УДК 004.94

Кулибаба Максим Игоревич,

Студент

Государственный Университет Аэрокосмического Приборостроения

Г. Санкт-Петербург

Мужайло Михаил Константинович,

Студент

Государственный Университет Аэрокосмического Приборостроения

Г. Санкт-Петербург

***Научный руководитель: Савельев А.И.*, к.т.н., доцент**

Государственный Университет Аэрокосмического Приборостроения

Г. Санкт-Петербург

РАЗРАБОТКА МЕТОДА ДЛЯ СОЗДАНИЯ МОДУЛЬНЫХ КИБЕРФИЗИЧЕСКИХ ПРОСТРАНСТВ В ПРОГРАММЕ UNITY

Аннотация: В статье представлен метод создания модельного киберфизического пространства. Цель работы состоит в том, чтобы разработать метод формирования виртуального киберфизического пространства для тестирования и отладки алгоритмов управления робототехническим средством мульти роторного типа.

Ключевые слова: моделируемое пространство, виртуальное моделирование, киберфизические пространства, разработка метода, БпЛА

Kulibaba Maxim Igorevich,

Student

State University of Aerospace Instrumentation

St. Petersburg

Muzhailo Mikhail Konstantinovich,

Student

State University of Aerospace Instrumentation

St. Petersburg

Scientific supervisor: **Savelyev A.I., Candidate of Technical Sciences, Associate Professor**

State University of Aerospace Instrumentation

St. Petersburg

DEVELOPMENT OF A METHOD FOR CREATING MODULAR CYBERPHYSICAL SPACES IN THE UNITY PROGRAM

Abstract: The article presents a method for creating a model cyberphysical space. The purpose of the work is to develop a method for forming a virtual cyber-physical space for testing and debugging control algorithms for a multi-rotor type robotic vehicle.

Keywords: simulated space, virtual modeling, cyberphysical spaces, space for testing algorithms

Для отработки управления БпЛА в реальных условиях было бы целесообразно заняться тестированием и подготовкой персонала и управления дронов в визуальном мире. Для этого необходимо разработать киберфизическое пространство, в котором бы происходила тестирование летательного аппарата.

Перед созданием программы необходимо было разработать основную логику программы, которой необходимо будет придерживаться на протяжении всей работы.

Основной принцип работы симулятора для тестирования БпЛА мульти роторного типа показан на рисунке 1:

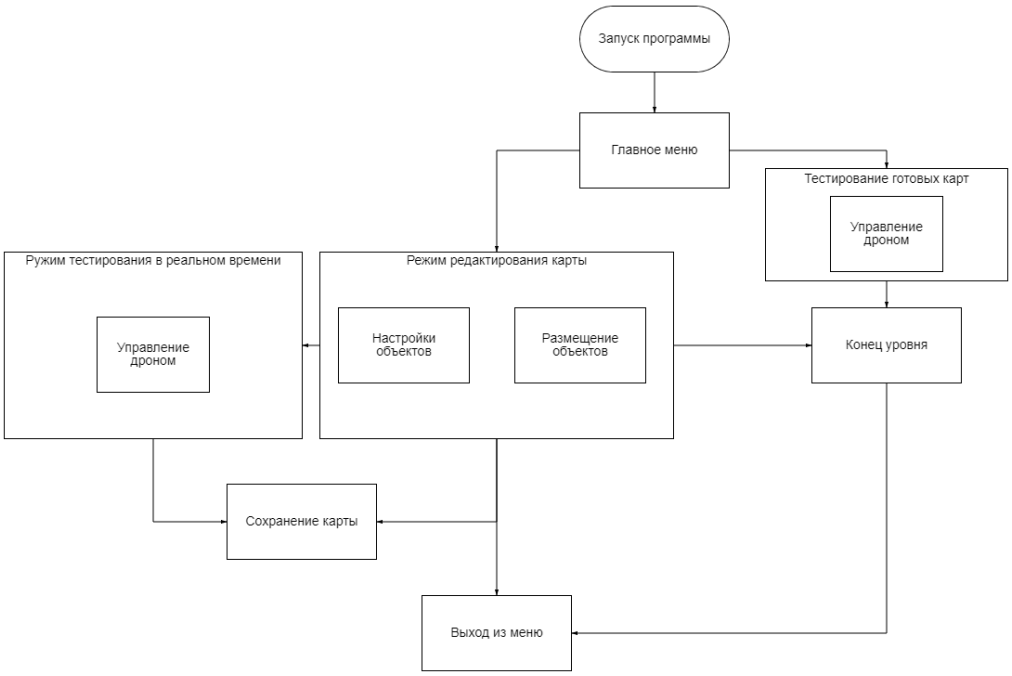

Рисунок 1 – Основной принцип работы симулятора

Принцип работы заключается в том, что при запуске программы у пользователя будет выбор, открыть и пройти существующую карту, для тестирования БпЛА или же создать новую.

В режиме редактирования карт создаётся новая карта, в которой можно создавать модульное тестируемое пространство. В этом режиме возможно, как и строительство, так и переключение в режим тестирования карты в реальном времени.

Сами объекты можно будет увеличивать и уменьшать в размерах, а также вращать в вокруг совей оси и не только.

Когда карта для тестирования БпЛА готова, её можно сохранить, чтобы в следующий раз продолжить тестирования либо редактирование карты. После чего можно выйти из приложения.

Когда все основные объекты были готовы, их нужно было импортировать в среду разработки для дальнейшего их использования в симуляторе. Для среды разработки была выбрана программа Unity3D. Обладая простотой и огромному сообществу с различными библиотеками, Unity3D является наилучшим вариантом для создания в ней симулятора для тестирования БпЛА.

Unity Technologies создала Unity3d, ультрасовременный кроссплатформенный движок для разработки игр и приложений. С помощью этого движка можно создавать программы для различных платформ, включая игровые консоли, мобильные устройства и настольные компьютеры.

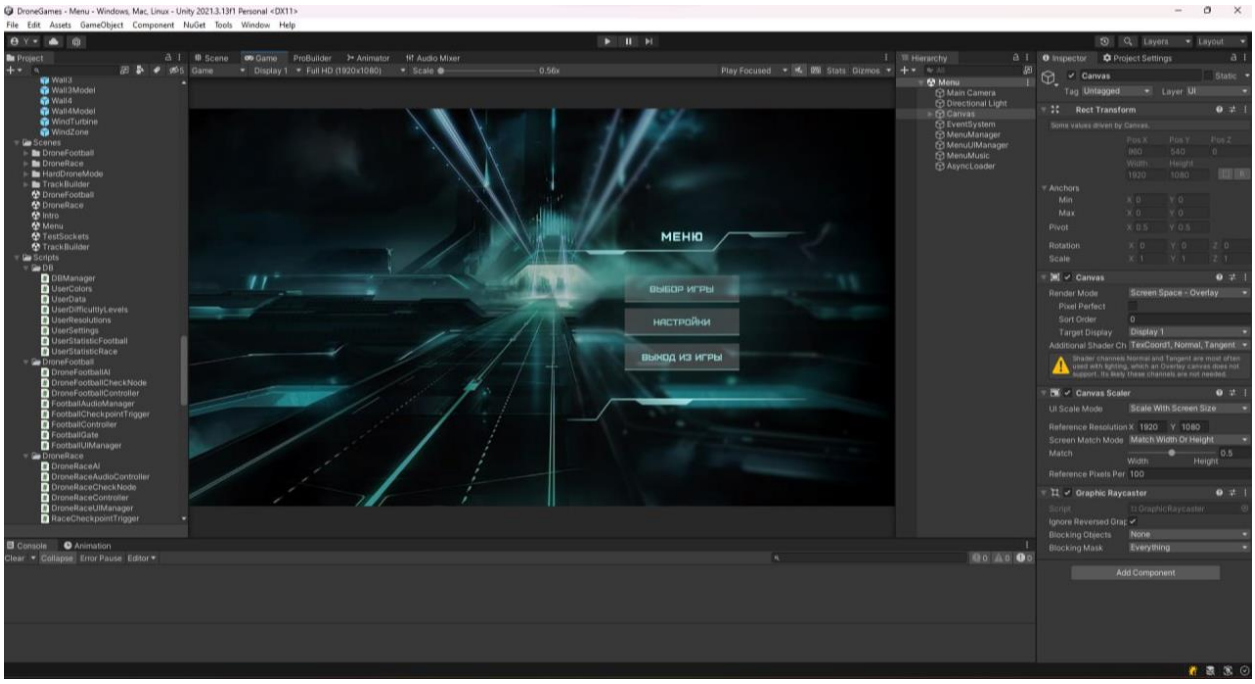

Рисунок 2 – Среда разработки Unity3D

Что бы импортировать объект из Blender3D в Unity3D необходимо, после создания модели в Blender, экспортировать все готовые модульные блоки в формате, который может быть использован в Unity3D. Для этого можно выбрать формат файлов. fbx или .obj. Воспользуемся форматом. fbx для экспорта модели. Необходимо выбрать опцию "File" в верхнем меню,

затем "Export" и выбрать формат. fbx из списка. После этого в указанную директорию сохранится нужный файл.

После экспорта модели из Blender, необходимо импортировать ее в Unity3D. Для этого можно выбрать опцию "Assets" в верхнем меню самой программы, затем "Import New Asset". В появившемся окне нужно выбрать файл модели, который был экспортирован из Blender, и нажать кнопку "Import".

Когда импорт модели в Unity3D завершён, необходимо настроить параметры импорта. Некоторые из наиболее важных параметров включают в себя масштабирование, поворот, настройку материалов и т.д. Настройки параметров импорта можно выполнить в окне "Inspector" после выбора модели в окне "Project".

Размещение модели на сцене. Когда были настроены параметры импорта моделей, их можно было размещать на сцене в Unity3D. Это можно сделать, перетащив модель из окна "Project" на сцену.

Когда были правильно импортированы и настроены все виды блоков, можно было приступать к разработке кода для расположения объектов в пространстве.

Создадим первую страницу с кодом, отвечающую за правильное расположение объектов в пространстве и за модульную состыковку объектов. Назовём его “BuilderManager”.

Для начала сделаем правильно начально расположение камеры в окне редактирования.

Листинг 1 – Скрипт создание правильного расположения камеры

```
private void Awake()
        {
            _mainCamera = Camera.main;
```

```
            _startPointerSize =
builderUI.pathArrow.sizeDelta;
            _selection = FindObjectOfType<Selection>();

_selection.Select(droneBuilderController.gameObject);
            _selection.Deselect();

            for (int i = 0; i <
builderUI.createButtons.Count; i++)
            {
                var i1 = i;

builderUI.createButtons[i].onClick.AddListener(delegate {
SelectObject(i1); });
            }
        }

        private void Start()
        {
            if (isLoadLevel)
            {
                StartCoroutine(LoadScene());
            }
            else if (isGameLevel)
            {
                loadingComplete += TestLevel;
                StartLevel();
            }
            else
            {
                CreateObjectsPoolScene();
            }
        }
```

Рисунок 3 – Режим редактирования

На рисунке 3 представлен главное меню режима редактирования, в котором можно выстраивать различные уровни.

После сделаем так чтобы выбираемый объект привязывался к курсору до момента пока не будет отпущена левая кнопка мыши. После чего происходит привязка объекта либо к пустой точке пространства, от которой создаётся начало отсчёта, либо к тригеру другого объекта.

В скрипте создать переменную для хранения позиции курсора в момент нажатия на объект. В методе записать текущую позицию курсора в переменную. В методе вычислить разницу между текущей позицией курсора и позицией курсора в момент нажатия на объект. Использовать полученную разницу для изменения позиции объекта.

Листинг 2 – Скрипт привязки к курсору и размещение объекта

```
public void PutObject()
        {
            try
            {
                if (pendingObjects.Count != 1) return;
```

```
TrackBuilderUtils.ChangeLayerRecursively(pendingObject.transf
orm, LayerMask.NameToLayer("TrackGround"));
                currentObjectType = null;
                pendingObject = null;
            }
            catch
            {
                // ignored
            }
        }
public void SelectObject(int index)
        {
            if(pendingObject != null)
                PlaceObjects();

            currentSelectObjectIndex = index;
            pendingObject = Instantiate(objects[index],
mousePos, transform.rotation);
            pendingObjects.Add(pendingObject);
            SceneManager.MoveGameObjectToScene(pendingObject,
levelScene);
            objectsPool.Add(pendingObject);
            _selection.Deselect();
            _selection.Select(pendingObject);
            currentObjectType =
pendingObject.GetComponent<TrackObject>();
            currentObjectType.isActive = true;

            if (currentObjectType.objectType ==
ObjectsType.Gate)
            {

currentObjectType.GetComponent<BuilderCheckpointTrigger>().ch
eckpointId =
                    droneBuilderCheckNode.nodes.Count;

droneBuilderCheckNode.AddNode(pendingObject.transform);
            }
        }

        private void PasteObject(GameObject obj)
        {
            if(obj == null)
                return;

            pendingObject = Instantiate(obj, mousePos,
copyObject.transform.rotation);
            pendingObjects.Add(pendingObject);

TrackBuilderUtils.ChangeLayerRecursively(pendingObject.transf
orm, LayerMask.NameToLayer("Track"));
```

```
            SceneManager.MoveGameObjectToScene(pendingObject,
levelScene);
            objectsPool.Add(pendingObject);
            _selection.Deselect();
            _selection.Select(pendingObject);
            currentObjectType =
pendingObject.GetComponent<TrackObject>();
            currentObjectType.isActive = true;

            if (currentObjectType.objectType ==
ObjectsType.Gate)
            {

currentObjectType.GetComponent<BuilderCheckpointTrigger>().ch
eckpointId =
                    droneBuilderCheckNode.nodes.Count;

droneBuilderCheckNode.AddNode(pendingObject.transform);
            }
        }
```

Так же неплохо бы сделать модульную сетку для ровного расположения блоков при правильной их геометрии по отношению друг к другу. Однако у нас будет изменяться параметры объекта, отвечающие за его размеры и углы поворота. Поэтому этот пункт отпадает из реализации.

Далее необходимо сделать возможность увеличивать и уменьшать объекты путём вращения колёсика мыши. С каждым шагом вращения объект будет уменьшаться либо увеличиваться на 0.125x. Полученный скрипт можно увидеть в Листинге 3:

Листинг 3 – Код увеличения и уменьшения объектов

```
private void ChangeObjectHeight(float value)
        {
            if(currentObjectType == null)
                return;

            currentObjectType.yOffset += value;
                // _mainCamera.transform.Translate(0, value,
0, Space.Self);
        }

        private void ChangeObjectScale(float value)
        {
            var newScale = pendingObject.transform.localScale
* value;
```

```
            var newOffset = currentObjectType.yOffset *
value;

            if (newScale.magnitude <= 8f &&
newScale.magnitude >= 0.125f)
            {
                pendingObject.transform.localScale =
newScale;
                currentObjectType.yOffset = newOffset;
                currentObjectType.maxMouseDistance *= value;
            }
        }
```

Когда объект уже подогнан по риазмерам, необходимо определиться с углом поворота данного объекта. Для этого добавим возможность вращать объект на определённый угол. Данный метод был был реализован при помощи изменения угла поворота относительно угла “Z” на величину, записанную в переменную “rotateAmount”

Листинг 4 – Вращения объекта

```
private void RotateObject(Vector3 axis, float rotateAmount,
Space space)
        {
            if(pendingObject == null)
                return;
            pendingObject.transform.Rotate(axis,
rotateAmount, space);
        }
```

Для модульного соединения блоков в пространстве с правильным их размещением друг с другом, был написан и разработан код добавляющий тригеры на каждый из блоков, которые в соприкосновении друг с другом сцепляют блоки между собой идеально ровно, строго соответствуя координатной решётке.

В Unity3D триггеры — это компоненты, которые могут обнаруживать столкновения объектов в игровом мире. Триггеры используются для создания различных эффектов и взаимодействий между объектами.

Каждый триггер представляет собой некую зону в визуальном пространстве, у которой могут быть настроены размеры и формы. Когда другой объект входит в зону триггера, событие OnTriggerEnter вызывается автоматически. Когда этот объект покидает зону триггера, вызывается событие OnTriggerExit.

Чтобы создать триггер в Unity3D, необходимо добавить компонент Collider к объекту и установить его тип на Trigger. Затем можно настроить размеры и форму триггера, используя параметры компонента Collider.

Важно помнить, что триггеры не имеют физического воздействия на объекты, которые входят в зону. Они просто обнаруживают столкновения и вызывают события OnTriggerEnter и OnTriggerExit.

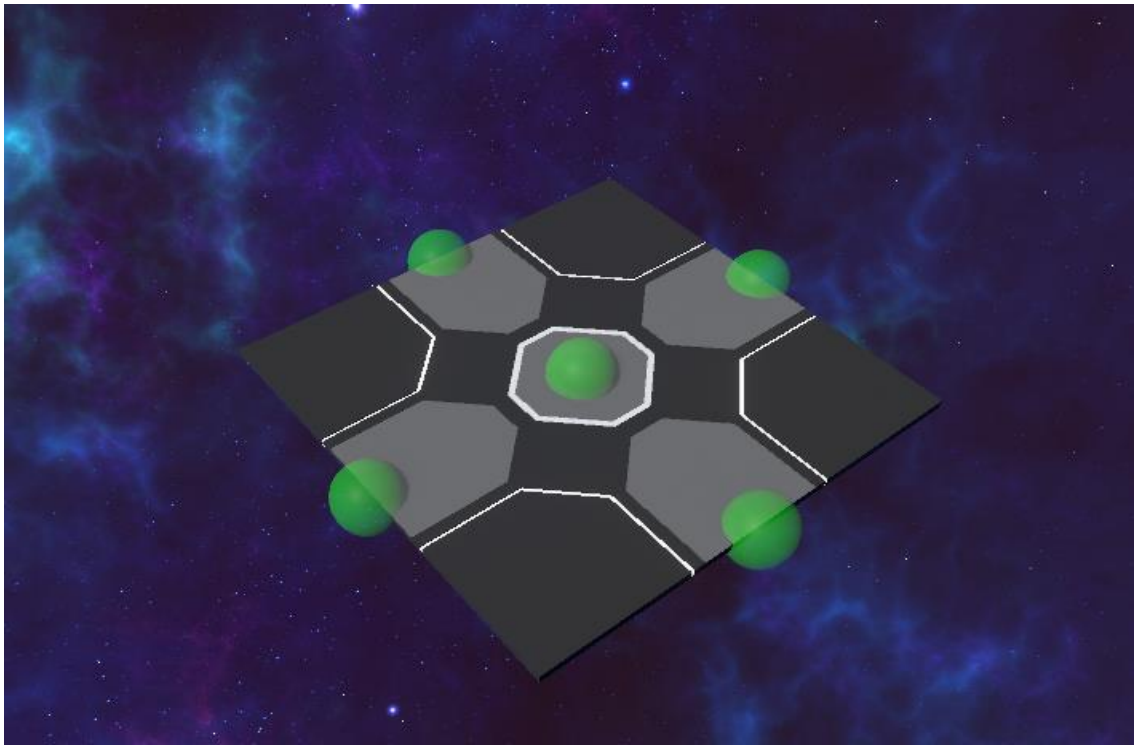

Рисунок 4 – Тригеры на объекте

Когда были добавлены тригеры на все существующие блоки, необходимо было запрограммировать их для связки блоков между собой различной формы. Полный код можно увидеть в Приложении Б

Листинг 5 – Тригерная система

```
namespace Builder
{
    public class Connection : MonoBehaviour
    {
        private TrackObject _trackObject;
        private BuilderManager _builderManager;
        private Selection _selection;
        private Vector3 _connectPosition;

        private void Awake()
        {
            _trackObject =
GetComponentInParent<TrackObject>();
        }

        private void Start()
        {
            _builderManager =
FindObjectOfType<BuilderManager>();
            _selection = FindObjectOfType<Selection>();
        }

        private void OnTriggerEnter(Collider other)
        {
            if (_trackObject.isActive &&
_builderManager.pendingObjects.Count == 1)
            {
                if(_builderManager.pendingObject == null)
                    return;

                if (_trackObject.objectType ==
ObjectsType.Floor &&
                    other.gameObject.layer ==
LayerMask.NameToLayer("FloorConnection"))
                {
                    _builderManager.PutObject();
                    _trackObject.transform.position =
other.transform.position;
                    _trackObject.transform.rotation =
other.transform.rotation;
                }
                else if (_trackObject.objectType ==
ObjectsType.Wall &&
                         other.gameObject.layer ==
LayerMask.NameToLayer("WallConnection"))
                {
                    _builderManager.PutObject();
                    _connectPosition = new
Vector3(other.transform.position.x,
```

```
                        other.transform.position.y +
_trackObject.yOffset, other.transform.position.z);
                    _trackObject.transform.position =
_connectPosition;
                }
                else if (_trackObject.objectType ==
ObjectsType.Slant &&
                         other.gameObject.layer ==
LayerMask.NameToLayer("SlantConnection"))
                {
                    _builderManager.PutObject();
                    _trackObject.transform.position =
other.transform.position;
                }
            }
        }

        private void OnTriggerStay(Collider other)
        {
            if (_trackObject.isActive &&
_builderManager.pendingObjects.Count == 1)
            {
                if (_trackObject.objectType ==
ObjectsType.Floor &&
                    other.gameObject.layer ==
LayerMask.NameToLayer("FloorConnection"))
                {
                    if
(Vector3.Distance(other.transform.position,
_builderManager.mousePos) > _trackObject.maxMouseDistance)
                    {
                        _selection.Move();
                    }
                }
                else if (_trackObject.objectType ==
ObjectsType.Wall &&
                         other.gameObject.layer ==
LayerMask.NameToLayer("WallConnection"))
                {
                    if
(Vector3.Distance(other.transform.position,
_builderManager.mousePos) > _trackObject.maxMouseDistance)
                    {
                        _selection.Move();
                    }
                }
                else if (_trackObject.objectType ==
ObjectsType.Slant &&
                         other.gameObject.layer ==
LayerMask.NameToLayer("SlantConnection"))
                {
```

```
                    if
(Vector3.Distance(other.transform.position,
_builderManager.mousePos) > _trackObject.maxMouseDistance)
                    {
                        _selection.Move();
                    }
                }
            }
        }
    }
}
```

После того как основные моменты были готовы, необходимо было сделать перемещение из режима редактирования карты в режим тестирования. Осуществляться переход между режимами будет посредством нажатия кнопки “Tab”.

Для режима тестирования камера переключается сразу на дрона, а также появляется возможность управлять БпЛА как с клавиатуры, так и с браслетов.

Ниже представлен код для перехода между режимами:

Листинг 7 – Код переключения режимов

```
private void CheckTabPanel()
        {
            if (Input.GetKeyDown(KeyCode.Escape) && isMove)
            {
                _isTabPanel = !_isTabPanel;
                if(_isTabPanel)

droneBuilderSoundController.droneFly.Stop();
                else

droneBuilderSoundController.droneFly.Play();

builderUI.editorTabPanel.SetActive(_isTabPanel &&
!isGameMode);
                builderUI.gameTabPanel.SetActive(_isTabPanel
&& isGameMode);
                builderUI.levelResultPanel.SetActive(false);
                Time.timeScale = _isTabPanel ? 0f : 1f;
            }
        }
```

При создании карты можно допустить ошибки, поставив объект не туда. Эту проблему можно решить двумя способами.

- Удаление объекта с карты;

- Повторное движение объекта.

Оба способа были добавлены в симулятор для упрощения и вариативности пользования.

Удаление объекта реализовано следующим образом. Когда выделяется объект, путём клика левой кнопки мыши по объекту, можно начать кнопку “Delete” и удаления его из директории. Ниже представлен листинг программы.

Листинг 8 – Удаления объектов

```
private void ClearObject()
        {
            foreach (var obj in objectsPool)
            {
                Destroy(obj);
            }
            objectsPool.Clear();
            pendingObjects.Clear();
        }
```

Второй способ реализован путём возвращения объекту режима перемещения. После того как объект был поставлен и привязан к пустой точке пространства, он лишается возможности перемещаться, так как убирается привязка к курсору. Добавлена кнопка для реализации этого пункта.

После выделения уже стоящего блока необходимо нажать на кнопку “Двигать” чтобы объект перешёл к предыдущему состоянию. Он снова привязан к курсору

Листинг 9 – Повторное движение объекта

```
public void MoveObjectsToPoolScene()
        {
            foreach (var obj in objectsPool)
            {

if(obj.GetComponent<BuilderCheckpointTrigger>())

droneBuilderCheckNode.nodes.Add(obj.GetComponent<BuilderCheck
pointTrigger>());
                SceneManager.MoveGameObjectToScene(obj,
levelScene);
            }

            if (droneBuilderCheckNode.nodes.Count > 0)
            {

builderUI.pathArrow.gameObject.SetActive(true);
            }

            FindObjectOfType<Server>().droneBuilderController
= droneBuilderController;
        }
```

Все основные моменты для симуляторы были добавлены. Можно приступать к тестированию виртуального киберфизического пространства.

Использованные источники

1. Unity - Manual: PlayerPrefs. URL: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (дата обращения: 23.04.2023).

2. Will Goldstone. Unity 3.x Game Development, Packt Publishing; 2 editions, 2011. – 488c.

3. Торн. А. Основы анимации в Unity / А. Торн. — пер. с англ. Рагимова Р. — Москва: ДМК Пресс, 2016. — 176 с.

4. Блэкманн, С. Моделирование с помощью Unity – М.: АСТПРЕСС, 2011. – 147 с.