре, благодаря чему ее можно использовать для задания динамического изменения свойств любых объектов на сцене при условии нахождении исходного объекта в коллайдере триггера.

Вращение объекта cube1 при нахождении в триггере объекта player

**public GameObject cube1;**

**void OnTriggerStay(Collider col) {**

**if (col.gameObject.name == “player")**

**{ cube1.transform.Rotate(1,0,0);} }**

1. Программная обработка столкновения объектов на основе функции OnCollisionEnter() с изменением цвета объекта или наложением на него текстуры.

Для обработки события столкновения объектов (точнее коллайдеров объектов) используется функция **OnCollisionEnter()**, которая требует обладания для сталкивающихся объектов свойством **Rigidbody**, т. е. упругостью твердого тела.

Пример кода, когда при столкновении движущегося объекта со скриптом (игрового объекта **gameObject)** с объектом с именем **Cube1** он меняет его цвет на красный:

**void OnCollisionEnter(Collision col)**

**{if (col.gameObject.name == "Cube1")**

**col.gameObject.GetComponent<Renderer>().material.color = new Color(1, 0, 0);}**

**}**

Для текстур, программно накладываемых на объект, в папке **Assets** проекта необходимо создать дополнительную папку **Resource,** в которой разместить используемые файлы текстур. Пример кода наложения текстуры в файле **pict1** на объект **Cube1** при нажатии на клавишу клавиатуры **Q**:

**void Update() {  
if (Input.GetKeyDown(KeyCode.Q))  
 { Cube1.GetComponent<Renderer>().material.mainTexture**

**= Resources.Load(“pict1") as Texture;**

**}** В качестве альтернативы можно просто объявить публичную переменную типа ***Texture***:

**public Texture texture1;**

связать в компоненте **Script** ее с файлом текстуры **pict1** в папке **Assets** и заменить последнюю строку в предыдущем коде на:

**= texture1;**

1. Программное управление движением танка, вращением его башни и вращением ствола с ограничениями по углу поворота.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class move\_tank : MonoBehaviour

{

Transform bashnya; //- объектная переменная для управления башней

Transform stvol; //- объектная переменная для управления стволом

float Speed = 0.5f; // - для регулирования скорости движения танка

float RotateSpeed = 1f; // - для регулирования скорости вращения башни

float min = 1f; // - минимальный угол поворота ствола

float max = 1000f; // - максимальный угол поворота ствола

public GameObject cam;

void Start()

{

bashnya = gameObject.transform.Find("bashnya");

stvol = bashnya.transform.Find("stvol");

}

// Update is called once per frame

void Update()

{

float z = Input.GetAxis("Vertical"); //Движение вперед-назад

if (z != 0)

{

transform.position += transform.TransformDirection(Vector3.right \* Speed \* z);//сдвигаемся по оси вдоль взгляда камеры на расстояние Speed\*z

}

float x = Input.GetAxis("Horizontal"); //Вращение лево-право

if (x != 0)

{

transform.Rotate(0f, x, 0f);// поворачиваем на угол x вокруг оси Y

}

float h = Input.GetAxis("Mouse X"); //Вращение башни влево-вправо

if (h != 0)

{

bashnya.Rotate(0f, h \* RotateSpeed, 0f);

}

float v = Input.GetAxis("Mouse Y"); //Опускание-поднятие ствола

if (v != 0)

{

float stvol\_angle = Mathf.Clamp(stvol.transform.localEulerAngles.z, min, max);

stvol.localEulerAngles = new Vector3(0f, 0f, stvol\_angle);

stvol.transform.Rotate(0f, 0f, v \* RotateSpeed);

} }}

1. Программная реализация выстрела танка с использованием префаба снаряда.
2. добавьте на сцену новый объект – снаряд с именем, например, **core**.
3. Добавьте к снаряду **core** «физику» командой **Add Component/Physics/Rigidbody**. поставьте галочку **is Kinematic**, чтобы *снаряд мог воздействовать на объекты на сцене, а на него воздействие могло оказываться только из кода*.
4. Создать **prefab** снаряда, действуя аналогично

Добавить для **ствола** танка новый скрипт, в котором, прежде всего, объявить публичную объектную переменную типа **GameObject** для ссылки на префаб cнаряда.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class script\_stvol : MonoBehaviour

{

public GameObject core;

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

if (Input.GetKeyDown(KeyCode.B))

{

Vector3 stvolll = transform.position + transform.TransformDirection(Vector3.right \* 6f);

Instantiate(core, stvolll, transform.rotation);

}

}

}

1. Скрипт префаба снаряда для обработки полета и попадания снаряда в цель, использование тегов для объектов-целей.
2. Для дополнительной обработки (кроме простого «расталкивания») столкновений снаряда с объектами на сценес помощью функции **OnCollisionEnter()**, задать для всех объектов-целей один и тот же маркер-тег, например, **goal**, для чего в **Инспекторе** под именем каждого объекта-цели в строке **Tag** вести выбранное имя тега. *При этом все объекты будут обнаруживаться по имени тега, а не по их именам name, которые у всех объектов обязаны быть разными. Это позволит все такие объекты обрабатывать одним и тем же программным кодом.*

скрипт на **префаб** снаряда в **Assets** для управления поведением снаряда:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class core : MonoBehaviour

{

float coreSpeed = 0.9f;

// Start is called before the first frame update

void Start()

{

Destroy(gameObject, 5f);

}

// Update is called once per frame

void Update()

{

transform.position += transform.TransformDirection(Vector3.right \* coreSpeed);

}

private void OnCollisionEnter(Collision col)

{

if (col.gameObject.tag == "goal")

{

col.gameObject.GetComponent<Renderer>().enabled = false;//скрытие снаряда после попадания в цель

}

}

}

1. Программная реализация вызова анимации взрыва при попадании снаряда в цель.
2. Добавьте в код скрипта префаба снаряда публичную объектную переменную, например, **explosion** для обращения к имеющейся в **Assets** проекта в разделе **Standard Assets/ParticleSystems/Prefabs** различным эффектам, например, эффекту взрыва **Explosion**. После задания имени для публичной объектной переменной свяжите ее в **Инспекторе** в скрипте префаба с этим объектом в списке **Assets**, аналогично тому, как это делалось в п. 4 инструкции.

Добавьте в код скрипта префаба в функцию **OnCollisionEnter()** в обработку условия попадания снаряда в цель генерацию эффекта взрыва **explosion** в точке столкновения с использованием метода **Instantiate()**.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class core : MonoBehaviour

{

float coreSpeed = 0.9f;

public GameObject explosion;

// Start is called before the first frame update

void Start()

{

Destroy(gameObject, 5f);

}

// Update is called once per frame

void Update()

{

transform.position += transform.TransformDirection(Vector3.right \* coreSpeed);

}

private void OnCollisionEnter(Collision col)

{

if (col.gameObject.tag == "goal")

{

col.gameObject.GetComponent<Renderer>().enabled = false;//скрытие снаряда после попадания в цель

Instantiate(explosion, gameObject.transform);

}

}

}

1. Программное озвучивание событий движений танка, выстрела и попадания снаряда в цель, подключение к объектам и настройка звуковых файлов.

Для добавления звуковых эффектов на сцену нужно, прежде всего, найти и перенести в папку **Assets** звуковые файлы

В инспекторе на все объекты-источники звука: танк, ствол, цели и камеру добавить компонент **Audio/AudioSourse**, в котором связать с объектной переменной **AudioClip** соответствующий звуковой файл

* на всех компонентах объектов-источников звука отключить галочку **Play On Awake,** чтобы звук не проигрывался сразу при загрузке сцены, кроме компоненты **AudioSourse** укамеры для проигрывания фонового звука;
* поставить галочку **Loop** в компонентах **AudioSourse** танка и камеры, чтобы звук проигрывался циклически, т. к. танк может долго непрерывно перемещаться по сцене;
* установить для каждого звука соответствующую громкость в параметре **Volume**;
* на всех компонентах объектов-источников звука установить режим **3D-**звука для всех звуков, кроме фонового в параметре **Spatial Blend**;
* изменить при необходимости минимальную и максимальную дистанцию для прослушивания каждого из звуков в режиме 3D Sound Settings.

Для проигрывания фонового звука не требуется создание программного кода – он будет проигрываться автоматически сразу после запуске сцены

1. Для проигрывания звука движения танка необходимо в его скрипт внести следующие добавления:

* объявить объектную переменную для источника - **AudioSource zvtank;**
* объявить булевскую переменную, которая хранит состояние запущен ли звук танка (изначально не запущен) - **bool isPlaying = false;**
* в метод **Start ()** добавить инициализацию переменной **zvtank**

**zvtank = GetComponent<AudioSource>();**

* в метод **Update ()** добавить проверку через условный оператор **if** - если танк двигается и звук не включён, то включить проигрывание источника звука на танке и отметить через булевскую переменную, что звук включен:

**if ((Input.GetAxis("Horizontal") != 0 || Input.GetAxis("Vertical") != 0) && !isPlaying)**

**{            zvtank.Play();             isPlaying = true;         }**

* добавить также в метод **Update ()** условие на проверку остановлен ли танк и играет ли звук, при выполнении которого выключитьпроигрывание звука на танкеи отметить, что звук выключен:

**if (Input.GetAxis("Horizontal") == 0 && Input.GetAxis("Vertical") == 0 && isPlaying)**

**{            zvtank.Stop();            isPlaying = false;        }**

1. Для запуска звука выстрела необходимо добавить в скрипт ствола в условие проверки нажатия клавиши клавиатуры для запуска снаряда строку:

**gameObject.GetComponent<AudioSource>().PlayOneShot(gameObject.GetComponent<AudioSource>().clip);**

1. Для запуска звука разрыва снаряда при попадании его в цель необходимо добавить в скрипт префаба снаряда в методе **OnCollisionEnter(Collision col)** в условие проверки столкновения снаряда с целью аналогичную строку, но с учетом объекта столкновения:

**col.gameObject.GetComponent<AudioSource>().PlayOneShot(col.gameObject.GetComponent<AudioSource>().clip);**

1. Создание панели управления скоростью танка на основе функции OnGUI().

**void OnGUI()**

**{***//создание прямоугольника-холста для размещения группы элементов панели* **GUI.BeginGroup(new Rect(10, h, 250, 300));** *//****h*** *– расстояние от верха сцены**//прорисовка холсте панели в виде прямоугольника с заголовком* **GUI.Box(new Rect(10, 0, 200, 200), "УПРАВЛЕНИЕ СКОРОСТЬЮ");***//вывод в прямоугольник текста надписи и скорости танка***GUI.Label(new Rect(15,30, 200,30),"Скорость танка  " + speed + "  ");***//задание скорости по слайдеру в его прямоугольнике*

**speed = GUI.HorizontalSlider(new Rect(15,50,170,30),speed,0.0f,10.0f);***//условие проверки нажатия на создаваемую кнопку для запуска функции* ***Hide()***

**if (GUI.Button(new Rect(10, 170, 90, 20), "Скрыть ПУ")) {  Hide(); }***//условие проверки нажатия на создаваемую кнопку для запуска функции* ***Show()***

**if (GUI.Button(new Rect(100, 170, 90, 20), "Показать ПУ")) {  Show(); }***//закрытие группы элементов панели*

**GUI.EndGroup();**   
**}**  
*//Функции скрытия-открытия панели*

**public void Hide()   {  h = -170;}  
public void Show()   {  h = 0;   }**

1. Программирование поворота башни танка-бота в сторону танка-игрока при попадании танка-бота в триггерный коллайдер танка-игрока, структура методов LookRotation() и Slerp().
2. Создать предварительно в папке **Assets** новый скрипт с именем, например, **«Bot»** для использования в дальнейшем танком-ботом со следующим программным кодом:
   1. объявить переменные типа **float** переменные для задания движений танка-бота – скорость передвижения, скорость поворота и скорость поворота башни;
   2. публичные объектные переменные типа **Transform** для управления его башнейи стволом и переменную типа **GameObject** для ссылки на префаб снаряда;
   3. булеву переменную **bool** для определения, может ли в данный момент танк произвести выстрелс начальным значением **true**.
3. Добавить в скрипт бота метод **OnTriggerStay(Collider other)**, т.к. выполнять какие-либо действия он должен при нахождении в большом триггере игрока. Внутри этого метода определить условие, что танк-бот находится именно в триггере игрока:

**if (other.tag=="Player")**

1. При выполнении условия, что танк-бот находится в триггере игрока, необходимо вычислить:
   1. дистанцию от бота до игрока, используя метод **Distance()**:

**float distance = Vector3.Distance(other.transform.position, transform.position);**

* 1. определить 3D-вектор направления от бота к игроку:

**Vector3 relativePos = (other.transform.position - transform.position);**

* 1. определить по вектору направления кватернион 3D-поворота башни и ствола (здесь второй кватернион доворачивает вокруг вертикальной оси на угол в 90 градусов для согласования осей в **Unity** и **3ds MAX**):

**Quaternion newRot= Quaternion.LookRotation(relativePos)\*Quaternion.AngleAxis(-90,Vector3.up );**

1. При попадании бота в сферический коллайдер танка-игрока медленно повернуть башню бота в направлении танка-игрока с заданной функцией **Slerp()** скоростью (здесь: **bash** – переменная, связанная с башней бота, **rotspeedbsh** – скорость поворота башни):

**bash.rotation = Quaternion.Slerp(bash.rotation, newrot, Time.deltaTime \*rotspeedbash);**

1. Программирование поворота и движения танка-бота в сторону танка-игрока при достижении определенной дистанции внутри триггерного коллайдера танка-игрока, структура функции Lerp().
2. При дистанции меньше 20 единиц (это расстояние задается специально меньше радиуса сферического коллайдера танка-игрока, поэтому бот должен будет вначале поворачивать башню и стрелять, а уже затем будучи внутри сферического коллайдера сам поворачиваться и одновременно двигаться навстречу танку-игроку), бот должен двигаться и поворачиваться в направлении к игроку (здесь: **rotSeedTank** – скорость поворота танка-бота, определяемая функцией **Slerp()**; **moveSpeed** – скорость движения танка-бота, определяемая функцией **Lerp()** ):

**if (distance < 20)**

**{transform.rotation = Quaternion.Slerp(transform.rotation, newRot, Time.deltaTime \* rotSeedTank);  
 transform.position = Vector3.Lerp(transform.position, other.transform.position, Time.deltaTime \* moveSpeed); }**

1. Программирование обработки попадания «луча» от башни танка-бота в танк-игрока с запуском короутины выстрела танка-бота, структура метода Physics.Raycast().

//МЕТОД «БРОСАНИЯ ЛУЧЕЙ» для определения момента, повернута ли башня на игрока

//Physics.Raycast создает «луч» в заданном направлении.

if (Physics.Raycast(stvol.position, stvol.TransformDirection(Vector3.right), out hit))

{ //если луч попал в коллайдер игрока и можно выстрелить

if ((hit.transform.tag == "Player") && canshoot)

//запускаем короутину для выстрела танка-бота

StartCoroutine(botshoot());

}

//КОРОУТИНА ДЛЯ ЗАПУСКА ВЫСТРЕЛА ТАНКА-БОТА

IEnumerator botshoot()

{

//указываем, что танк-бот стрелять пока не может

canshoot = false;

//определяем координату для положения снаряда танка-бота

Vector3 forwardofstvol = stvol.transform.position + stvol.transform.TransformDirection(Vector3.zero);

//создаем снаряд из префаба снаряда в требуемой координате относительно ствола

GameObject newcore = Instantiate(core, forwardofstvol, stvol.rotation);

//3 секи перзарядка

yield return new WaitForSeconds(3f);

//указываем, что танк может сделать выстрел

canshoot = true;

}

1. Программный код короутины для запуска снаряда танка-бота с задержкой по времени, назначение короутины.

//КОРОУТИНА ДЛЯ ЗАПУСКА ВЫСТРЕЛА ТАНКА-БОТА

IEnumerator botshoot()

{

//указываем, что танк-бот стрелять пока не может

canshoot = false;

//определяем координату для положения снаряда танка-бота

Vector3 forwardofstvol = stvol.transform.position + stvol.transform.TransformDirection(Vector3.zero);

//создаем снаряд из префаба снаряда в требуемой координате относительно ствола

GameObject newcore = Instantiate(core, forwardofstvol, stvol.rotation);

//3 секи перзарядка

yield return new WaitForSeconds(3f);

//указываем, что танк может сделать выстрел

canshoot = true;

}