**ОПИСАНИЕ**

Проект реализован на базе движка Unity 2022 (язык C#).

Данный проект по автогенерации датасета построен на основе взаимодействия трех различных уровней классов. На верхнем уровне существует некоторый глобальный управляющий элемент **GlobalController** – при старте он в автоматическом режиме последовательно через интерфейсы (**ITransformable**, **IRaycastable** и **IScreenshotable**) вызывает методы реализующих эти интерфейсы классов (**ObjectTransformer, RaycastGenerator** и **ScreenshotMaker**). Количество таких последовательных вызовов ограничено рядом факторов.

Интерфейсы являются промежуточным уровнем между глобальным управлением и реализацией. Они же связывают ручное управление (которое представлено классами (**InputObject** и **InputScreenshotCamera**) с той же реализацией. Ручное управление является важным инструментом отладки (проверки выполнения граничных значений, корректного выполнения кода, успешного сохранения данных и др.).

Классы **ObjectTransformer, RaycastGenerator** и **ScreenshotMaker**, являющиеся реализацией соответствующих интерфейсов, представляют из себя последовательный цикл: **ObjectTransformer** – трансформация объекта (его перемещение в специально отведенном для него пространстве на сцене, вращение объекта, его масштабирование и замена исследуемого объекта на следующий по списку после полного прохождения цикла «перемещение-вращение-масштабирование»); **RaycastGenerator** – запуск лучей, с помощью которых определяется расстояние от плоскости до поверхности объекта; **ScreenshotMaker** – получение изображение объекта с нужного ракурса и сохранение в требуемом формате с уникальным именем.

Пользователю доступны гибкие инструменты регулирования процессом автогенерации:

1. Изменение значений объекта:
   1. шаг перемещения на сцене;
   2. значение угла поворота;
   3. шаг масштабирования.
2. Добавление новых объектов к имеющемуся списку;
3. Управление количеством лучей, испускаемых от плоскостей;
4. Изменение времени итерации глобального управления;

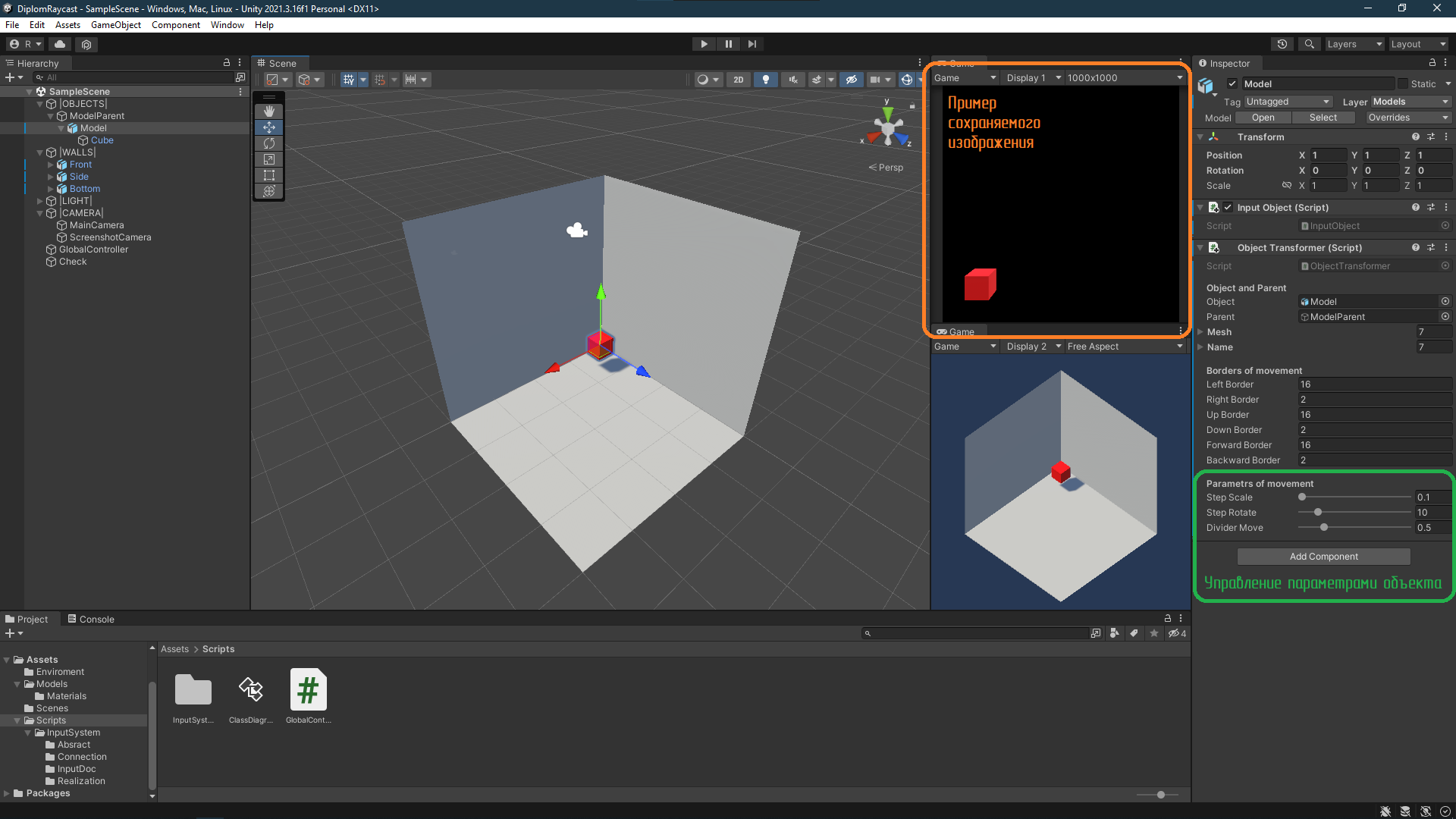


Рис. 1 Интерфейс Unity

Пример одной итерации:

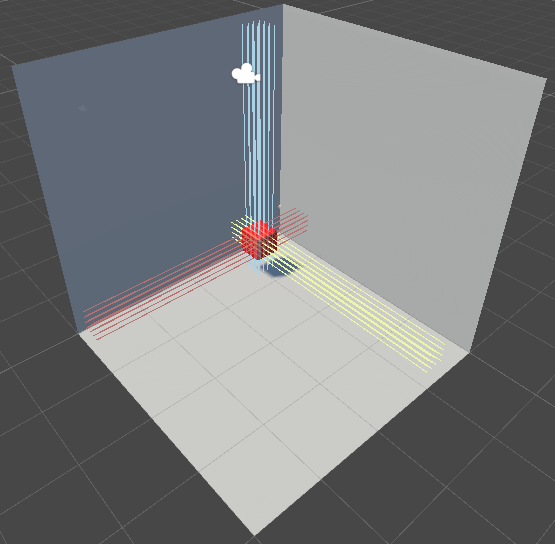


Рис. 2. Пример работы

Объект (куб) был перемещен на установленный шаг, от плоскостей до объекта было пущено заданное количество лучей (эти данные были записаны в файл .txt – см. рис.3), после чего было сохранено изображение объекта (см. рис.4).

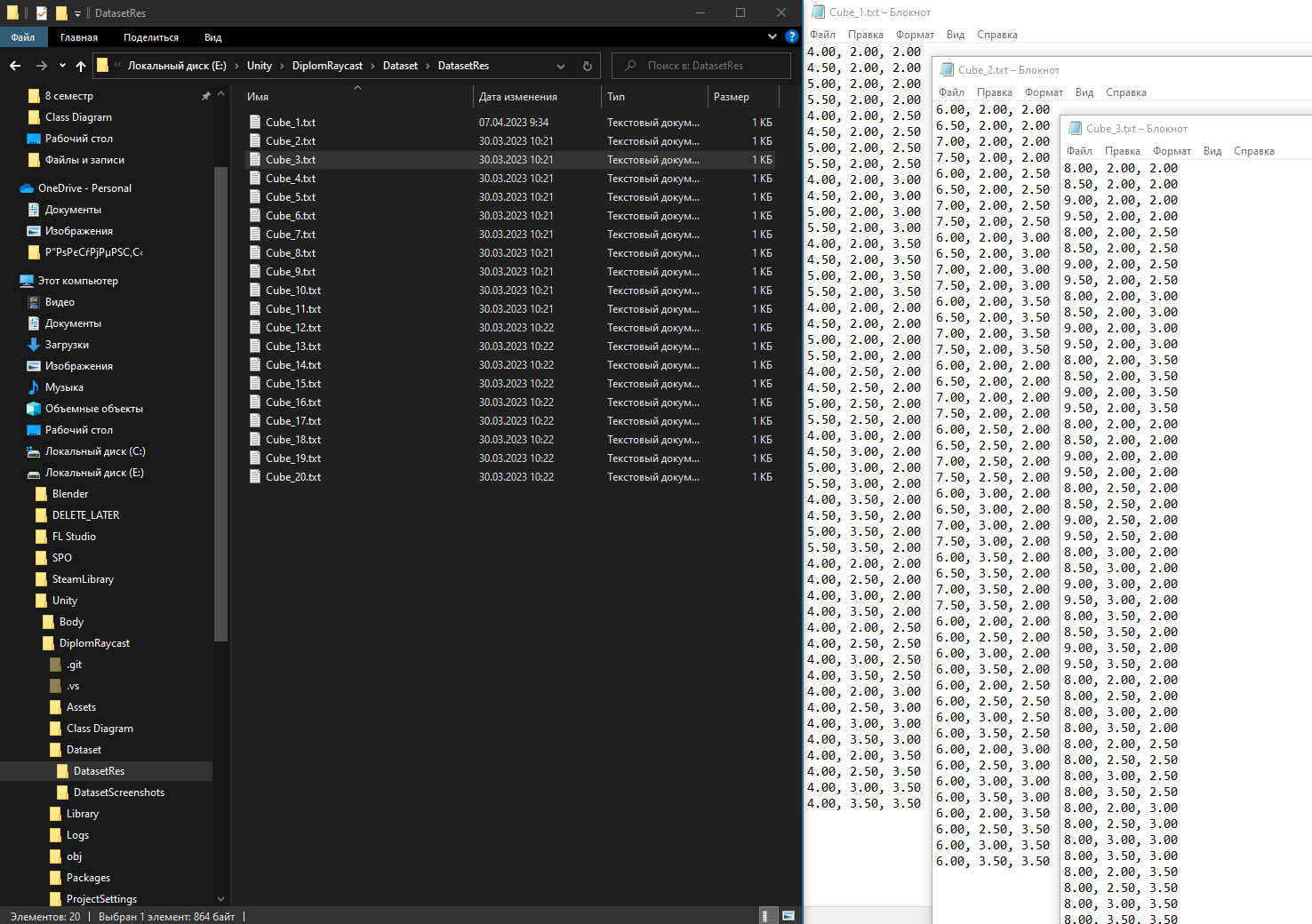


Рис. 3. Записанные данные

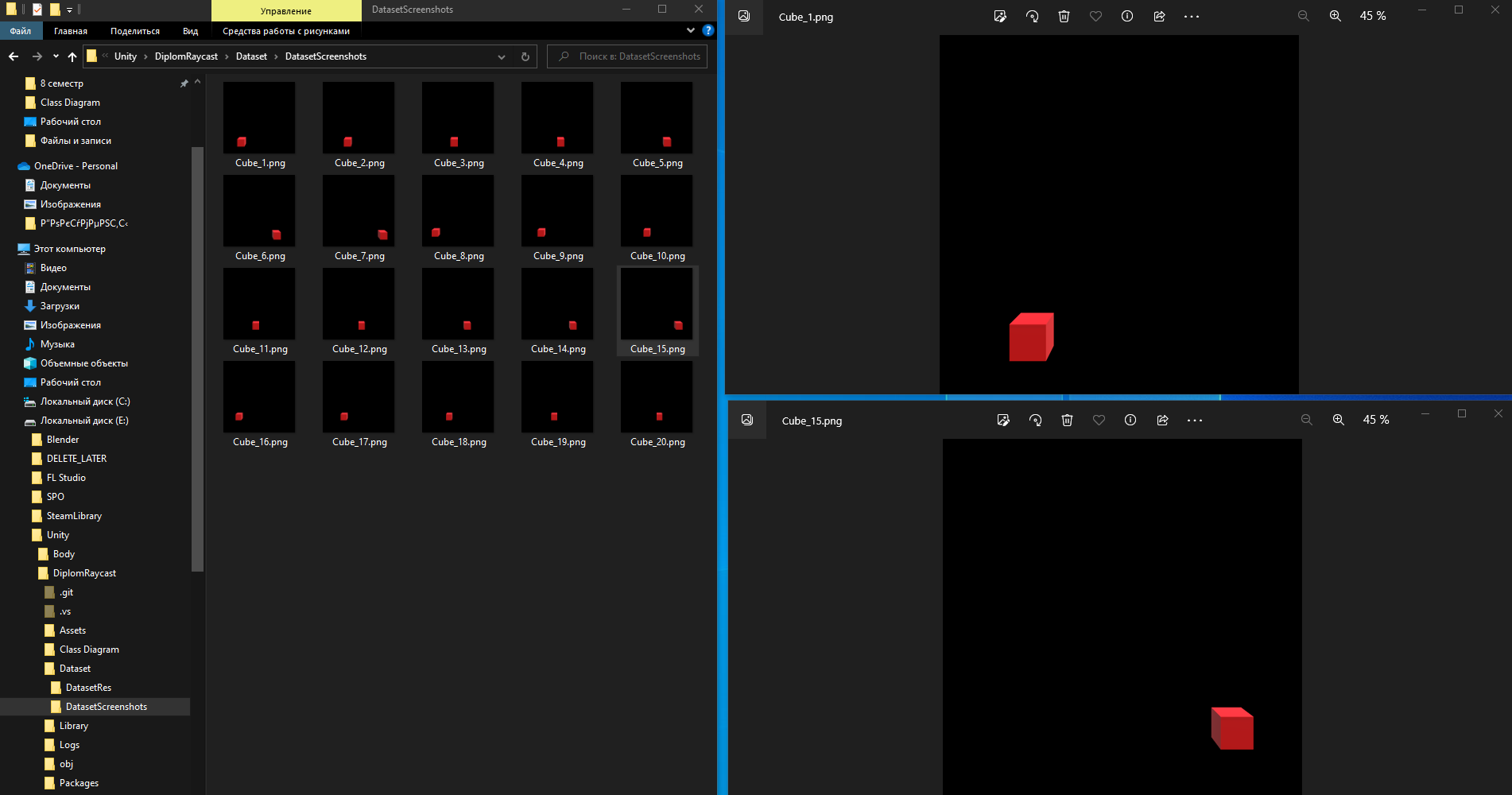


Рис. 4. Изображения

**АБСТРАКЦИЯ (ИНТЕРФЕЙСЫ)**

**ITransformable**

using UnityEngine;

public interface ITransformable

{

void ManualInputMove(Vector3 inputDirection);

void AutomaticMove();

Vector3 GetCurrentModelPosition();

Vector3 GetModelSize();

string GetModelName();

int GetTransformIteration();

}

**IRaycastable**

using System.Collections.Generic;

using UnityEngine;

public interface IRaycastable

{

void ManualInputRaycast();

List<Vector3> AutomaticRaycast(Vector3 startPosition, Vector3 area);

}

**IScreenshotable**

public interface IScreenshotable

{

void ManualInputScreenshot();

void AutomaticScreenshot();

void AutomaticScreenshot(string \_name, int number);

}

**СВЯЗЬ**

**InputObject**

using System;

using UnityEngine;

using UnityEngine.InputSystem;

public class InputObject : MonoBehaviour

{

private InputDoc \_inputDoc;

private ITransformable \_movable;

private void Awake()

{

if (\_inputDoc == null)

{

\_inputDoc = new InputDoc();

}

\_movable = GetComponent<ITransformable>();

if (\_movable == null)

{

throw new Exception($"There is no IMovable on the object: {gameObject.name}");

}

}

private void OnEnable()

{

\_inputDoc.Enable();

\_inputDoc.Management.Move.performed += OnMovePerfermed;

}

private void OnMovePerfermed(InputAction.CallbackContext obj)

{

var inputDirection = \_inputDoc.Management.Move.ReadValue<Vector3>();

\_movable.ManualInputMove(inputDirection);

}

private void OnDisable()

{

\_inputDoc.Disable();

\_inputDoc.Management.Move.performed -= OnMovePerfermed;

}

}

**InputScreenshotCamera**

using System;

using UnityEngine;

using UnityEngine.InputSystem;

public class InputScreenshotCamera : MonoBehaviour

{

private InputDoc \_inputDoc;

private IScreenshotable \_screenshotable;

private void Awake()

{

if (\_inputDoc == null)

{

\_inputDoc = new InputDoc();

}

\_screenshotable = GetComponent<IScreenshotable>();

if (\_screenshotable == null)

{

throw new Exception($"There is no IJumpable on the object: {gameObject.name}");

}

}

private void OnEnable()

{

\_inputDoc.Enable();

\_inputDoc.Management.Screenshot.performed += OnScreenshotPerfermed;

}

private void OnScreenshotPerfermed(InputAction.CallbackContext obj)

{

\_screenshotable.ManualInputScreenshot();

}

private void OnDisable()

{

\_inputDoc.Management.Screenshot.performed -= OnScreenshotPerfermed;

\_inputDoc.Disable();

}

}

**РЕАЛИЗАЦИЯ**

**ObjectTransformer**

using UnityEngine;

public class ObjectTransformer : MonoBehaviour, ITransformable

{

#region Parametrs in Inspector

[Header("Object and Parent")]

[SerializeField] private GameObject \_object;

[SerializeField] private GameObject \_parent;

[SerializeField] private Mesh[] \_mesh;

[SerializeField] private string[] \_name;

[Space(2)]

[Header("Borders of movement")]

[SerializeField] private float \_leftBorder;

[SerializeField] private float \_rightBorder;

[SerializeField] private float \_upBorder;

[SerializeField] private float \_downBorder;

[SerializeField] private float \_forwardBorder;

[SerializeField] private float \_backwardBorder;

[Space(2)]

[Header("Parametrs of movement")]

[SerializeField][Range(0.1f, 1.0f)] private float \_stepScale;

[SerializeField][Range(1, 60)] private byte \_stepRotate;

[SerializeField][Range(0.1f, 2.0f)] private float \_dividerMove;

#endregion

private Vector3 \_inputDirection;

private Vector3 \_startPosition;

private Quaternion \_startRotation;

private Vector3 \_startScale;

private Vector3 \_stepPosition;

private Vector3 \_angleRotation;

private Vector3 \_factorScale;

private byte \_bitCounterRotation;

private byte \_bitCounterScale;

private byte \_maxBitCounter;

private byte \_xBitMask;

private byte \_yBitMask;

private byte \_zBitMask;

private byte \_numberOfMesh;

private int \_transformIteration;

private void Awake()

{

\_inputDirection = Vector3.one;

\_startPosition = \_parent.transform.position;

\_startRotation = \_object.transform.rotation;

\_startScale = \_parent.transform.localScale;

\_stepPosition = \_startPosition;

\_angleRotation = Vector3.zero;

\_factorScale = Vector3.zero;

\_bitCounterRotation = 1;

\_bitCounterScale = 1;

\_maxBitCounter = 8;

\_xBitMask = 1;

\_yBitMask = 2;

\_zBitMask = 4;

\_numberOfMesh = 1;

\_transformIteration = 0;

}

public void ManualInputMove(Vector3 inputDirection)

{

\_stepPosition.x = (float)System.Math.Round(\_stepPosition.x + inputDirection.x / \_dividerMove, 1);

if (\_stepPosition.x > \_leftBorder)

\_stepPosition.x = \_rightBorder;

else if (\_stepPosition.x < \_rightBorder)

\_stepPosition.x = \_leftBorder;

\_stepPosition.y = (float)System.Math.Round(\_stepPosition.y + inputDirection.y / \_dividerMove, 1);

if (\_stepPosition.y < \_downBorder)

\_stepPosition.y = \_upBorder;

else if (\_stepPosition.y > \_upBorder)

\_stepPosition.y = \_downBorder;

\_stepPosition.z = (float)System.Math.Round(\_stepPosition.z + inputDirection.z / \_dividerMove, 1);

if (\_stepPosition.z < \_backwardBorder)

\_stepPosition.z = \_forwardBorder;

else if (\_stepPosition.z > \_forwardBorder)

\_stepPosition.z = \_backwardBorder;

\_object.transform.position = \_stepPosition;

Debug.Log("Object moved (Manual Input)");

}

public void AutomaticMove()

{

\_transformIteration++;

\_stepPosition.x = (float)System.Math.Round(\_stepPosition.x + \_inputDirection.x / \_dividerMove, 1);

if (\_stepPosition.x > \_leftBorder)

{

\_stepPosition.x = \_rightBorder;

\_stepPosition.z = (float)System.Math.Round(\_stepPosition.z + \_inputDirection.z / \_dividerMove, 1);

if (\_stepPosition.z > \_forwardBorder)

{

\_stepPosition.z = \_backwardBorder;

\_stepPosition.y = (float)System.Math.Round(\_stepPosition.y + \_inputDirection.y / \_dividerMove, 1);

if (\_stepPosition.y > \_upBorder)

{

\_stepPosition = \_startPosition;

AutomaticRotate();

if (\_bitCounterRotation >= \_maxBitCounter)

{

AutomaticScale();

\_bitCounterRotation = 1;

if (\_bitCounterScale >= \_maxBitCounter)

{

\_bitCounterScale = 1;

\_transformIteration = 0;

\_object.GetComponentInChildren<MeshFilter>().mesh = \_mesh[\_numberOfMesh % \_mesh.Length];

\_object.GetComponentInChildren<MeshCollider>().sharedMesh = \_mesh[\_numberOfMesh % \_mesh.Length];

\_object.transform.GetChild(0).name = \_name[\_numberOfMesh++ % \_name.Length];

}

}

}

}

}

\_parent.transform.position = \_stepPosition;

}

private void AutomaticRotate()

{

if ((\_bitCounterRotation & \_xBitMask) == \_xBitMask)

\_angleRotation.x += \_stepRotate;

if ((\_bitCounterRotation & \_yBitMask) == \_yBitMask)

\_angleRotation.y += \_stepRotate;

if ((\_bitCounterRotation & \_zBitMask) == \_zBitMask)

\_angleRotation.z += \_stepRotate;

if (\_angleRotation.x <= 360 && \_angleRotation.y <= 360 && \_angleRotation.z <= 360)

\_object.transform.rotation = \_startRotation \* Quaternion.Euler(\_angleRotation);

else

{

\_angleRotation = Vector3.zero;

\_bitCounterRotation++;

}

}

private void AutomaticScale()

{

if ((\_bitCounterScale & \_xBitMask) == \_xBitMask)

\_factorScale.x = (float)System.Math.Round(\_factorScale.x + \_stepScale, 1);

if ((\_bitCounterScale & \_yBitMask) == \_yBitMask)

\_factorScale.y = (float)System.Math.Round(\_factorScale.y + \_stepScale, 1);

if ((\_bitCounterScale & \_zBitMask) == \_zBitMask)

\_factorScale.z = (float)System.Math.Round(\_factorScale.z + \_stepScale, 1);

if (\_factorScale.x <= 5 && \_factorScale.y <= 5 && \_factorScale.z <= 5)

\_parent.transform.localScale = \_startScale + \_factorScale;

else

{

\_parent.transform.localScale = \_startScale;

\_factorScale = Vector3.zero;

\_bitCounterScale++;

}

}

public Vector3 GetCurrentModelPosition()

{

return gameObject.GetComponentInChildren<Renderer>().bounds.min;

}

public Vector3 GetModelSize()

{

return gameObject.GetComponentInChildren<Renderer>().bounds.size;

}

public string GetModelName()

{

return gameObject.transform.GetChild(0).name;

}

public int GetTransformIteration()

{

return \_transformIteration;

}

}

**RaycastGenerator**

using System;

using System.Collections.Generic;

using UnityEngine;

public class RaycastGenerator : MonoBehaviour, IRaycastable

{

#region Parametrs in Inspector

[SerializeField][Range(1, 100)] private int \_countRay;

[SerializeField] private LayerMask \_maskRay;

[SerializeField][ColorUsage(false)] private Color \_color;

#endregion

private Vector3 \_startPosition;

private Vector3 \_currentPosition;

private Vector3 \_directionRay;

private Vector3 \_OffsetAxis1;

private Vector3 \_OffsetAxis2;

private RaycastHit \_hit;

private List<Vector3> \_vertexes;

private string \_planeType;

private void Awake()

{

\_planeType = gameObject.name;

switch (\_planeType)

{

case "Front":

\_directionRay = Vector3.forward;

break;

case "Side":

\_directionRay = Vector3.right;

break;

case "Bottom":

\_directionRay = Vector3.up;

break;

}

\_vertexes = new List<Vector3>();

}

public void ManualInputRaycast()

{

}

public List<Vector3> AutomaticRaycast(Vector3 startPosition, Vector3 area)

{

\_vertexes.Clear();

CalculateOffset(startPosition, area);

for (int i = 0; i < \_countRay; i++)

{

for (int j = 0; j < \_countRay; j++)

{

Physics.Raycast(\_currentPosition, \_directionRay, out \_hit, 20);

Debug.DrawRay(\_currentPosition, \_directionRay \* 20, \_color, 10); // 60 seconds

//Debug.Log(Physics.Raycast(\_ray, 5));

Debug.Log($"{\_planeType}. vertex: " + \_hit.point);

\_vertexes.Add(\_hit.point);

\_currentPosition += \_OffsetAxis1;

}

\_currentPosition = \_startPosition + \_OffsetAxis2 \* (i + 1);

}

Debug.Log("==========================================");

return \_vertexes;

}

private void CalculateOffset(Vector3 startPosition, Vector3 area)

{

switch (\_planeType)

{

case "Front":

\_startPosition.Set(startPosition.x, startPosition.y, 0);

\_OffsetAxis1.Set(area.x / \_countRay, 0, 0);

\_OffsetAxis2.Set(0, area.y / \_countRay, 0);

break;

case "Side":

\_startPosition.Set(0, startPosition.y, startPosition.z);

\_OffsetAxis1.Set(0, area.y / \_countRay, 0);

\_OffsetAxis2.Set(0, 0, area.z / \_countRay);

break;

case "Bottom":

\_startPosition.Set(startPosition.x, 0, startPosition.z);

\_OffsetAxis1.Set(area.x / \_countRay, 0, 0);

\_OffsetAxis2.Set(0, 0, area.z / \_countRay);

break;

}

\_currentPosition = \_startPosition;

}

}

**ScreenshotMaker**

using System;

using UnityEngine;

public class ScreenshotMaker : MonoBehaviour, IScreenshotable

{

public void ManualInputScreenshot()

{

Debug.Log("Screenshot was taken (Manual Input)\n");

ScreenCapture.CaptureScreenshot($"Dataset/DatasetScreenshots/ManualScreenshot {DateTime.Now.ToString("MM/dd/yyyy HH/mm/ss")}.png", 1);

}

public void AutomaticScreenshot()

{

Debug.Log("Screenshot was taken (Automatic Input without parametrs)\n");

ScreenCapture.CaptureScreenshot($"Dataset/DatasetScreenshots/AutoScreenshot {DateTime.Now.ToString("MM/dd/yyyy HH/mm/ss")}.png", 1);

}

public void AutomaticScreenshot(string name, int number)

{

Debug.Log($"Screenshot was taken (Automatic Input with parametrs: {name} {number})\n");

ScreenCapture.CaptureScreenshot($"Dataset/DatasetScreenshots/{name}\_{number}.png");

}

}

**МЕНЕДЖЕР ПРОЕКТА**

**GlobalController**

using System;

using System.Collections;

using System.Collections.Generic;

using System.IO;

using UnityEngine;

public class GlobalController : MonoBehaviour

{

[SerializeField][Range(2, 10)] private byte \_iterationTime;

private IScreenshotable \_screenshotable;

private IRaycastable \_raycastableFront;

private IRaycastable \_raycastableSide;

private IRaycastable \_raycastableBottom;

private ITransformable \_transformable;

private Vector3 \_startPosition;

private Vector3 \_area;

private string \_name;

private List<Vector3> \_vertexes;

private StreamWriter \_file;

private void Awake()

{

\_screenshotable = GameObject.Find("ScreenshotCamera").GetComponent<IScreenshotable>();

if (\_screenshotable == null)

throw new Exception($"There is no IScreenshotable on the object: {gameObject.name}");

\_raycastableFront = GameObject.Find("Front").GetComponent<IRaycastable>();

if (\_raycastableFront == null)

throw new Exception($"There is no IRaycastable on the object: {gameObject.name}");

\_raycastableSide = GameObject.Find("Side").GetComponent<IRaycastable>();

if (\_raycastableSide == null)

throw new Exception($"There is no IRaycastable on the object: {gameObject.name}");

\_raycastableBottom = GameObject.Find("Bottom").GetComponent<IRaycastable>();

if (\_raycastableBottom == null)

throw new Exception($"There is no IRaycastable on the object: {gameObject.name}");

\_transformable = GameObject.Find("Model").GetComponent<ITransformable>();

if (\_transformable == null)

throw new Exception($"There is no IMovable on the object: {gameObject.name}");

}

private void Start()

{

//StartCoroutine(AllOperations());

//AllOperations();

//InvokeRepeating("AllOperations", \_iterationTime, \_iterationTime);

InvokeRepeating("StartAllOperation", \_iterationTime, \_iterationTime);

}

private void StartAllOperation()

{

StartCoroutine(AllOperations());

}

IEnumerator AllOperations()

{

Debug.Log("Started Coroutine at timestamp : " + Time.time);

Debug.Log("==========================================");

\_transformable.AutomaticMove();

\_startPosition = \_transformable.GetCurrentModelPosition();

\_area = \_transformable.GetModelSize();

\_name = \_transformable.GetModelName();

Debug.Log("\_startPosition: " + \_startPosition.ToString("F5"));

Debug.Log("\_area: " + \_area.ToString("F5"));

Debug.Log("\_name: " + \_name);

Debug.Log("==========================================");

yield return new WaitForSeconds(\_iterationTime / 2);

\_screenshotable.AutomaticScreenshot(\_name, \_transformable.GetTransformIteration());

\_vertexes = \_raycastableBottom.AutomaticRaycast(\_startPosition, \_area);

\_vertexes.AddRange(\_raycastableFront.AutomaticRaycast(\_startPosition, \_area));

\_vertexes.AddRange(\_raycastableSide.AutomaticRaycast(\_startPosition, \_area));

//\_vertexes = \_vertexes.Union(\_vertexes).ToList();

Debug.Log("==========================================");

Debug.Log("We have a list of vertexes:");

foreach (Vector3 vertex in \_vertexes)

{

Debug.Log("vert = " + vertex);

}

Debug.Log("==========================================");

Debug.Log("Finished Coroutine at timestamp : " + Time.time);

WriteToFile();

\_vertexes.Clear();

}

private void WriteToFile()

{

\_file = new StreamWriter($"Dataset\\DatasetRes\\{\_name}\_{\_transformable.GetTransformIteration()}.txt");

char[] ch = { '(', ')' };

foreach (Vector3 vertex in \_vertexes)

{

\_file.WriteLine(vertex.ToString().Trim(ch));

}

\_file.Close();

}

}