Командный практический тур

Общая формулировка

Разработка AR-приложения для ученых, занимающихся исследованием космоса. Приложение получает данные с сервера о состоянии ионосферы Земли и положении космических спутников, привязанные к определенным геокоординатам.

Актуальность и значимость

Данные 4D Space — являют собой информацию, о полном электронном содержании ионосферы. Данная информация и ее анализ помогают регулировать радиоволны для усиления принимаемого радиосигнала на земле от спутников.

Основная проблема этих данных — сложность визуализации. Данные заданы 4х-мерным пространством, и соответственно для качественного анализа нужно строить несколько плоских графиков, отражающих профиль слоев ионосферы.

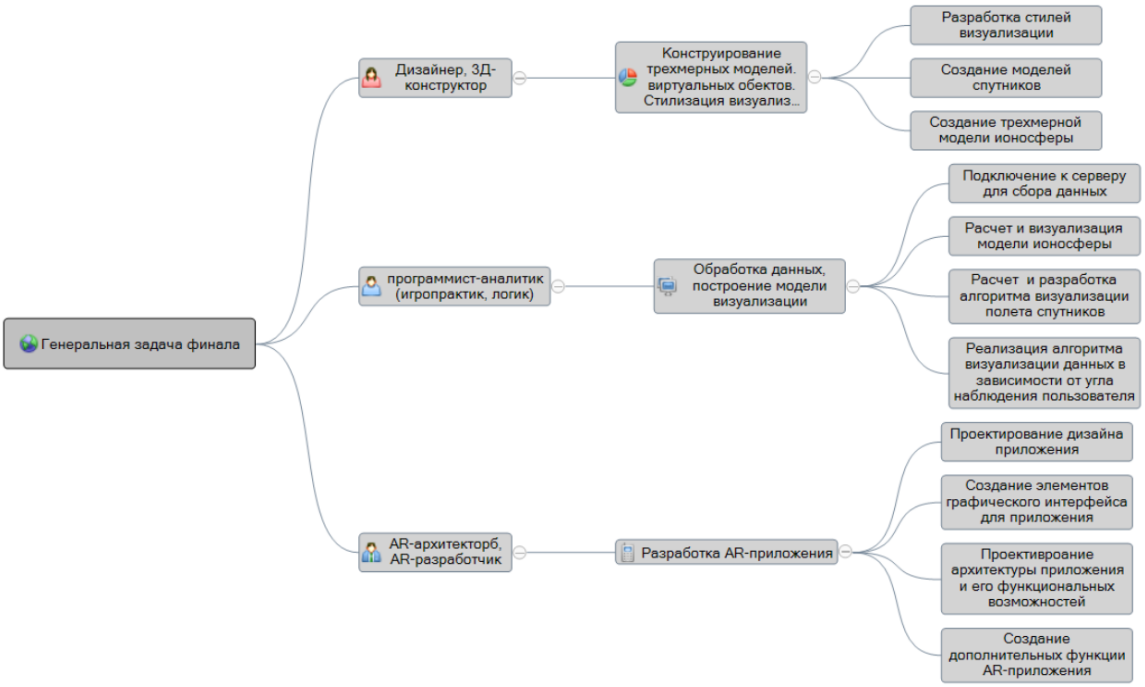
Задача участников разработать инструмент визуализации этих данных с использованием технологии дополненной реальности.

Мобильное AR-приложение позволит построить трехмерную визуализацию собранных данных, выступит в качестве наиболее удобного интерфейса анализа такого рода информации.

Данные меняются динамически во времени, а поскольку они привязаны к определенным геокоординатам, то при смене сектора за которым наблюдает пользователь, «картинка» состояния ионосферы также должна перестраиваться — AR-визуализация автоматически изменяется в зависимости от времени и геокоординат, попадающих в сектор обзора камеры мобильного устройства пользователя.

Подзадачи и распределение по ролям

Можно выделить три основные направления для работы над задачей и решаемые ими подзадачи.



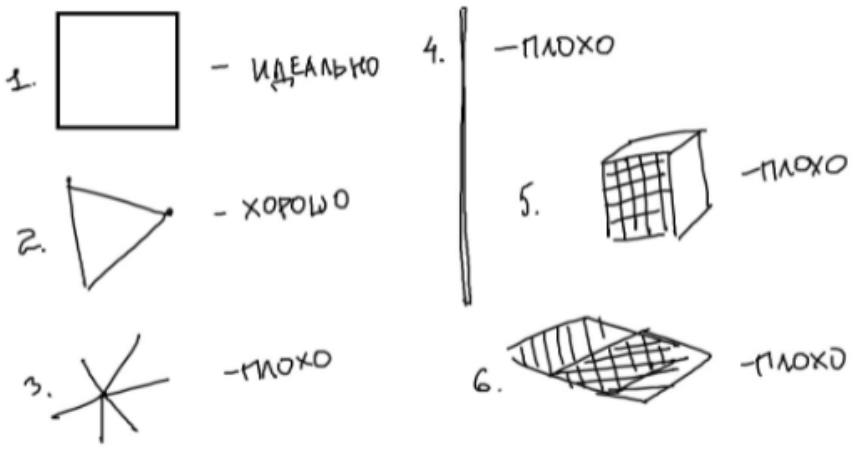
Задачи 3D-дизайнера

3D-дизайнеру предстоит разработка трехмерных моделей виртуального мира, а именно: модель Земли, модель атмосферы, 32 спутника. Кроме того, требуется получить дополнительный сет графических объектов в стилистике фантазийных миров. Например, вселенная океан, Земля превращается в рыбу, а спутники в водоросли. Тематика стилизации может быть любой.

Для всех 3D моделей рекомендуется:

1. В идеале, форма всех полигонов должна быть приближена к квадрату.
2. Tris(треугольники) хуже квадратов, но лучше многоугольников. На фигуру их должно приходится $\sim$ 2000.
3. Избегайте моделей, у которых из одной вершины выходит более 5 ребер. Допустимо некоторое количество таких вершин, но, помните, чем из больше, тем больше ресурсов потребуется системе.
4. Если полигон четырехугольный, но соотношение сторон превышает разумные пределы (допустим, соотношение 20:1), то стоит переделать модель под несколько полигонов.
5. Если модель имеет простую форму, но при этом на этой простой форме много полигонов, то такое количество полигонов грузит систему, а результат как от простой модели.
6. Если два полигона перекрывают друг друга (при просмотре образуется «рябь», если вращаться вокруг объекта), то это приводит к ошибкам рендера.

Наглядно все пункты представлены на картинке ниже. Все пункты влияют на итоговый вес 3D-модели, о котором не стоит забывать. Именно относительно данных рекомендаций будут оцениваться все работы 3D-архитектора.



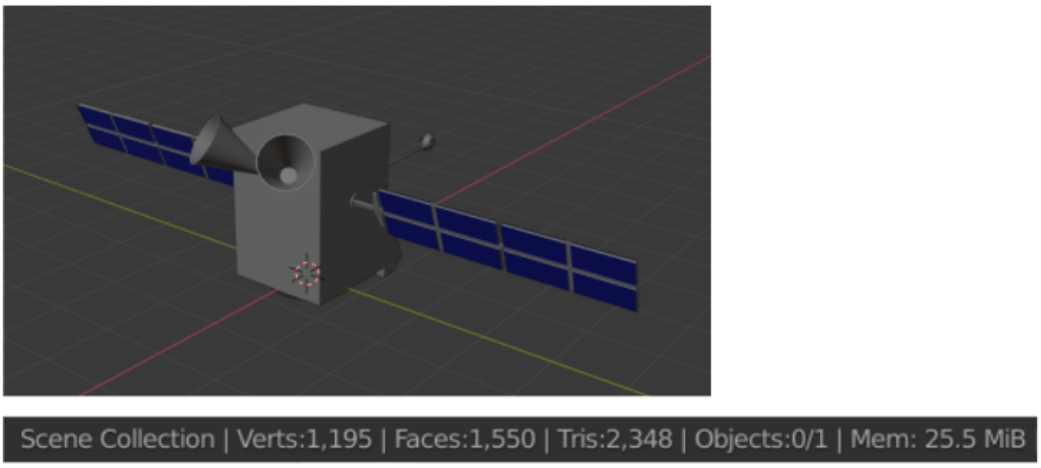
Конструирование 3D-модели спутников

Спутников, которые необходимо отобразить всего 32. Было решено разработать несколько основных принципиально различных моделей спутников (7) и некоторое количество деталей спутников(экраны, антенны, приемники и т.п.) и в дальнейшем производить их модификацию, например, замена или добавление деталей.

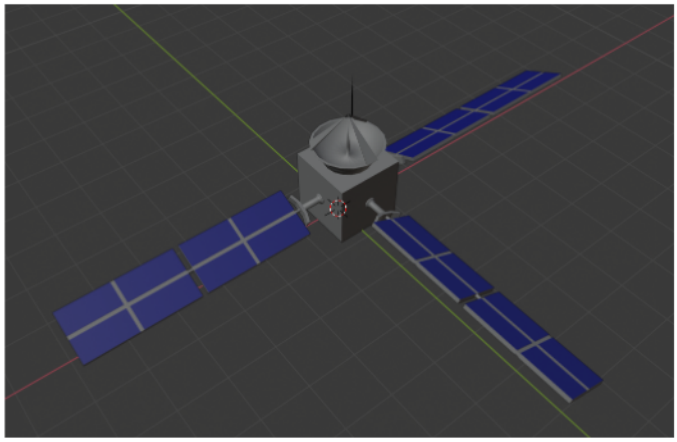
Участникам предоставлялась возможность проявить свой творческий потенциал и продемонстрировать компетенции владения навыками конструирования трехмерных моделей. В данном пункте мы приводим лучшее решение представленное командой ребят из ГБОУ Инженерно-технологическая школа № 777 города Санкт-Петербурга.

Ниже представлены все модели спутников с их основными характеристиками. Для создания 3D-моделей использовался Blender. Использовались простейшие текстуры с заливкой однородным цветом, чтобы не увеличивать размер моделей.

Модель Спутник G01

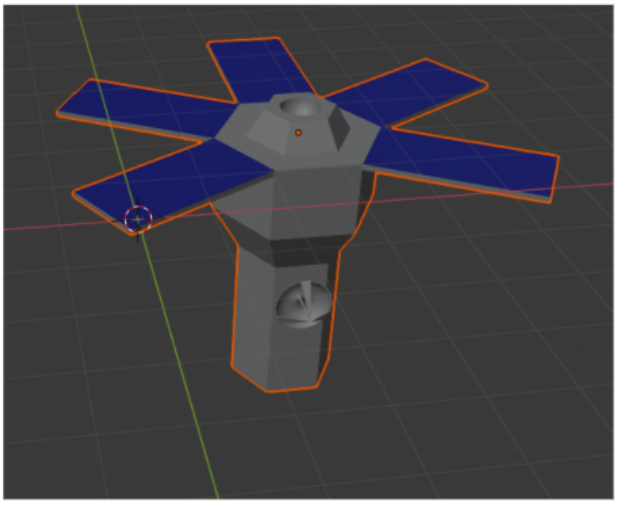


Модель Спутник G02



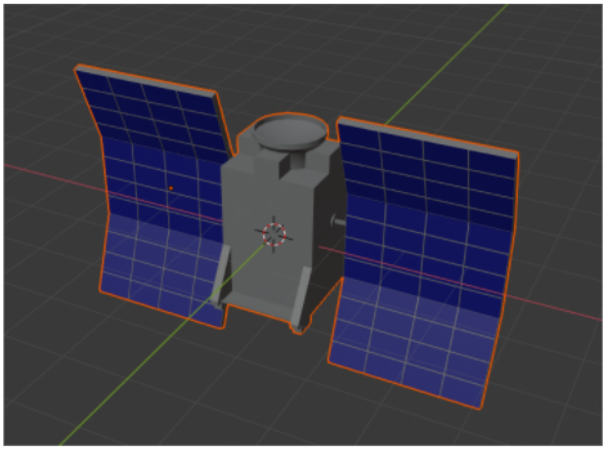
Scene Collection | Verts:1,212 | Faces:1,513 | Tris:2,332 | Objects:0/1 | Mem: 28.4 MiB

Модель Спутник G03



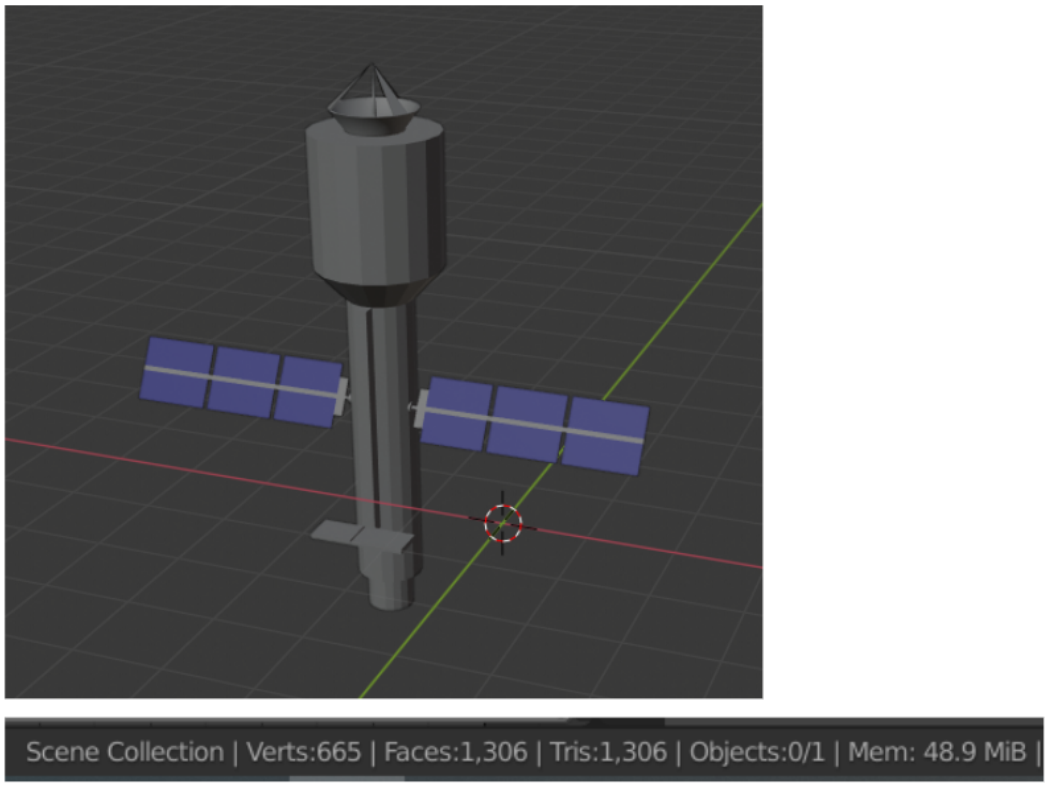
Scene Collection | Verts:516 | Faces:837 | Tris:1,008 | Objects:1/1 | Mem: 29.5 MiB

Модель Спутник G04

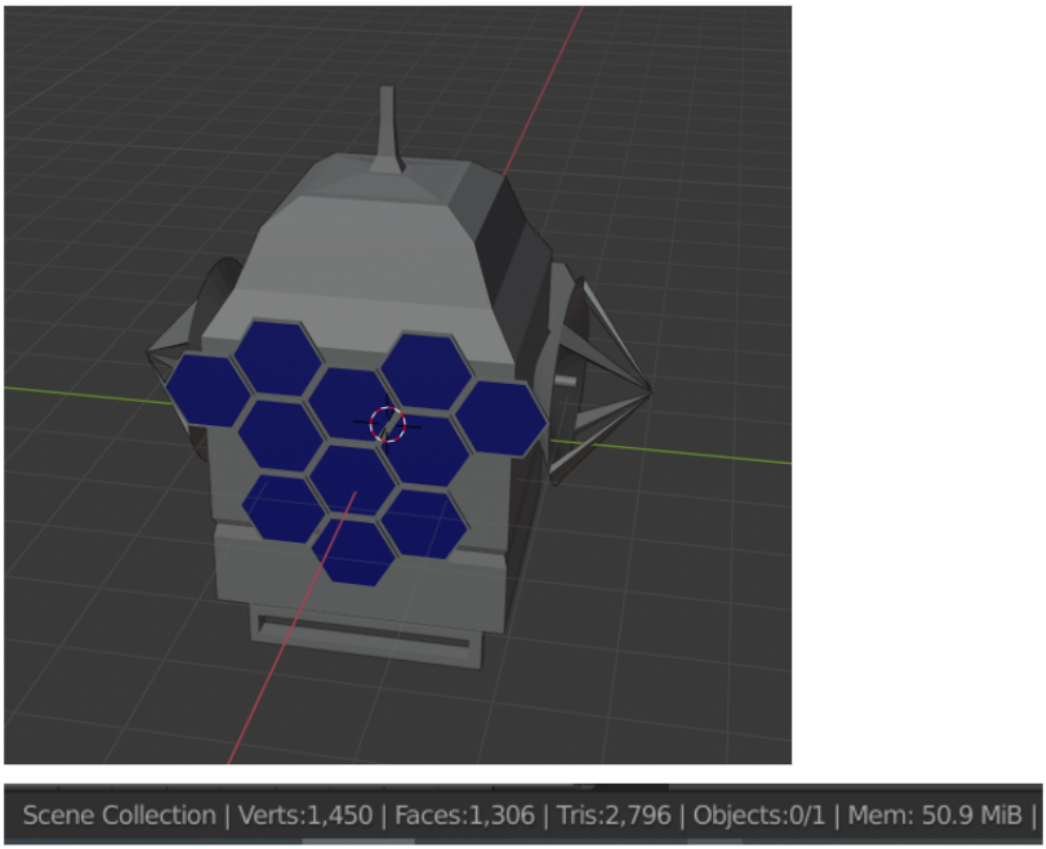


Scene Collection | Verts:2,727 | Faces:5,194 | Tris:5,404 | Objects:1/1 | Mem: 32.2 MiB |

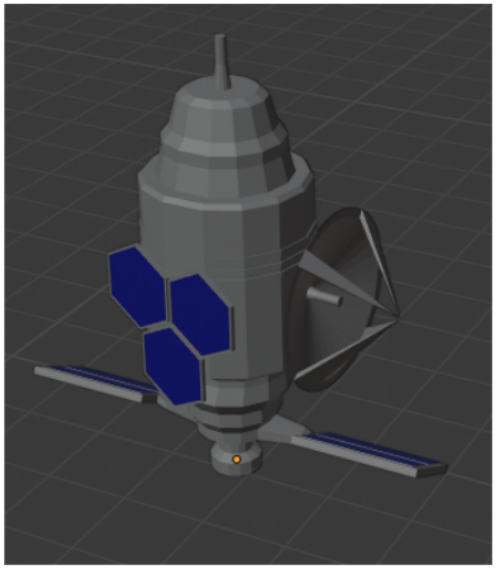
Модель Спутник G05



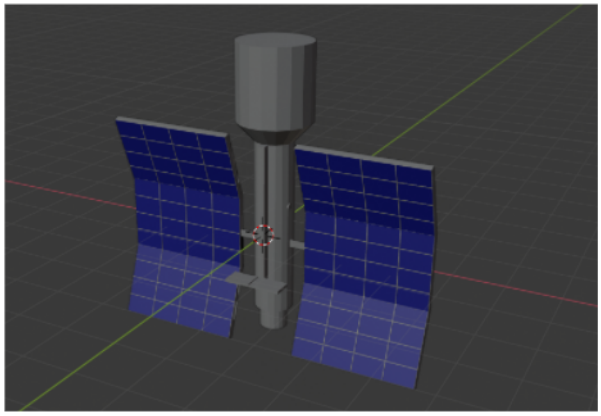
Модель Спутник G06



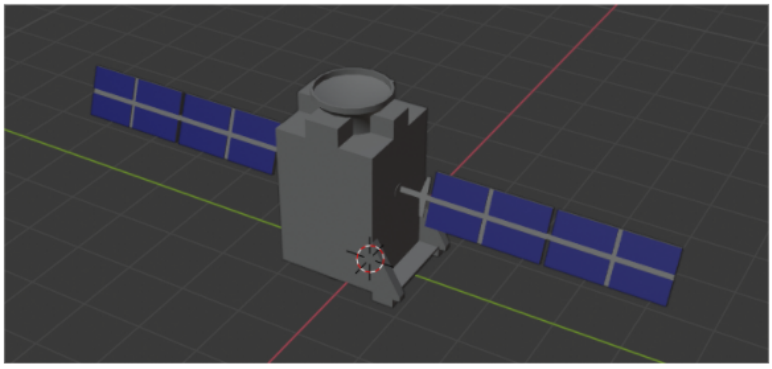
Модель Спутник G07

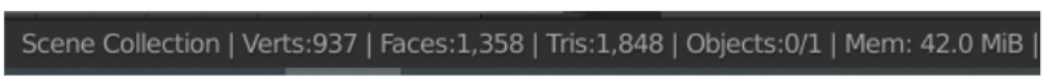


Модель Спутник G08

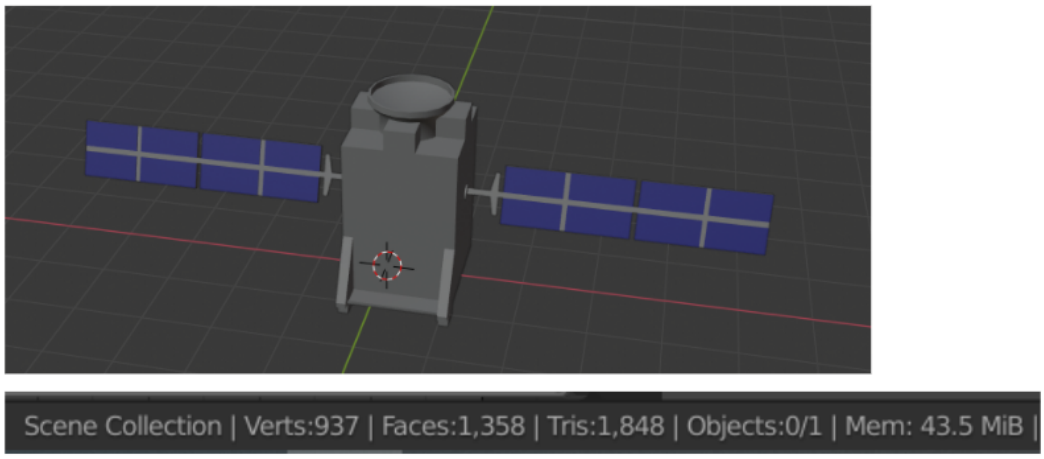


Модель Спутник G09

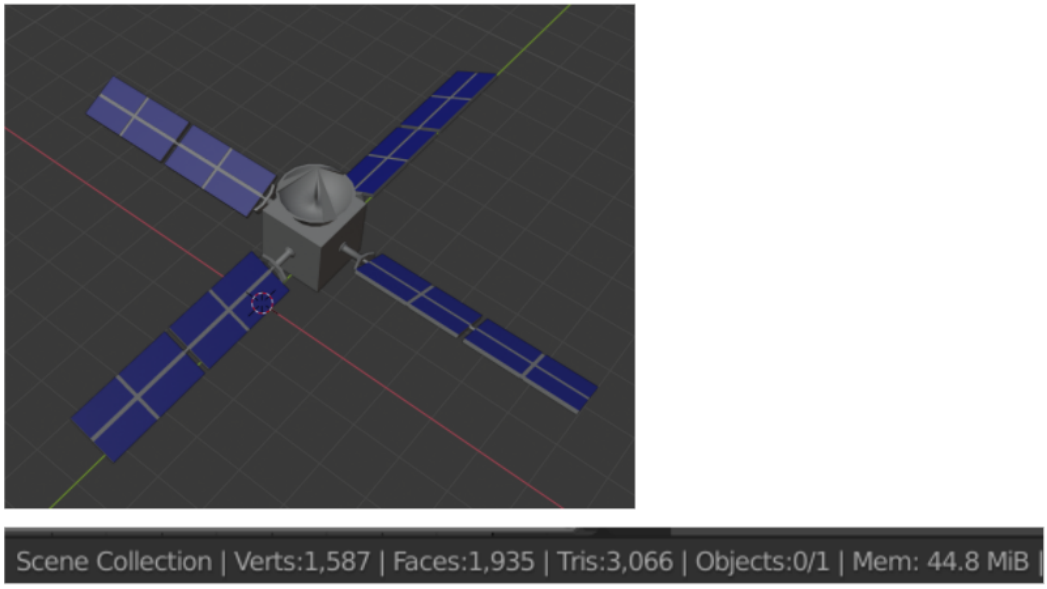




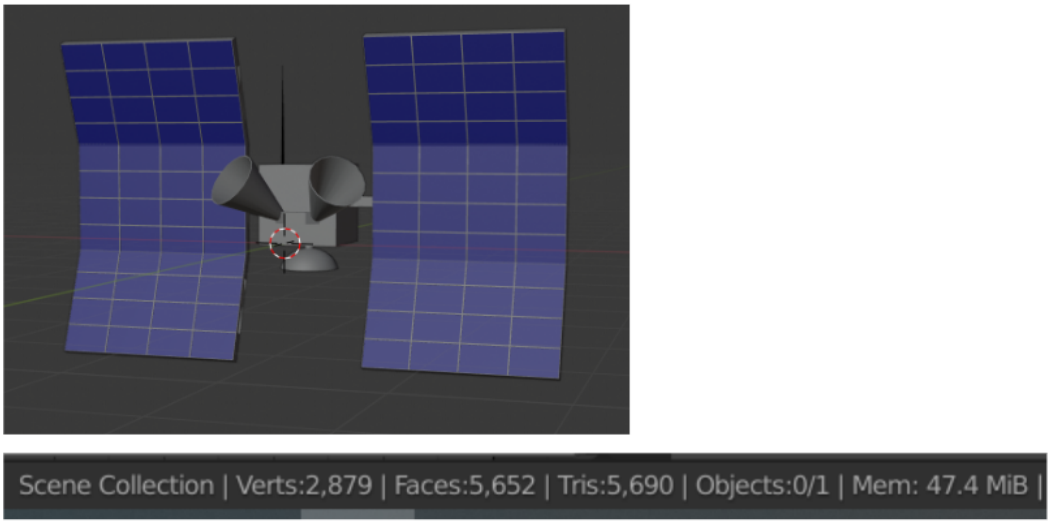
Модель Спутник G10



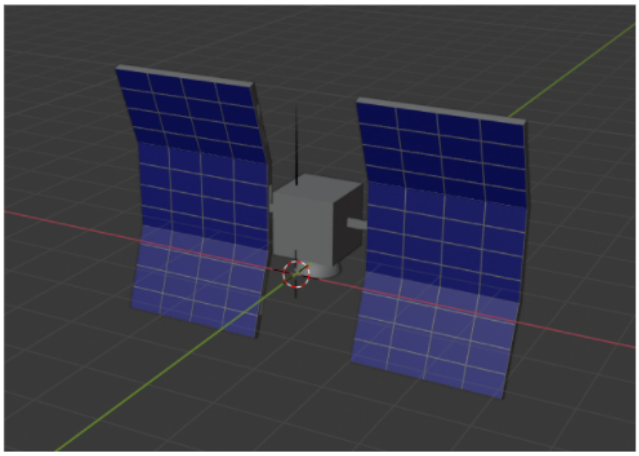
Модель Спутник G11



Модель Спутник G12

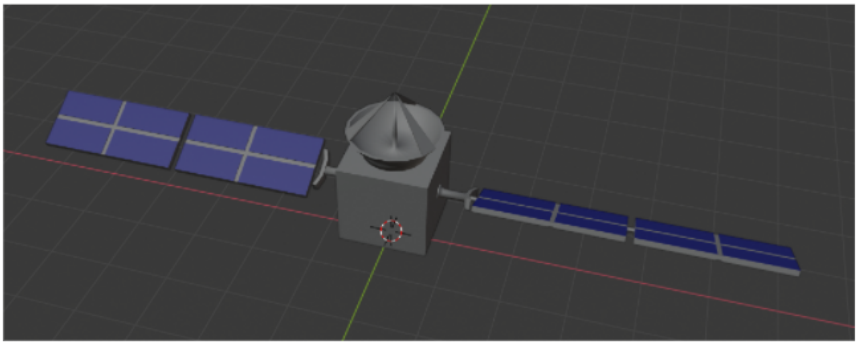


Модель Спутник G13



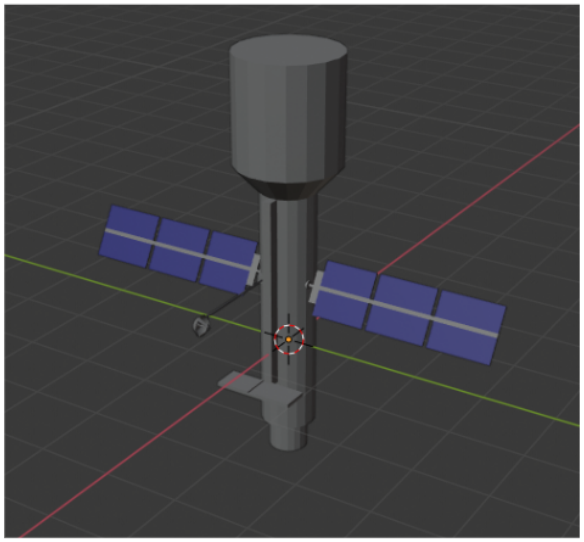
Scene Collection | Verts:2,601 | Faces:5,136 | Tris:5,154 | Objects:0/1 | Mem: 49.0 MiB |

Модель Спутник G14



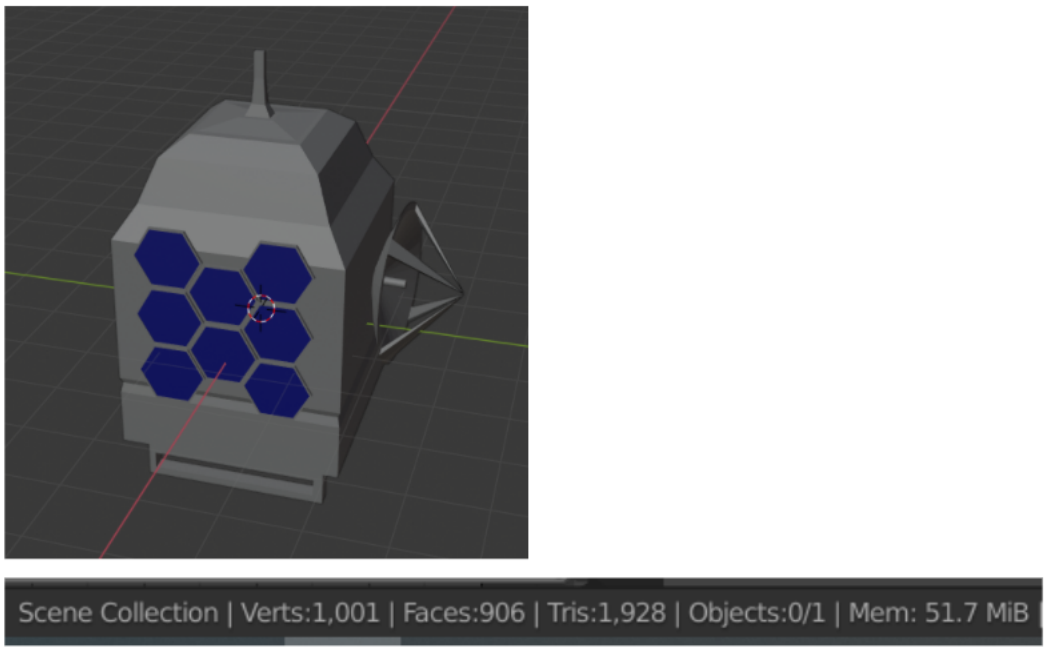
Scene Collection | Verts:889 | Faces:1,047 | Tris:1,722 | Objects:0/2 | Mem: 35.1 MiB |

Модель Спутник G15

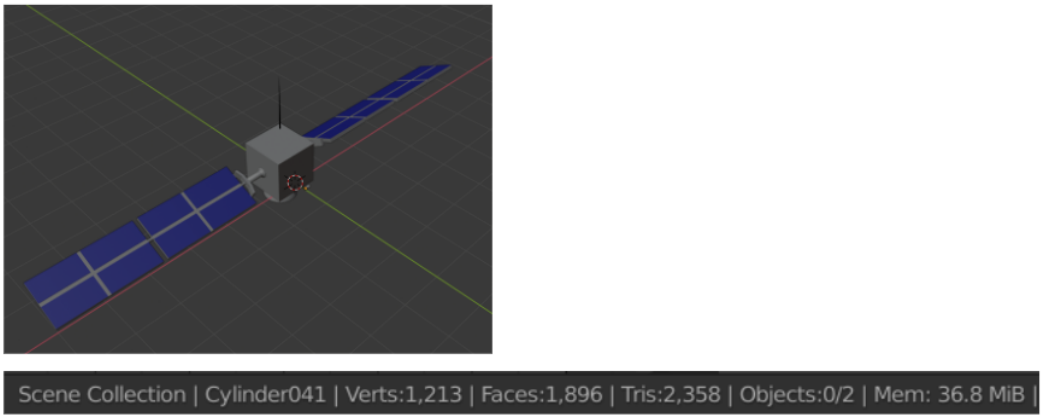


Scene Collection | Cylinder001.004 | Verts:727 | Faces:1,421 | Tris:1,421 | Objects:0/1 | Mem: 50.0 MiB |

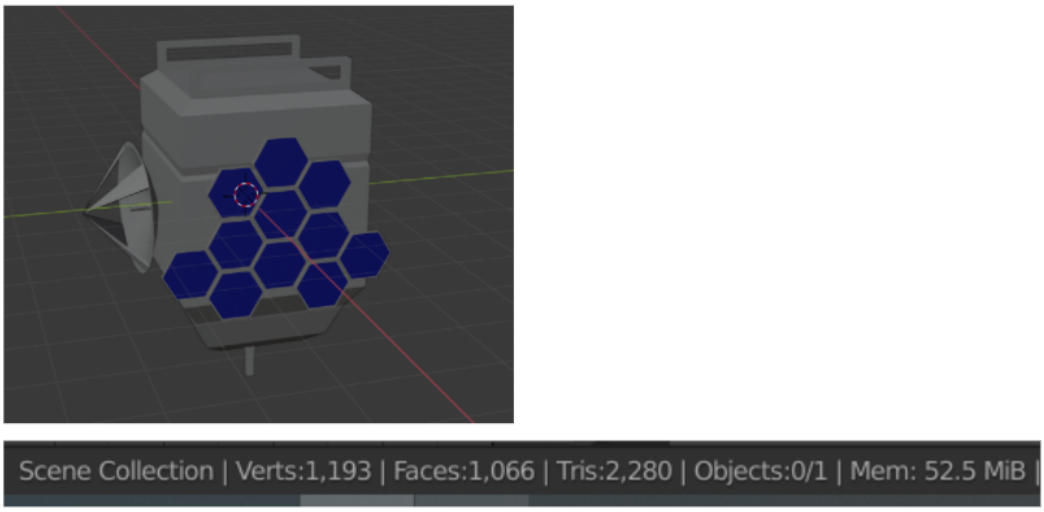
Модель Спутник G16



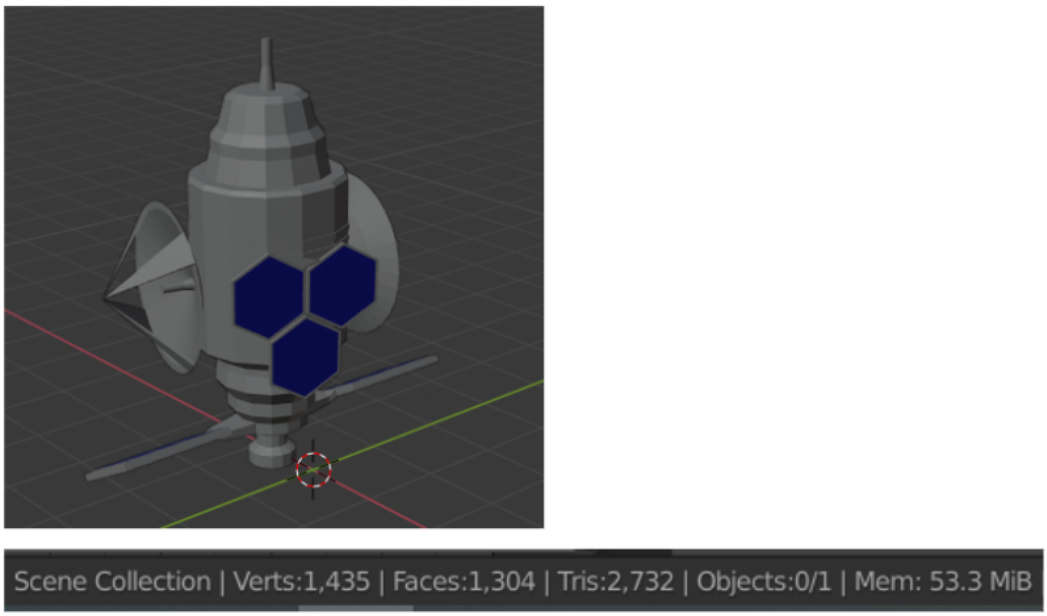
Модель Спутник G17



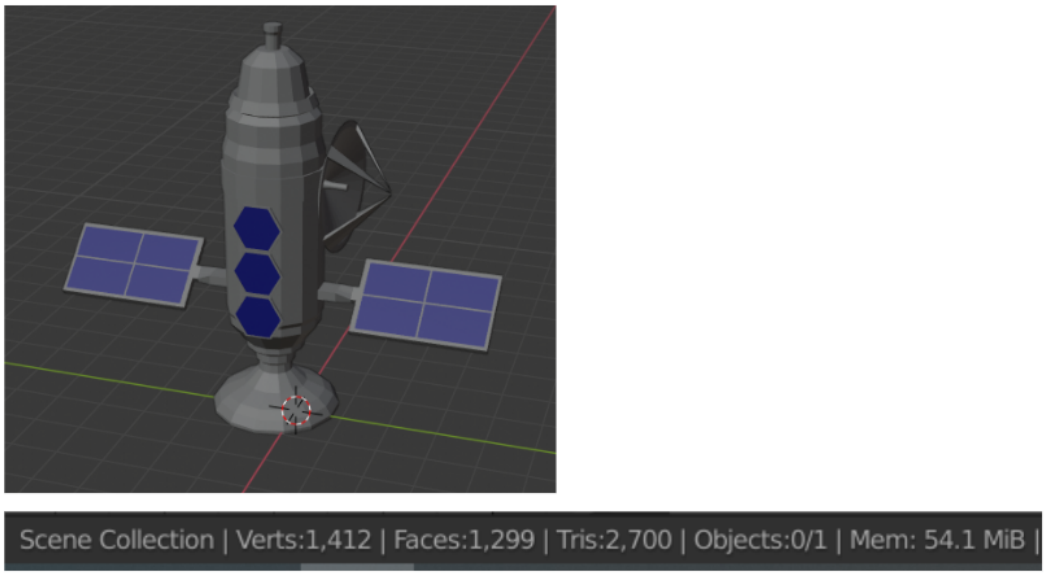
Модель Спутник G18



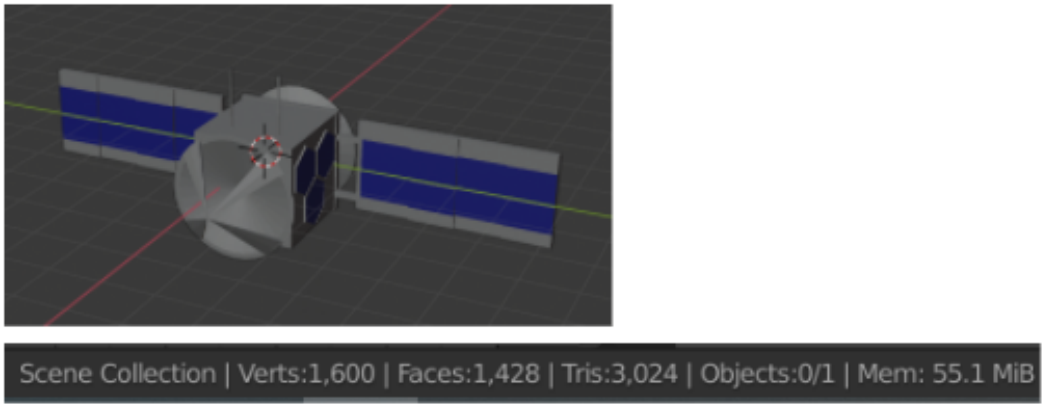
Модель Спутник G19



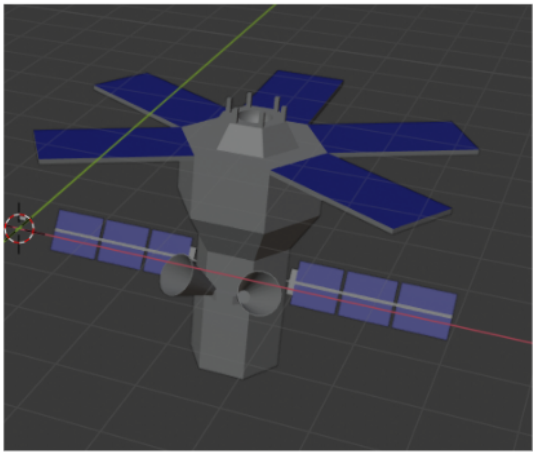
Модель Спутник G20

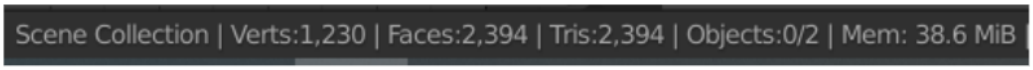


Модель Спутник G21

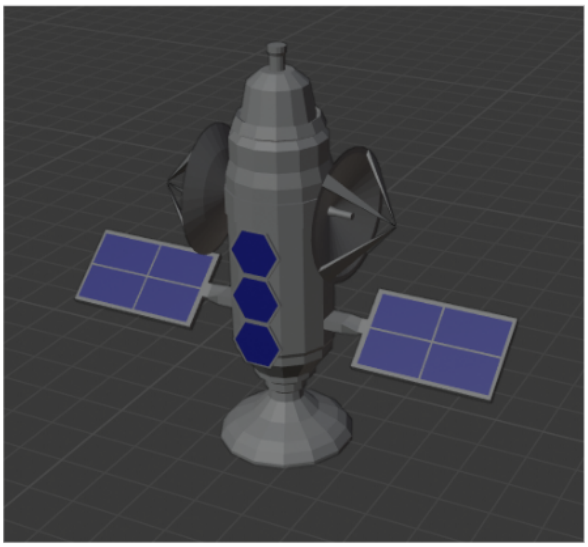


Модель Спутник G22

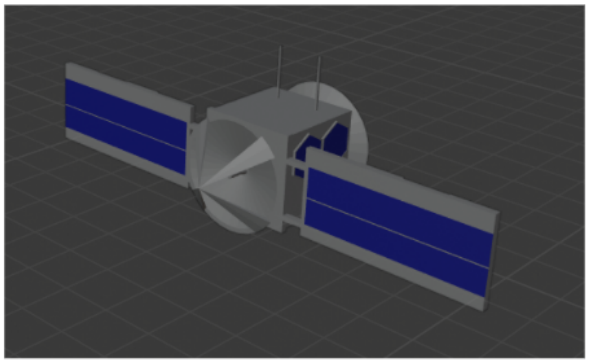


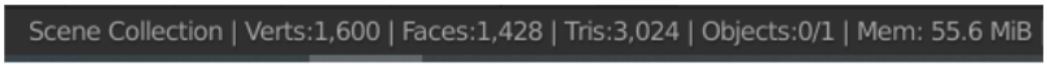


Модель Спутник G23

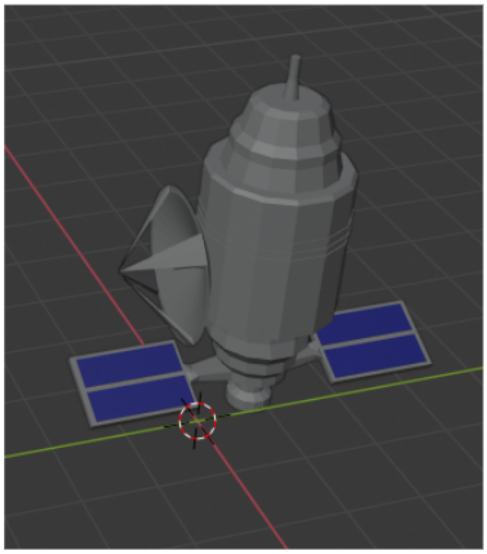


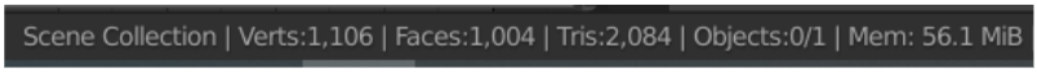
Модель Спутник G24



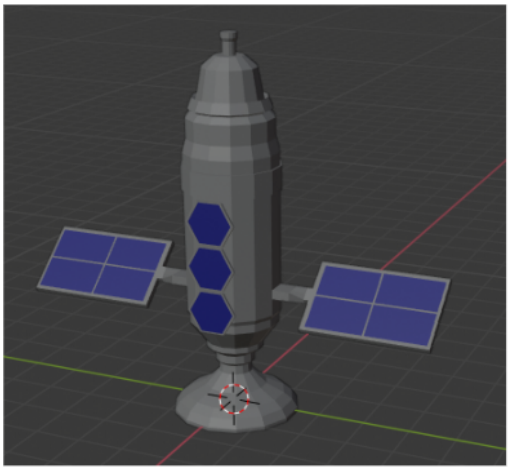


Модель Спутник G25

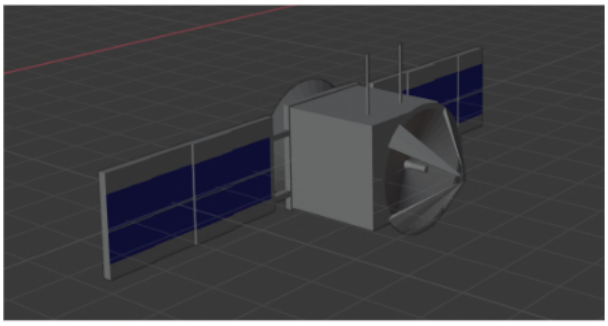


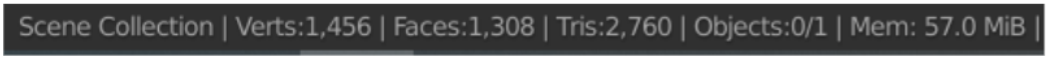


Модель Спутник G26

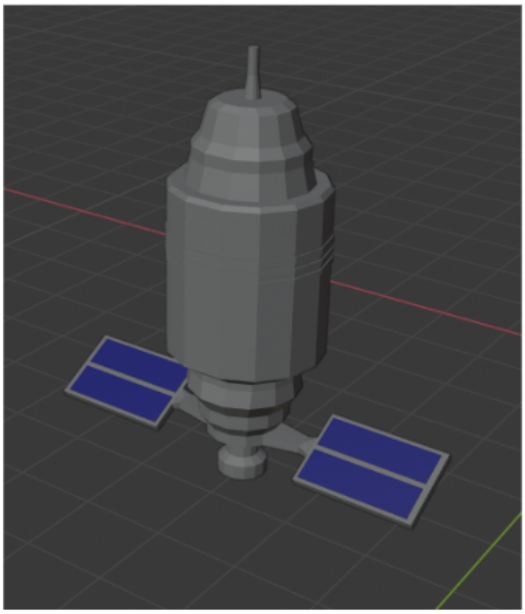


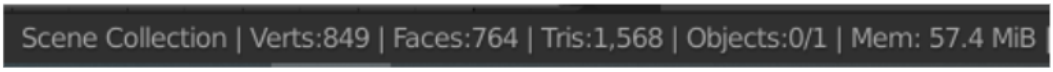
Модель Спутник G27



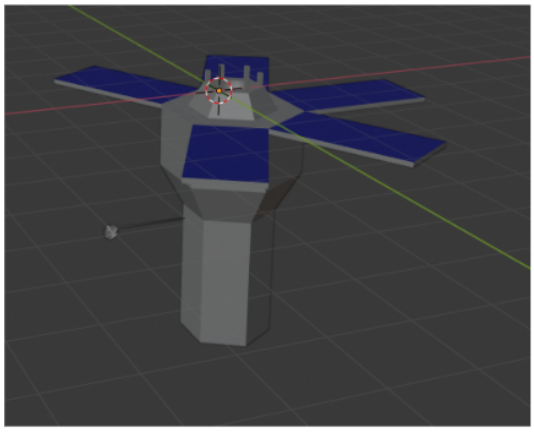


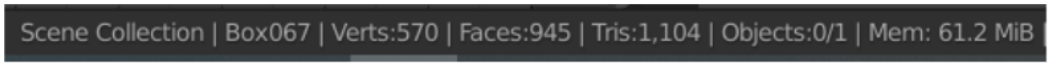
Модель Спутник G28



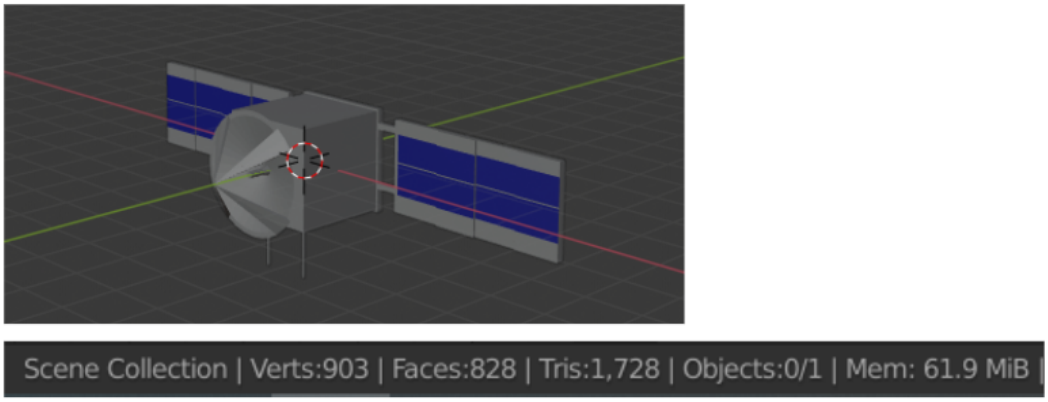


Модель Спутник G29

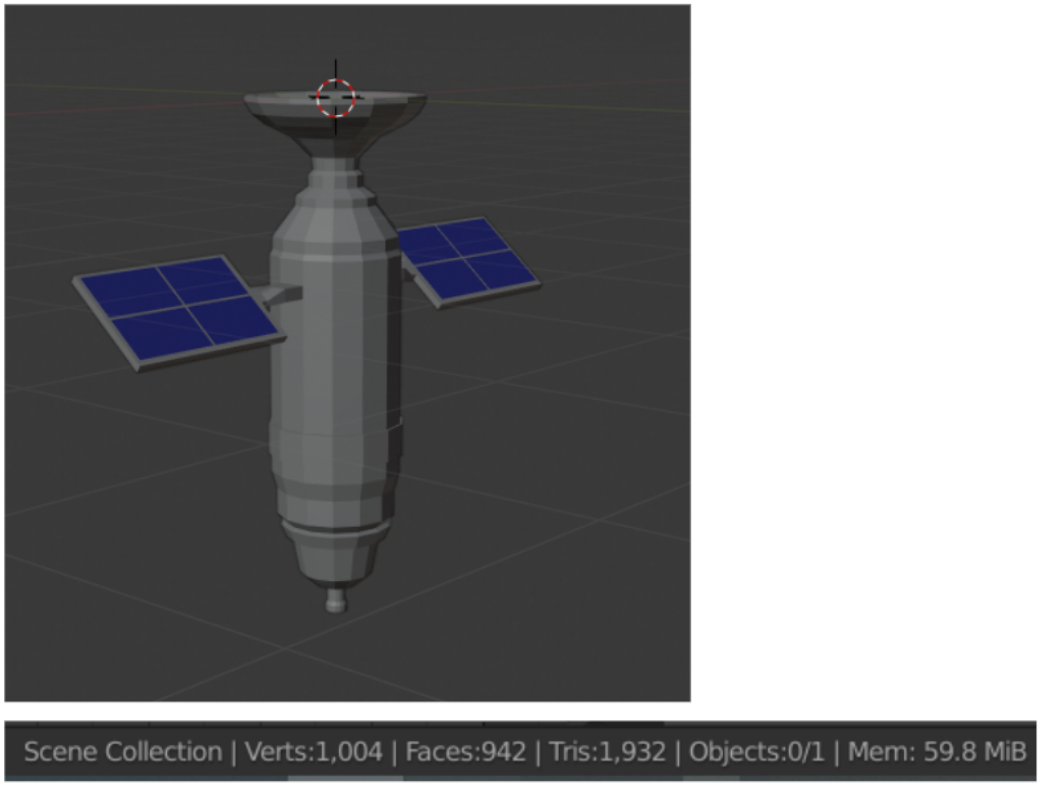




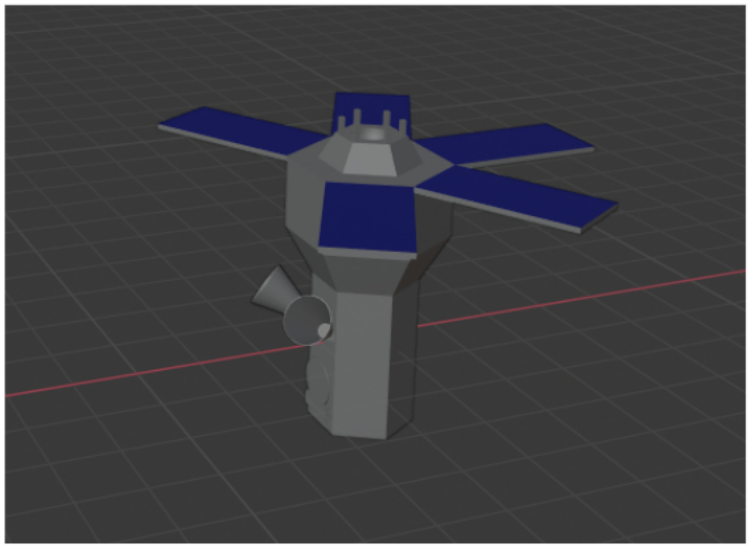
Модель Спутник G30



Модель Спутник G31

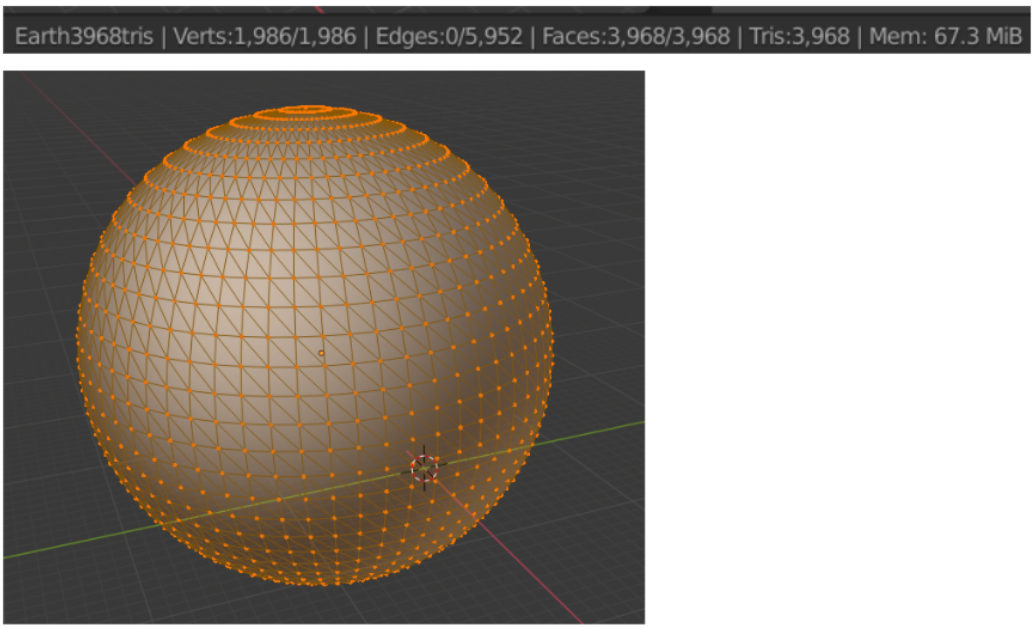


Модель Спутник G32

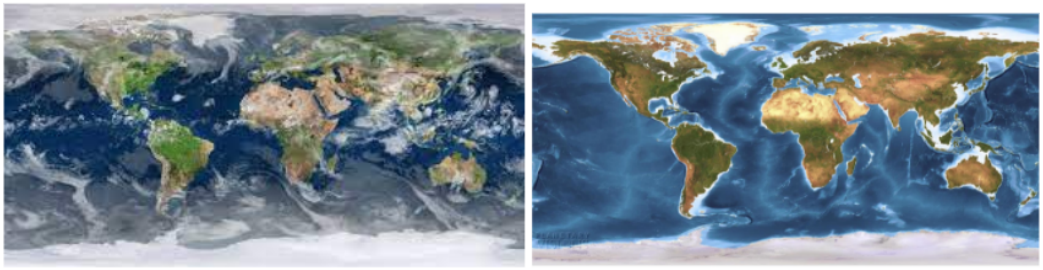


Конструирование 3D-модель земли

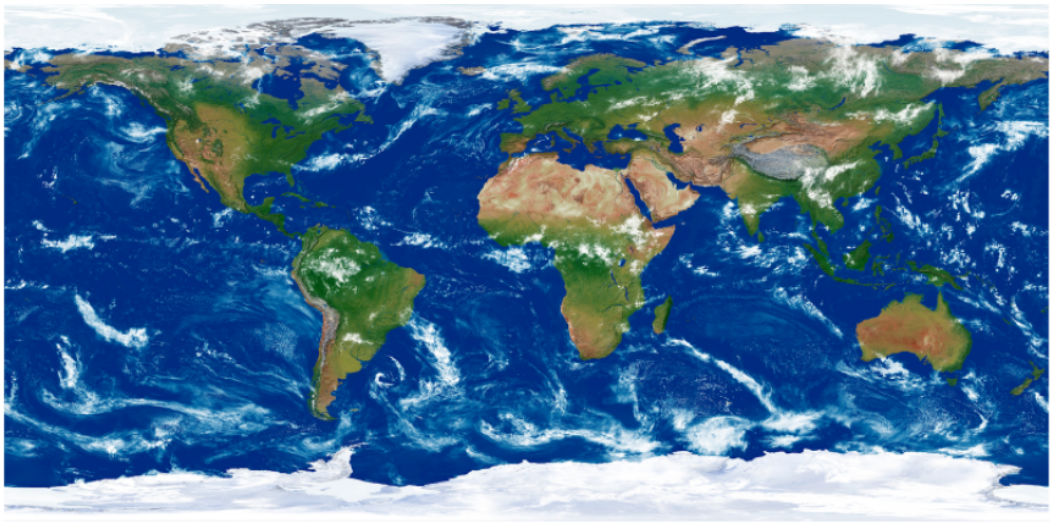
Для модели планеты Земля была выбрана разработка модели идеального шара.



Для создания текстуры было подобрано два референсных изображения, которые представлены ниже.



С помощью графического редактора из референсных изображений была создана итоговая текстура для Земли, с разрешением 2048 на 1024 пикселей для качественной детализации планеты.

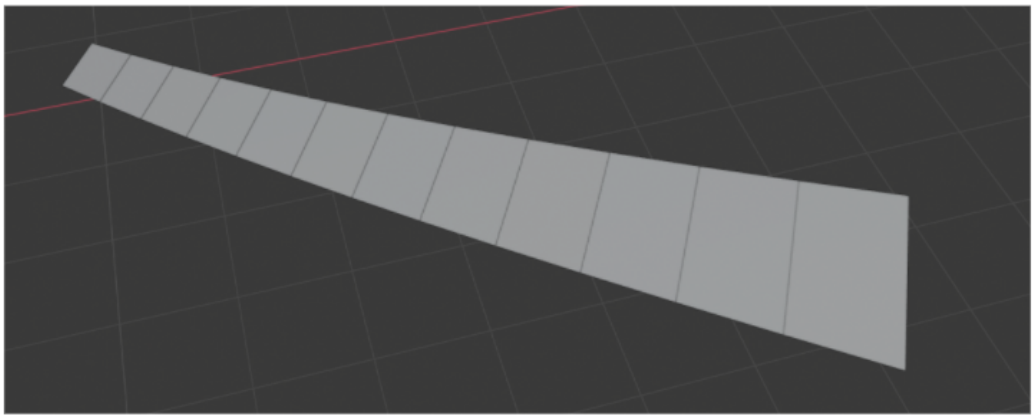


Конструирование 3D-модели атмосферы

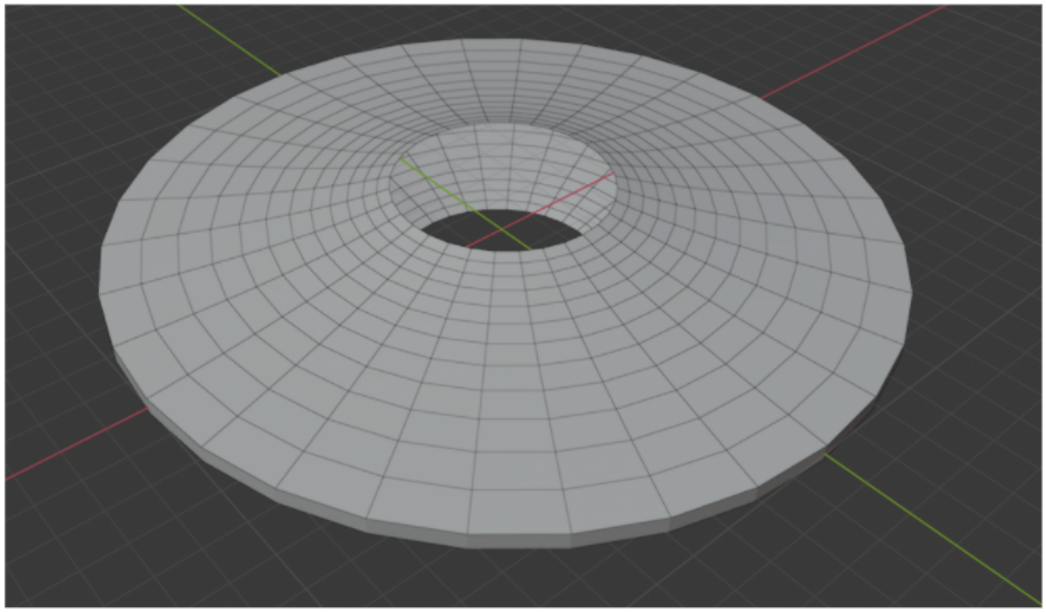
Для отображения ионосферы в будущем приложении было разработано три варианта.

Первый — одна 3D-модель кусочка ионосферы трапециевидной формы для дальнейшей генерации целого круга данных.

Второй — 3D-модель сектора ионосферы, разбитого на уровни



Первый — отображение всего среза ионосферы

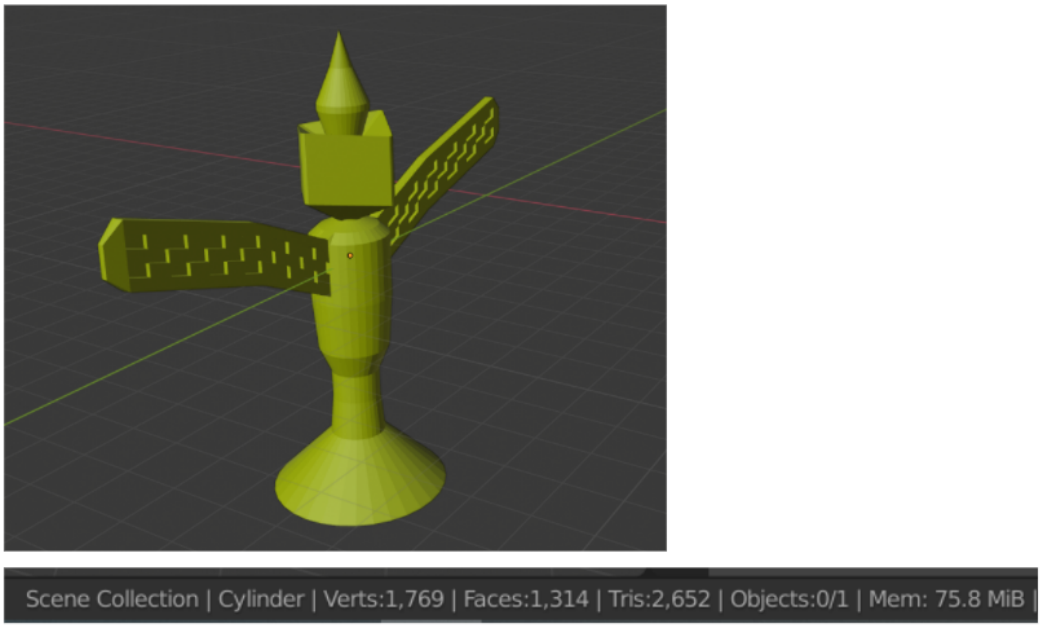


Разработка стилизованных 3D-моделей

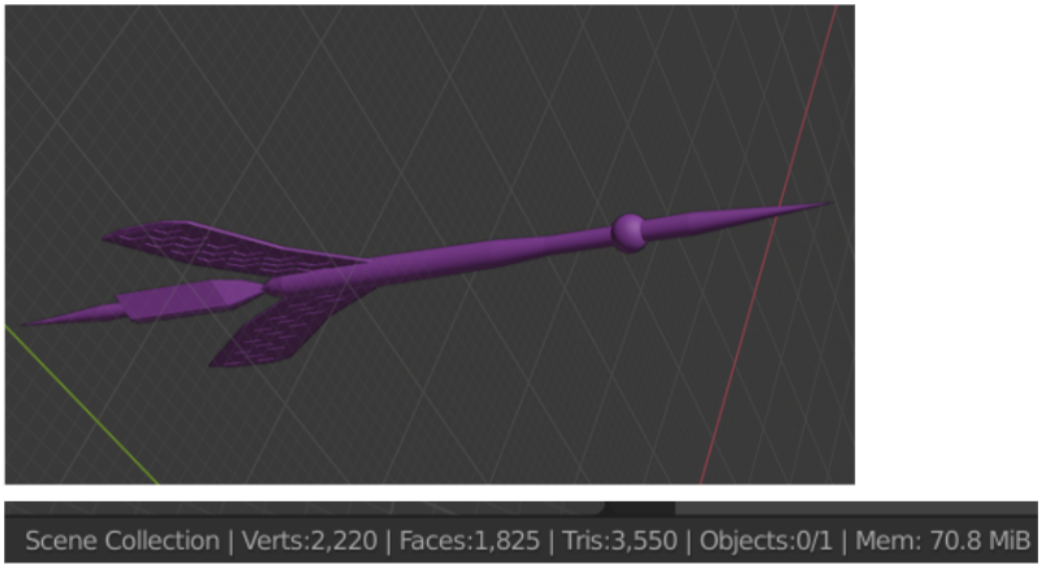
Для стилизации приложения была выбрана тематика компьютерной игры Cyberpunk 2077.

Так же, как и в моделях основной стилизаций, было изначально сконструировано несколько основных моделей и дополнительные детали, а потом применена модификация спутников. Также, было выбрано три цвета для текстур.

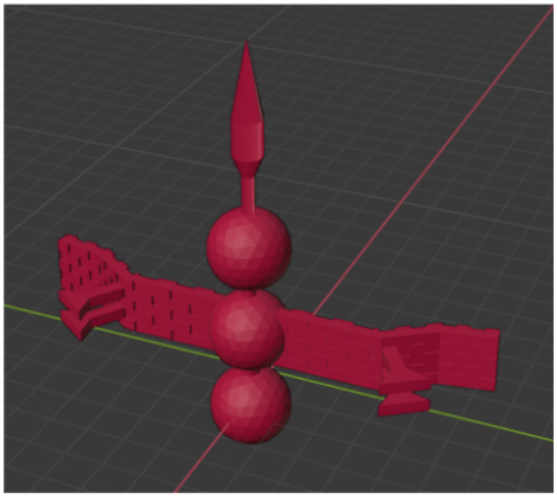
Модель Спутник G01



Модель Спутник G02

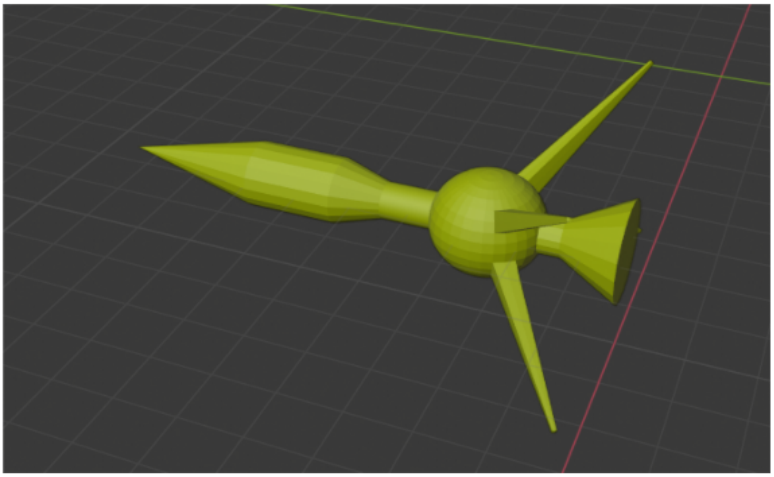


Модель Спутник G03



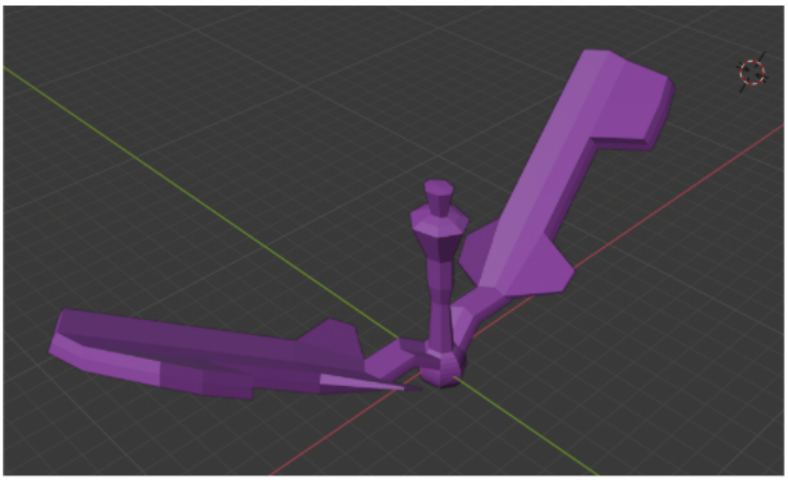
Scene Collection | Verts:2,375 | Faces:3,342 | Tris:3,342 | Objects:0/1 | Mem: 74.2 MiB |

Модель Спутник G04



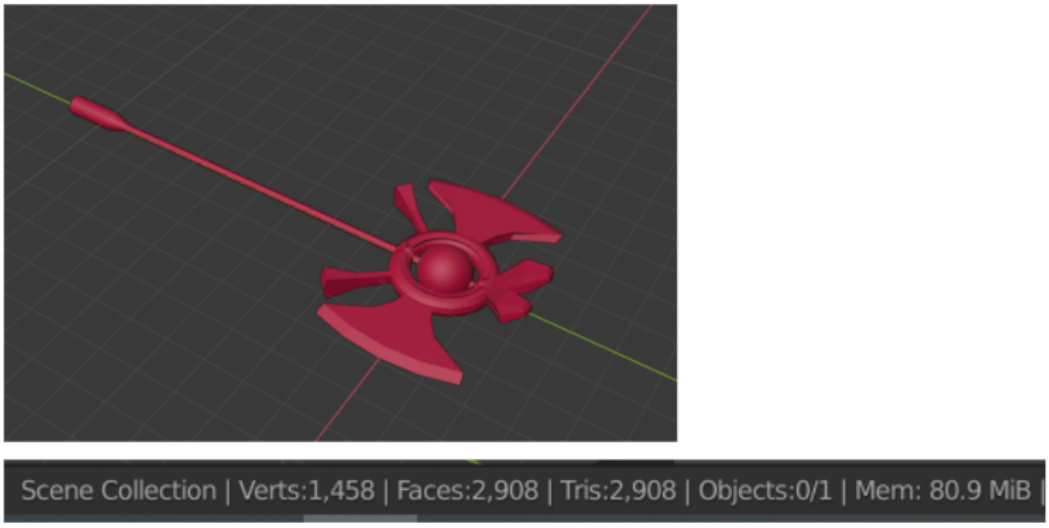
Scene Collection | Verts:695 | Faces:1,382 | Tris:1,382 | Objects:0/1 | Mem: 76.6 MiB

Модель Спутник G05

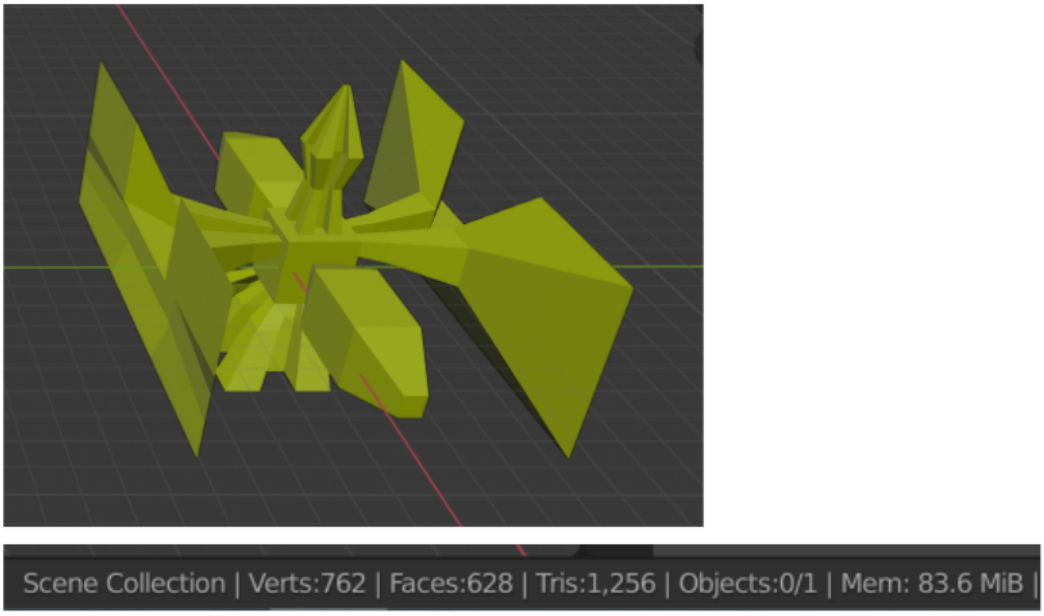


Scene Collection | Verts:311 | Faces:296 | Tris:584 | Objects:0/1 | Mem: 77.4 MiB |

Модель Спутник G06



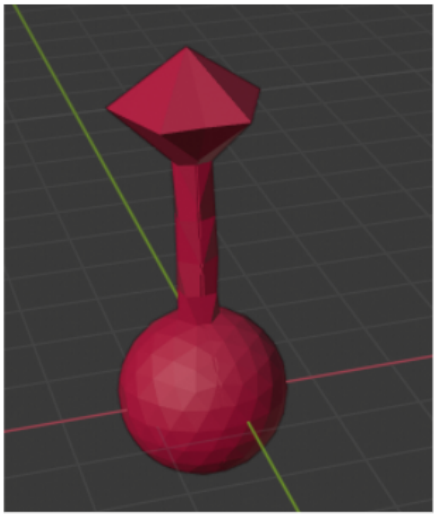
Модель Спутник G07



Модель Спутник G08

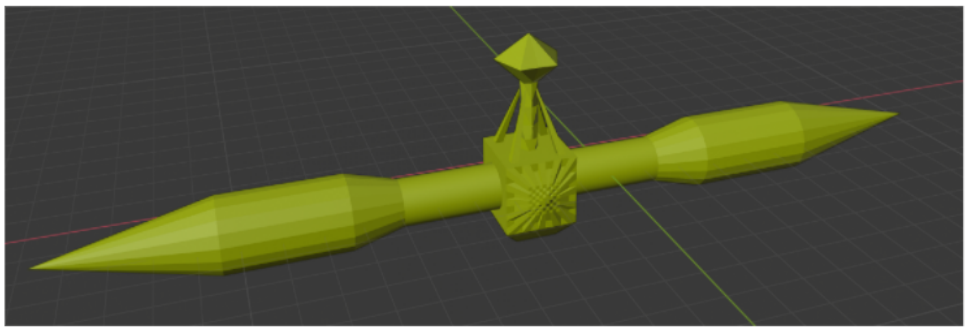
Scene Collection | Verts:740 | Faces:708 | Tris:708 | Objects:0/1 | Mem: 86.2 MiB |

Модель Спутник G09



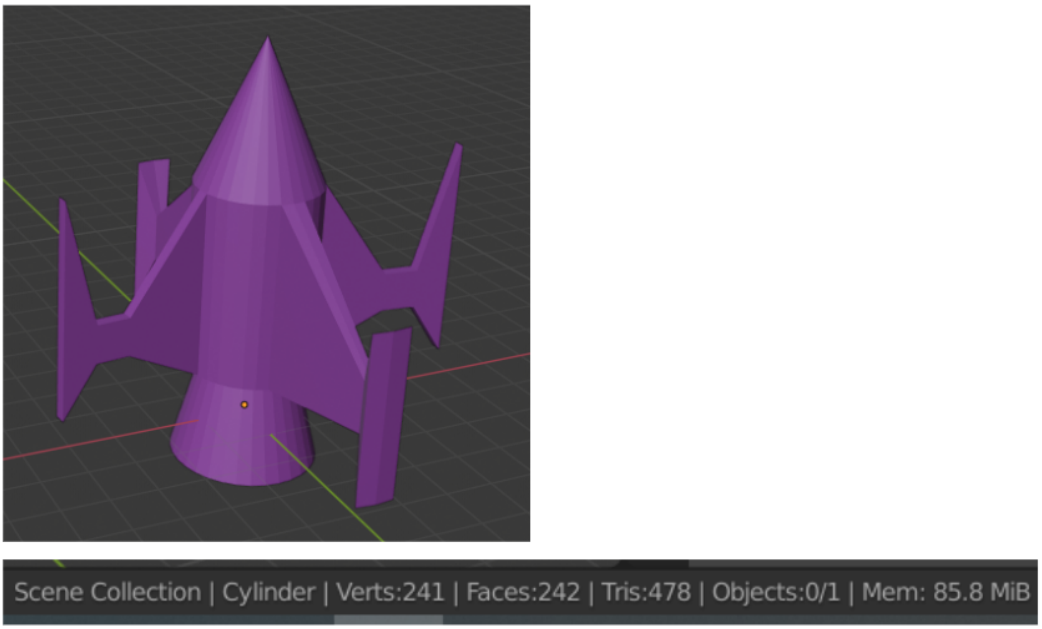
Scene Collection | Verts:312 | Faces:496 | Tris:496 | Objects:0/1 | Mem: 86.0 MiB |

Модель Спутник G10

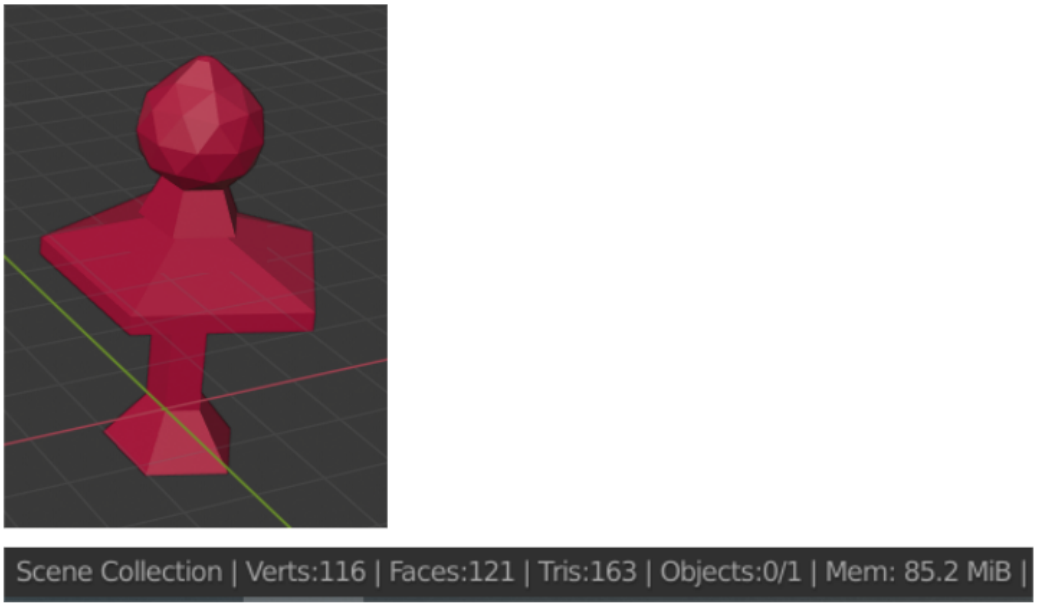


Scene Collection | Verts:1,710 | Faces:2,508 | Tris:2,508 | Objects:0/1 | Mem: 88.9 MiB |

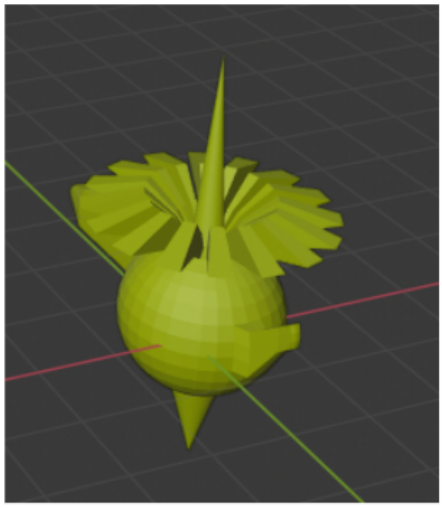
Модель Спутник G11



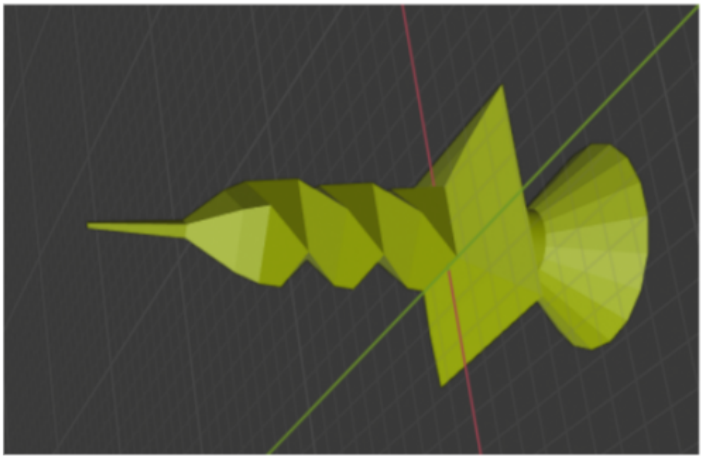
Модель Спутник G12



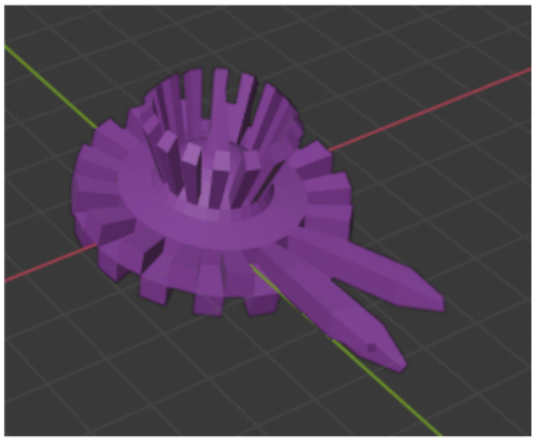
Модель Спутник G13

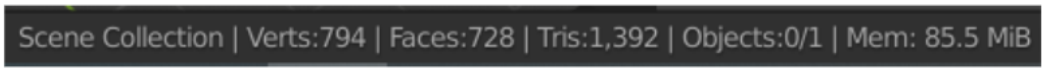


Модель Спутник G14

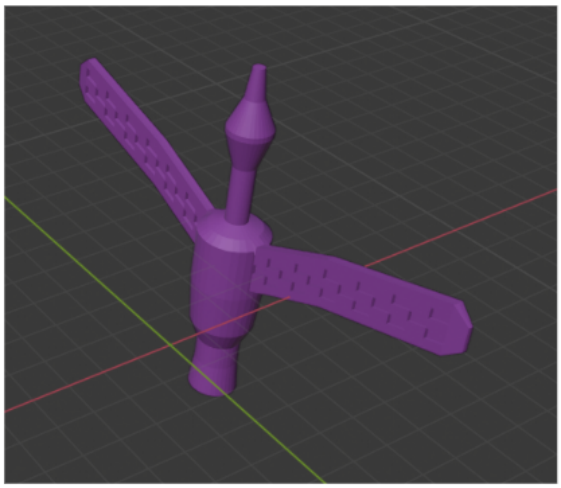


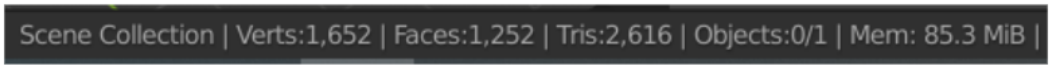
Модель Спутник G15



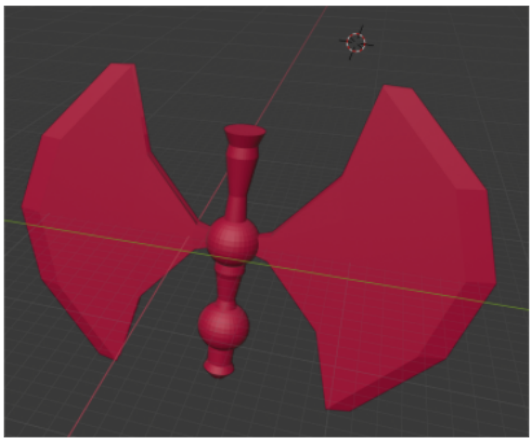


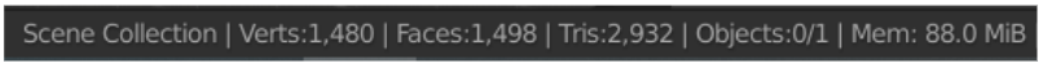
Модель Спутник G16



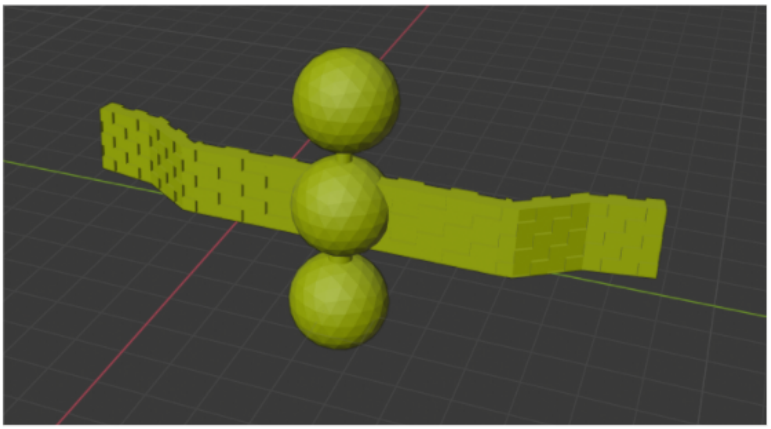


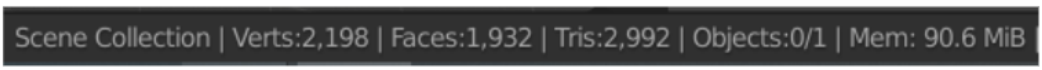
Модель Спутник G17



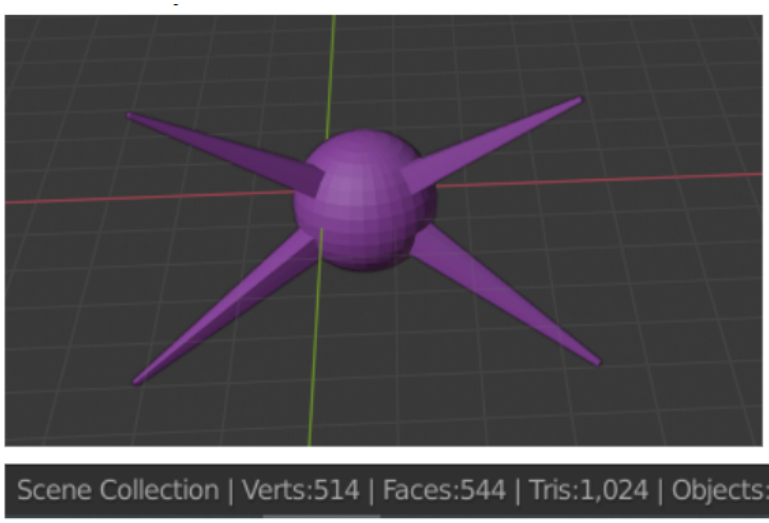


Модель Спутник G18

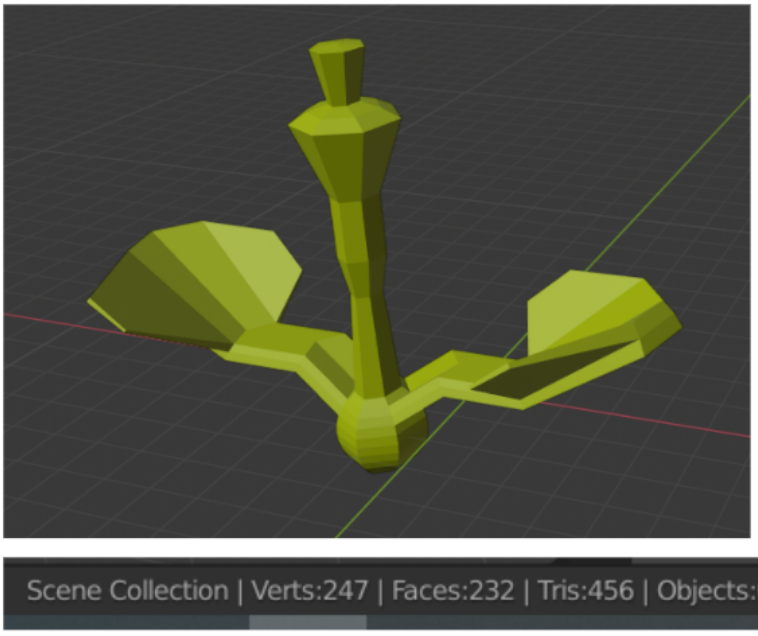




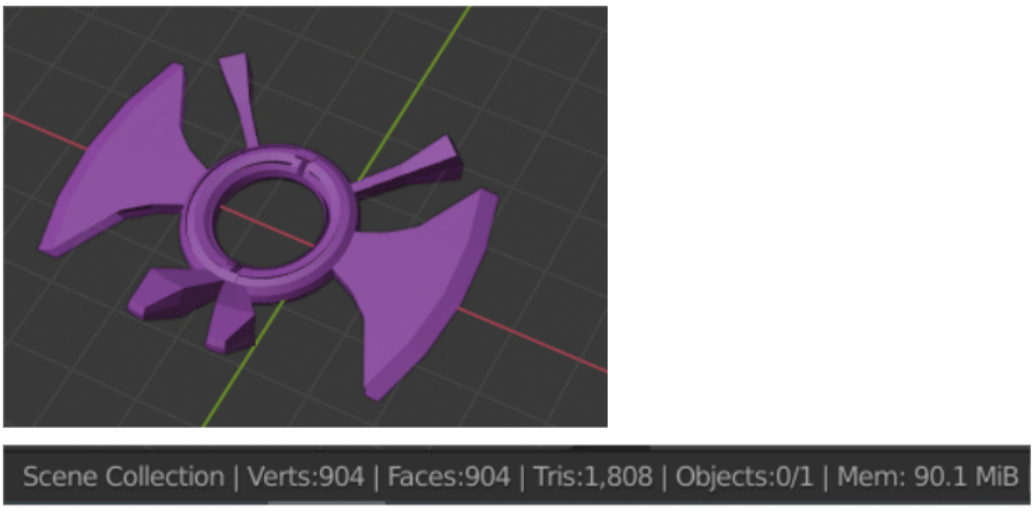
Модель Спутник G19



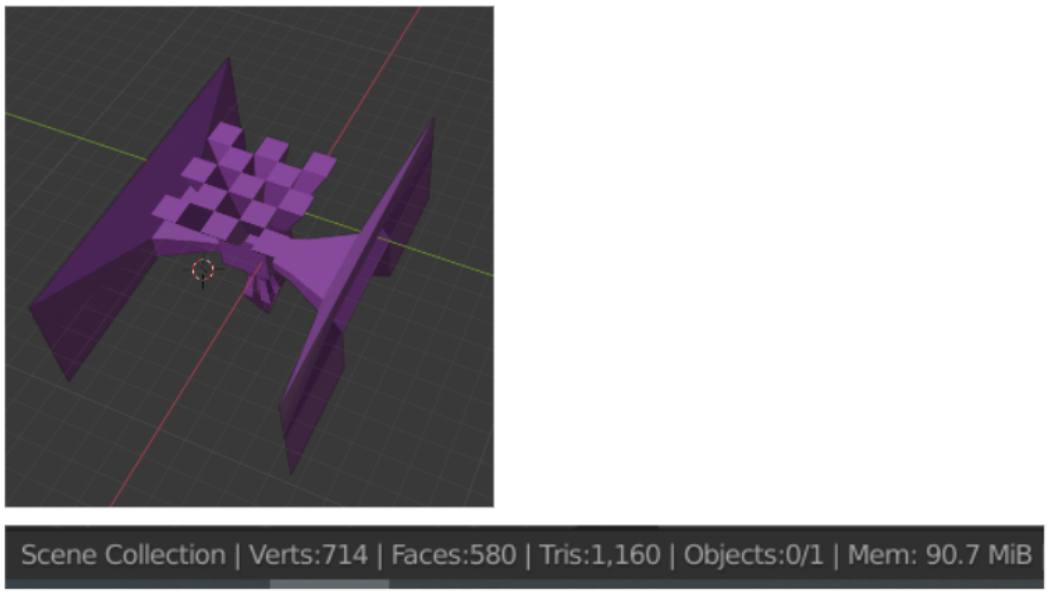
Модель Спутник G20



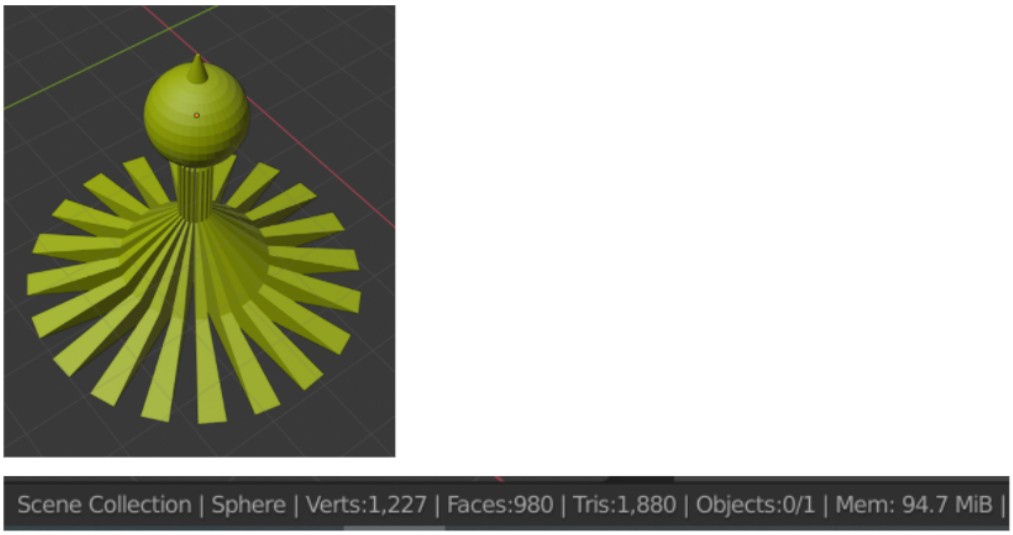
Модель Спутник G21



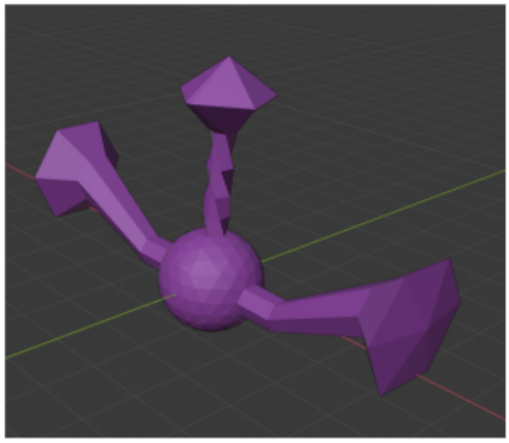
Модель Спутник G22



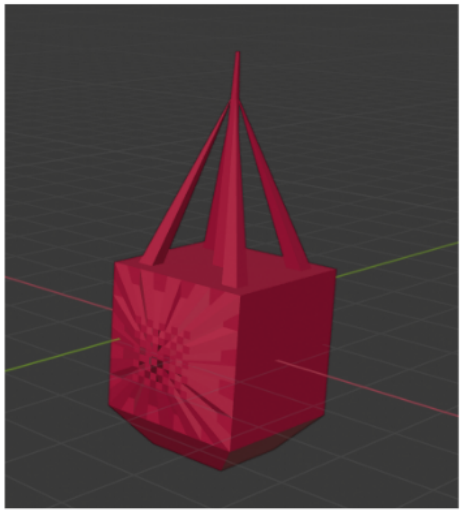
Модель Спутник G23



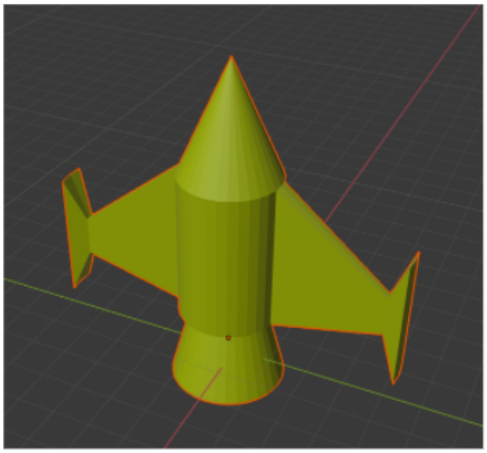
Модель Спутник G24



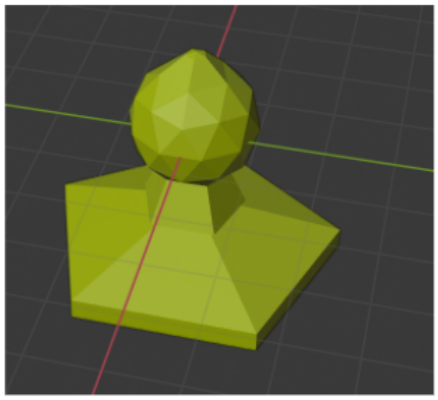
Модель Спутник G25



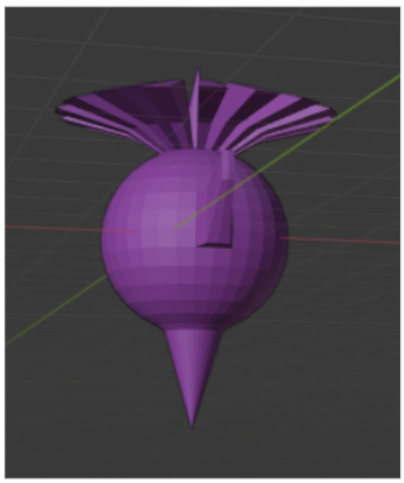
Модель Спутник G26

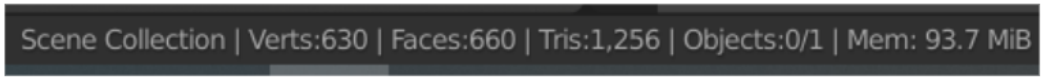


Модель Спутник G27

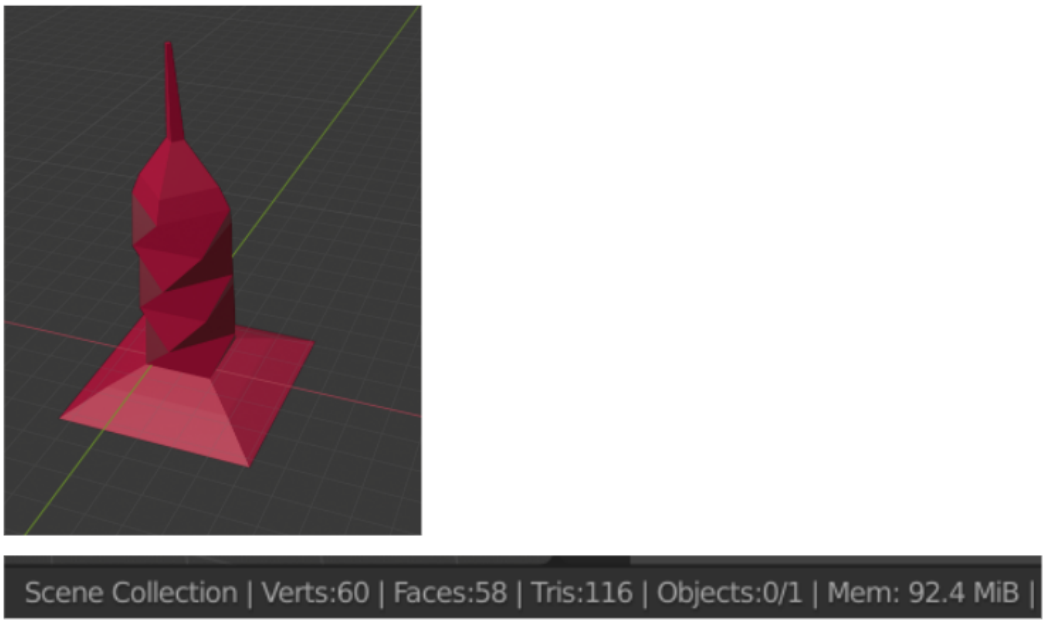


Модель Спутник G28

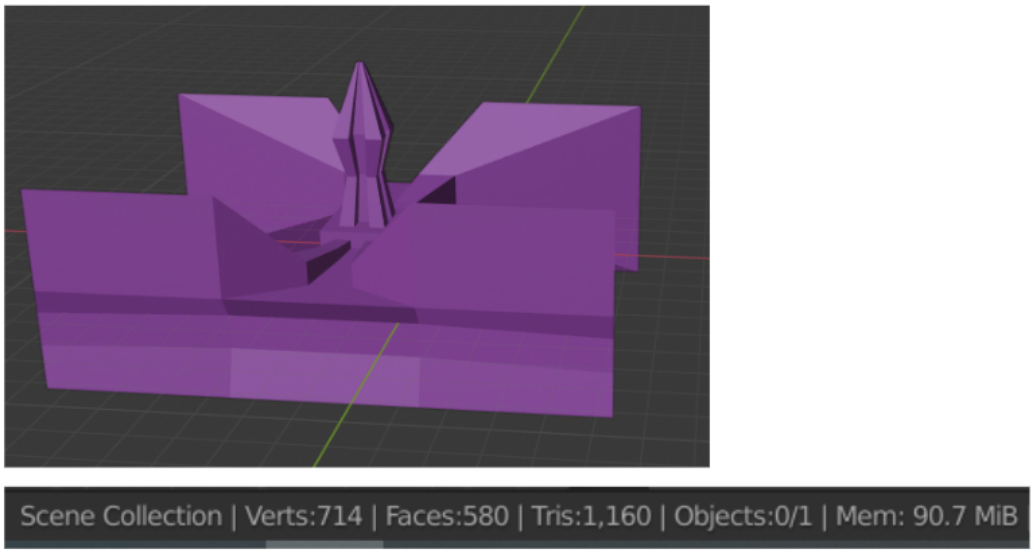




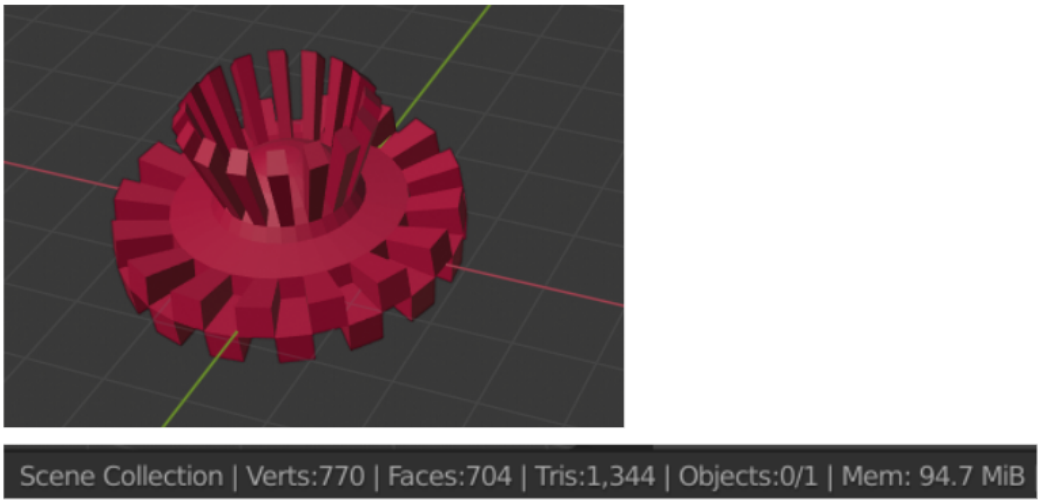
Модель Спутник G29



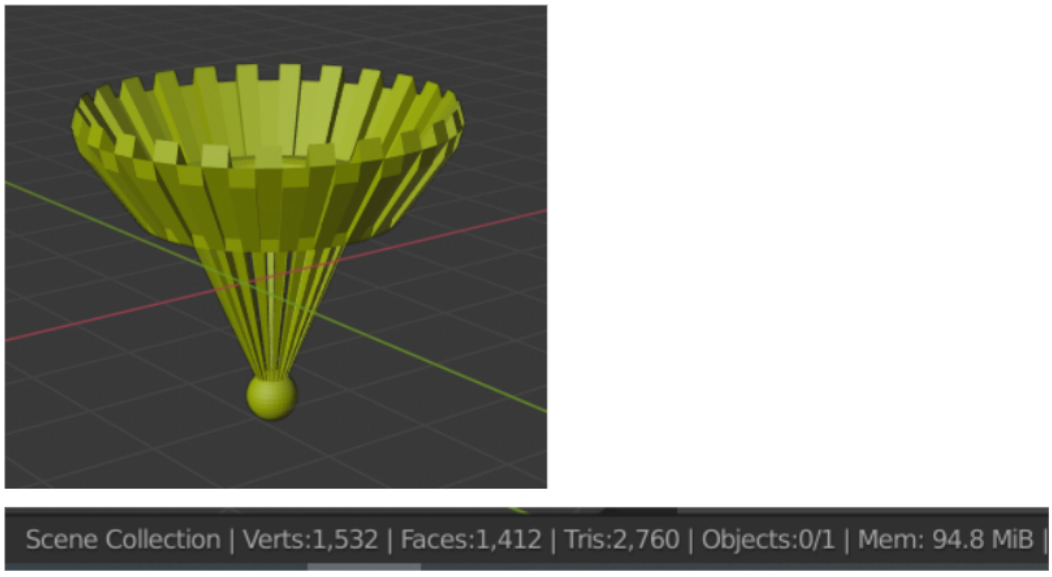
Модель Спутник G30



Модель Спутник G31



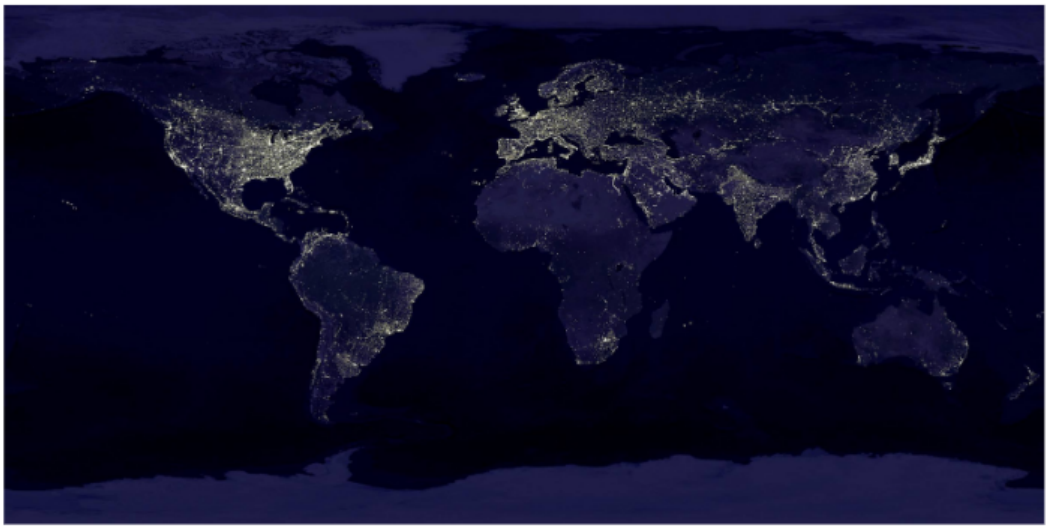
Модель Спутник G32



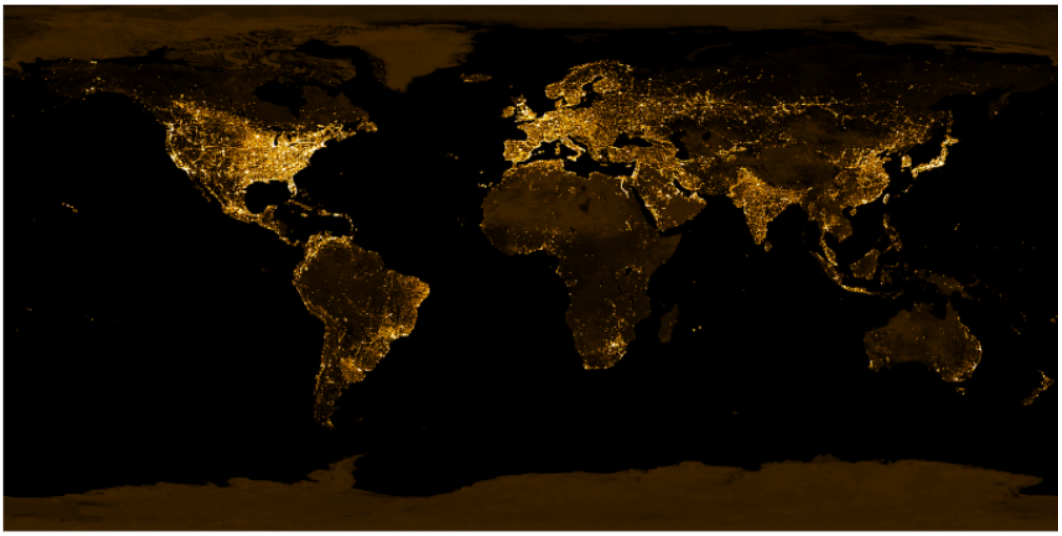
Кроме моделей спутников была стилизована модель Земли. Форма объекта не менялась и перерабатывалась только текстура планеты.

В этот раз для референса было взято ночное изображение земли и применена цветокоррекция для соответствия выбранной тематике.

Референс:



Итоговая текстура

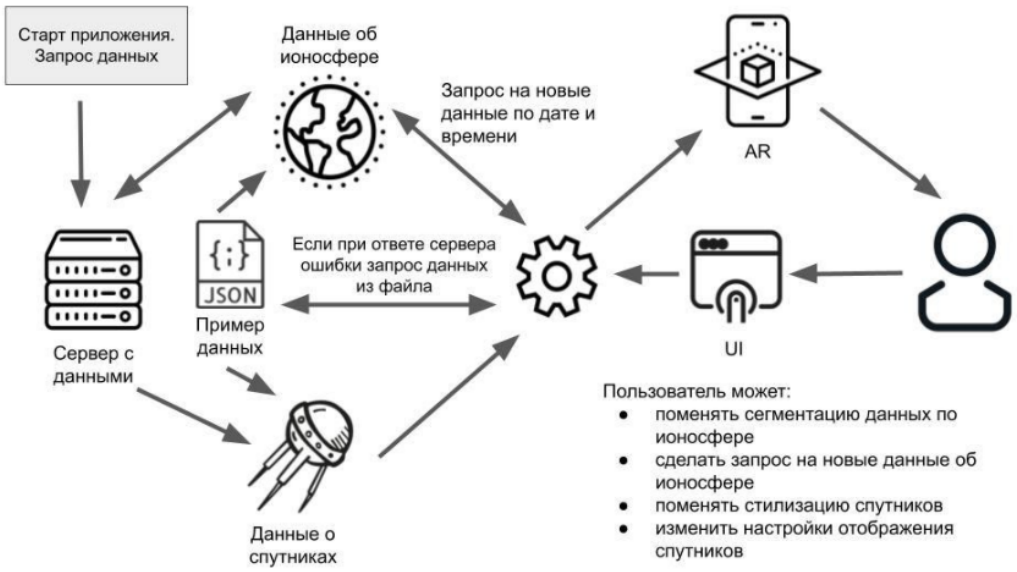


Текстура идентична размерам текстуры основной стилизации.

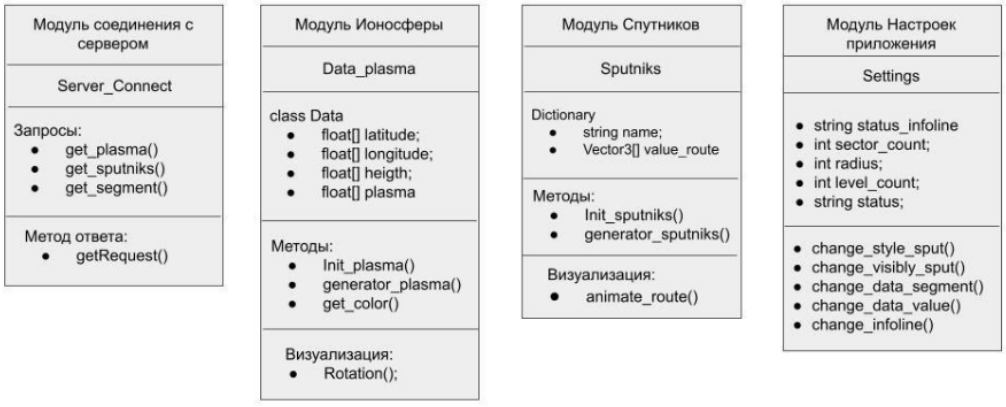
Задачи AR-разработчика (архитектора AR-приложения)

AR-разработчик проектирует архитектуру приложения, выбирает тип дополненной реальности, определяет цветовую схему, разрабатывает вайфрейм (концепцию создания структуры дизайна интерфейса), стилизует кнопки и иконки по все функциональные возможности будущего AR-приложения.

Проектирование архитектуры приложения и его функциональных возможностей



Модель данных AR-приложения



На картинке выше представлено взаимодействие всех модулей будущего AR-приложения и их основные свойства и функции.

Проектирование дизайна AR-приложения

Дизайн приложения можно создать в любом из графических редакторов или воспользоваться специальным инструментарием, например, Figma.

Шаг 1. Разработка цветовой схемы будущего приложения.

Для приложения были выбраны 5 цветов.

- Черный — основной цвет приложения.
- Синий — для строки состояния.
- Зеленый — корректное соединение с сервером и элементов настроек

- Оранжевый — возникновение ошибок, тексты кнопок
- Серый — элементы интерфейса и текст.

Шаг 2. Выбор шрифтового решения.

При выборе шрифта были выбраны два бесплатных шрифта.

Для заголовка на главной странице используем шрифт RockSalt-Regular, взятый из Google Fonts (https://fonts.google.com/specimen/Rock+Salt).

В качестве основного шрифта будет использоваться Roboto. Данный шрифт не имеет засечек и легко читается.

Шаг 3. Создание графических элементов для приложения

При разработке дизайна было создано две иконки на кнопки.

Для первой было взято два референсных изображения

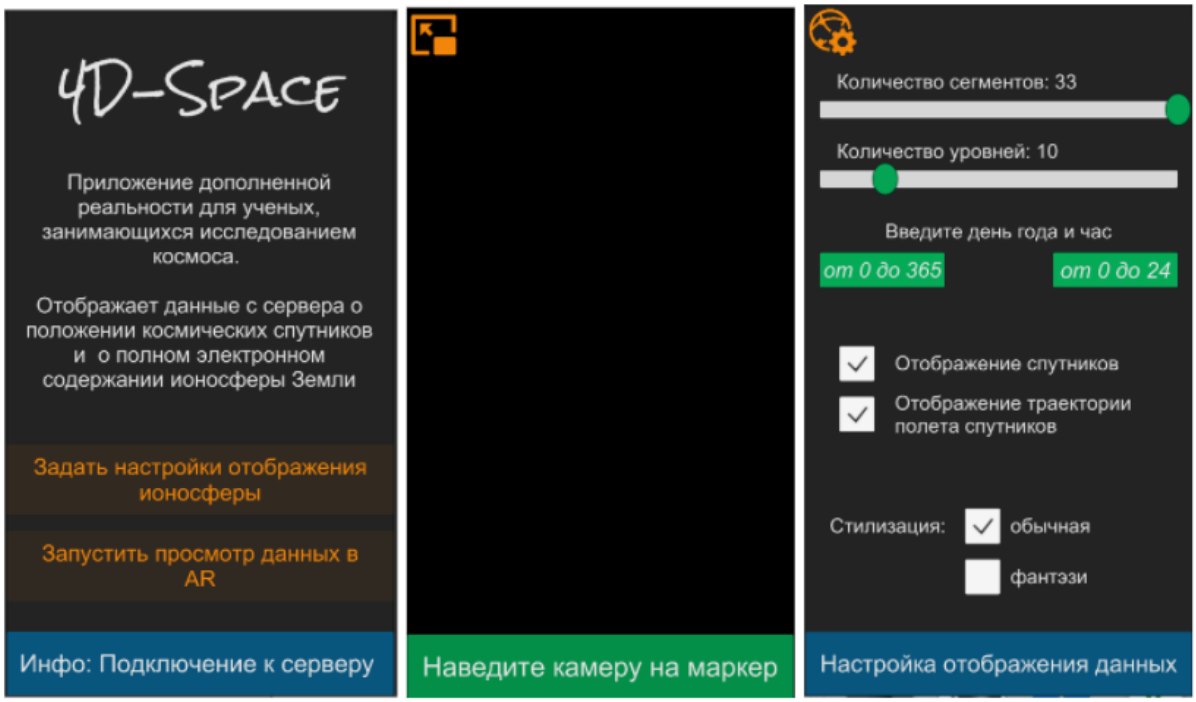
Результат >

Вторая иконка была создана из простых фигур: прямоугольники и стрелочка.

Также, разработан логотип AR-приложения

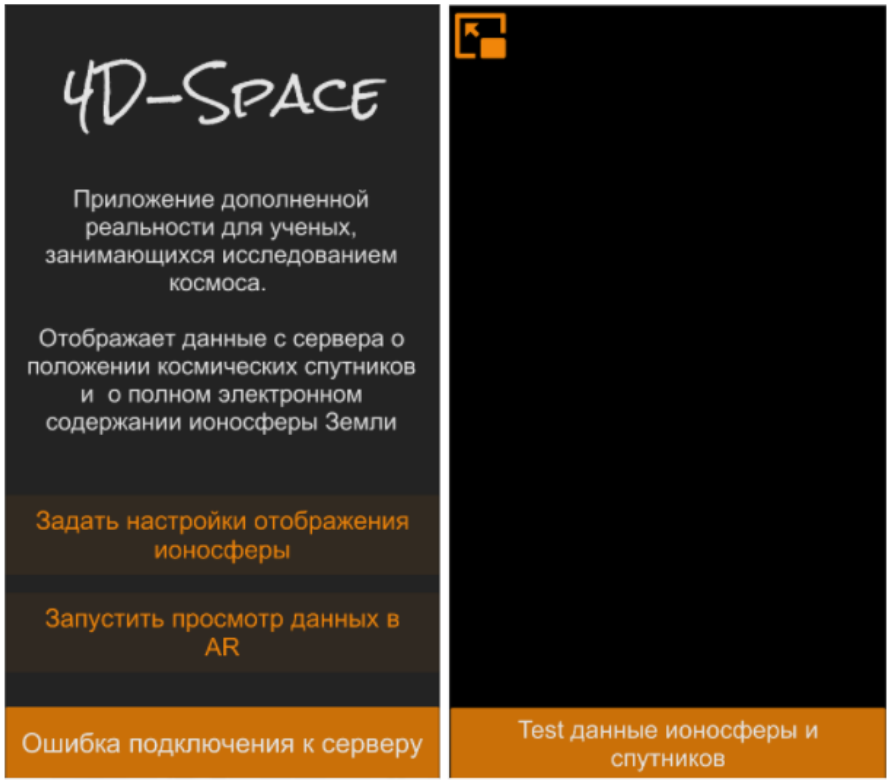
Шаг 4. Создание вайрфреймов AR-приложения.

Было отрисовано 5 экранов будущего приложения.



На главном экране во время подключения к серверу можно перейти на экран настройки или запустить просмотр данных. Если данные получены корректно с сервера, то строка состояния окрашивается в зеленый цвет и появляется соответствующая надпись.

Все настройки было решено вынести на отдельный экран для увеличения рабочего пространства при просмотре данных и в связи с предполагаемым редким их изменением.

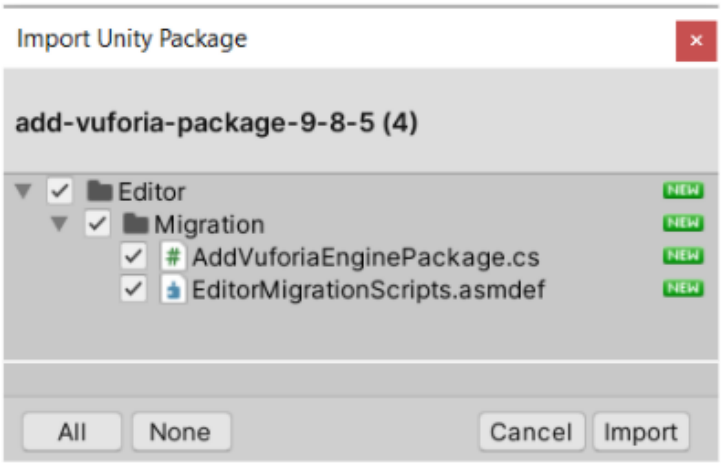


Окрашивание строки состояния в оранжевый цвет будет означать ошибку при подключении к серверу и чтении данных из файла.

Разработка системы привязки виртуальных объектов в приложении

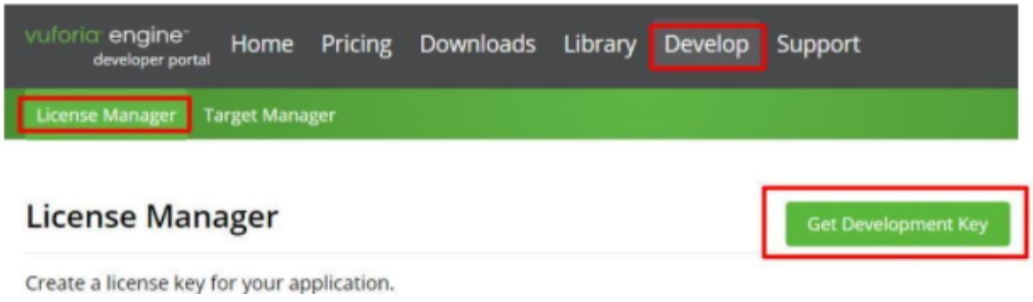
В данном проекте есть три варианта, которые инициализируют появление в камере мобильного устройства визуализации в виде объектов дополненной реальности. В первом из них предполагается использование одного плоского маркера. Второй вариант — обойтись вовсе без инициализирующего объекта (безмаркерная дополненная реальность), третий — взять в качестве маркера куб. Для решения, основанного на любом из трех описанных вариантов, подходит расширение Vuforia, которое можно добавить в проекты на Unity.

Для начала устанавливаем Vuforia в проект. Для этого необходимо зарегистрироваться на официальном сайте (https://developer.vuforia.com/) и перейти в раздел Downloads. Выбрать для скачивания расширение для Unity, которое называется «add-vuforia-package-9-8-5.unitypackage». Далее импортируем скачанный пакет в проект.

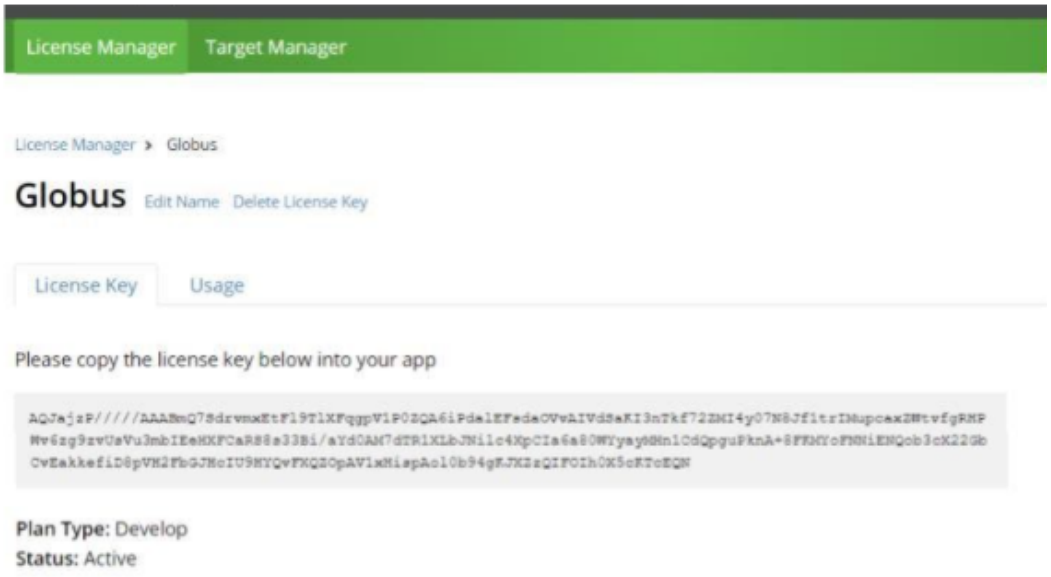


После импортирования Vuforia понадобится некоторое время на автоматическое скачивание библиотеки с Git репозитория.

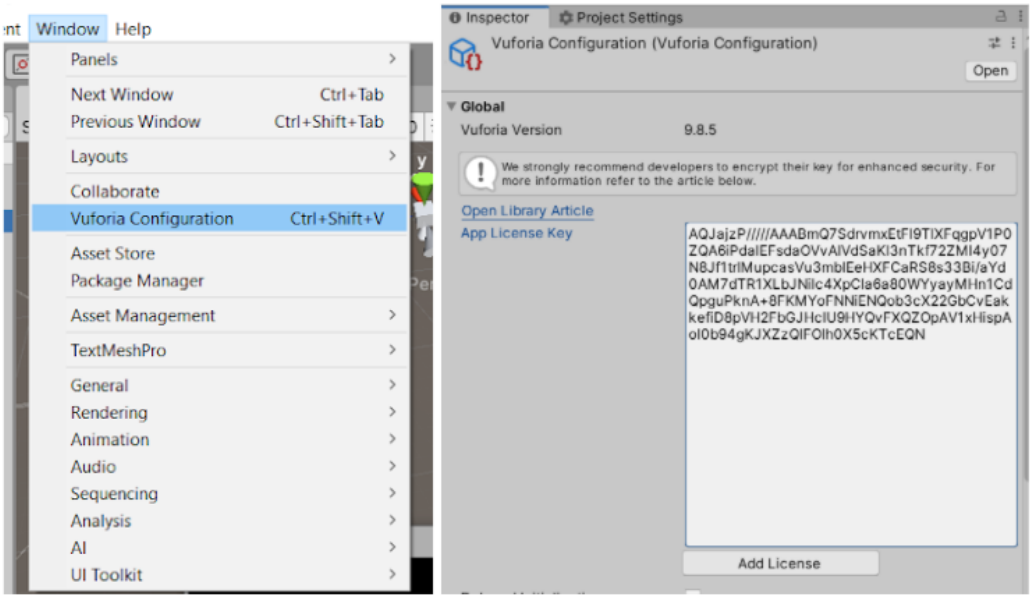
Теперь нужно настроить Vuforia для работы, а именно добавить ключ лицензии. Чтобы получить ключ переходим на сайте в раздел Develop, где сразу попадаем на страницу получения ключей.



Нажимаем кнопку «Get Development Key» и на открывшейся странице записываем имя будущего ключа. Например, «Globus». Создаем ключ и попадаем на страницу с информацией об использовании ключа



Переходим в настройки Vuforia, которые появляются после импортирования библиотеки. В Global настройках вставляем полученный ключ.



Теперь необходимо создать таргеты (маркеры), на которые привязывается дополненная реальность.

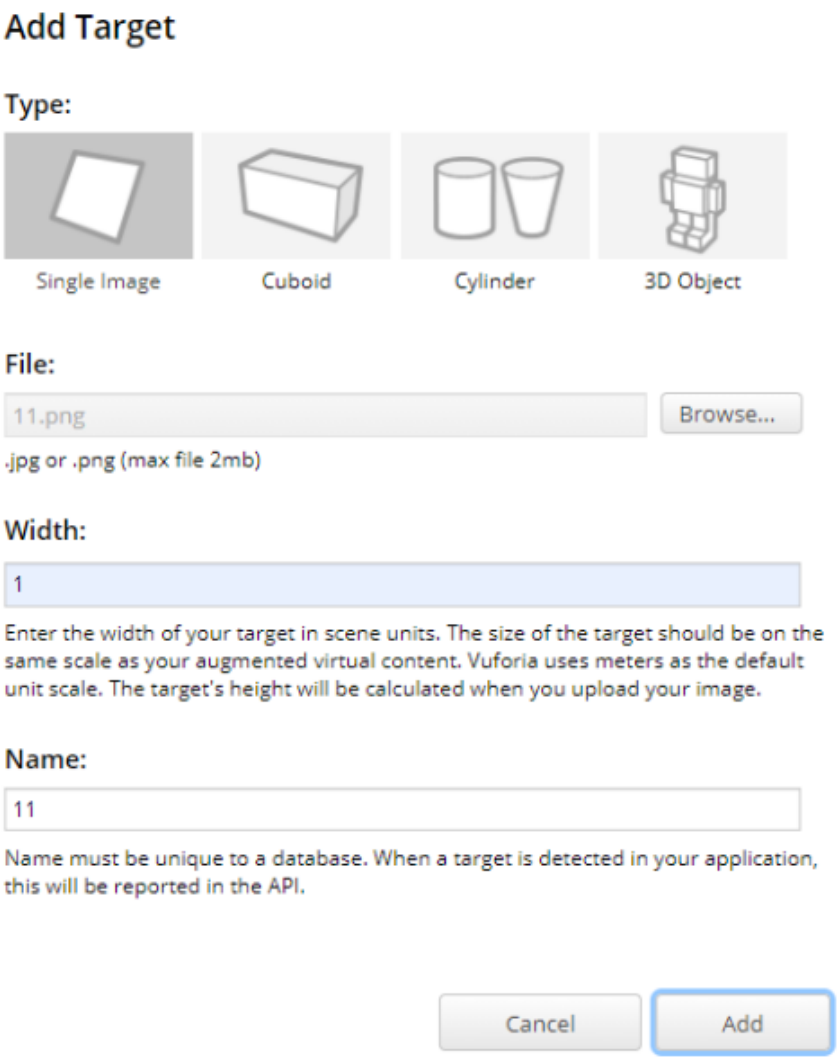
Точки на маркерах в Vuforia создаются на углах и контрастах. Поэтому для таргета необходимо выбрать картинку, где таких точек будет достаточно много.

Привязка к маркеру.

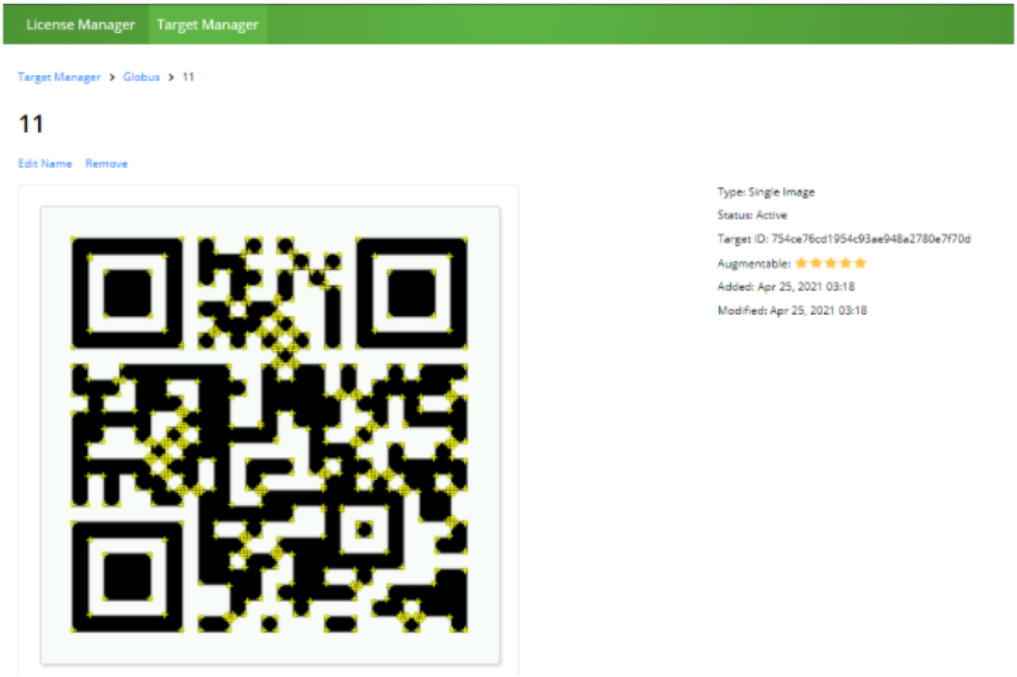
Первым способом реализации дополненной реальности в проекте является привязка к одному маркеру. Хорошим примером таких таргетов является QR-коды.

Для того, чтобы обычная картинка стала маркером нам необходимо загрузить ее на сайт. В разделе Develop выбираем вкладку Target Manager. И создаем новую БД для нашего проекта. Когда БД создана переходим в нее и нажимаем кнопку Add Target.

В открывшемся окне по умолчанию выбран тип Single Image, что нам и нужно для реализации привязки к маркеру. В пункте File выбираем картинку, которую хотим сделать маркером.



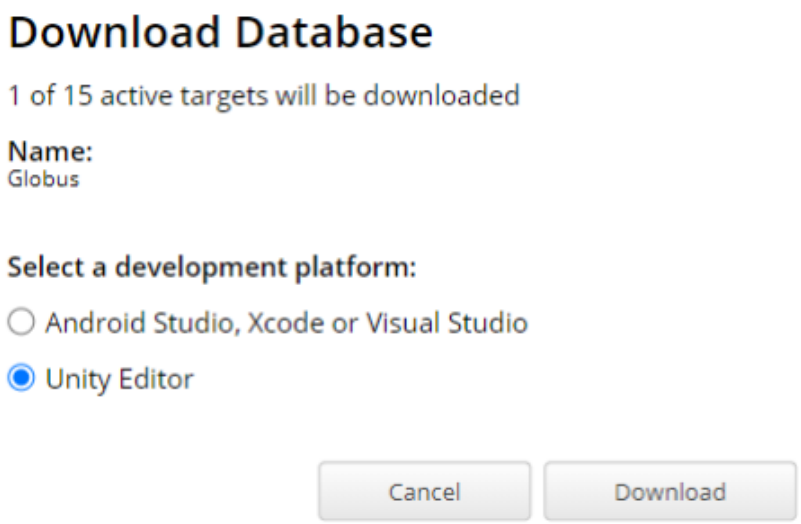
В пункте Width можно указать 1 и поправить в Unity размеры нашего маркера. И задаем имя нашему маркеру.



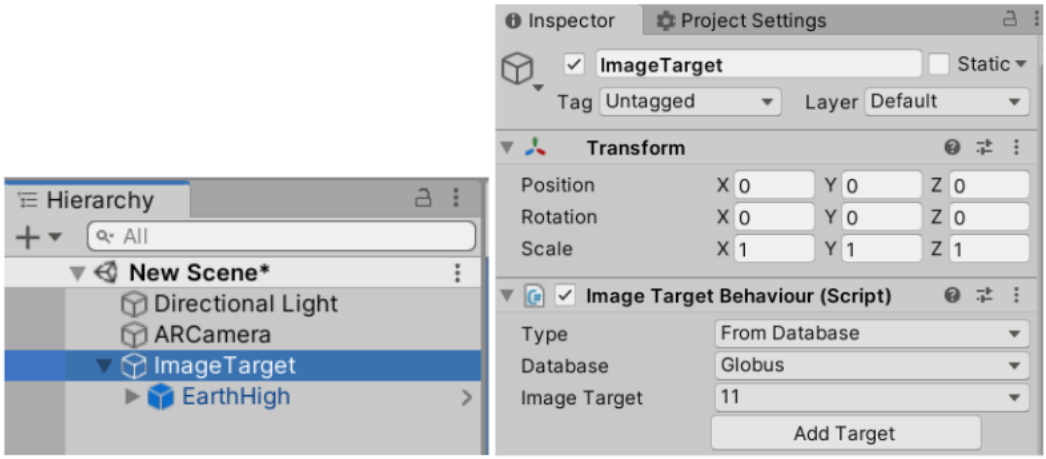
После создания таргета Vuforia позволяет отобразить точки привязки на нем и дает оценку в виде звезд.

Оценка 4-5 звезд означает, что оверлей будет устойчив на маркере и сам маркер будет быстро распознаваться. Маркеры с оценкой 3 и менее обычно плохо распознаются устройствами и оверлей периодически может исчезать, поэтому маркер должен быть с оценкой как можно выше.

После создания маркера необходимо скачать нашу базу данных. При скачивании выбираем пункт Unity Editor.



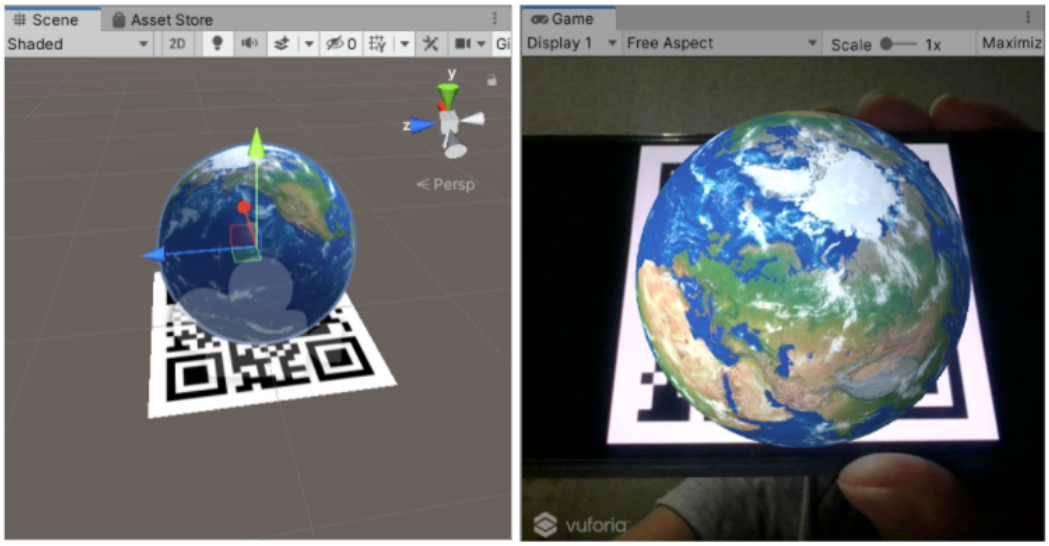
Скачанный маркер импортируем в проект, настраиваем его для отображения дополненной реальности. Для этого а сцену в проекте добавляем объекты ARCamera, которая отвечает за трекинг маркеров в реальности. После ее добавления можно удалить MainCamera. И добавляем ImageTarget.



В настройках ImageTarget указываем Type = FromDatabase и выбираем импортированную БД. В нашем случае она называется Globus. В Image Target выбираем маркер к которому хотим сделать привязку — 11.

Все, что является дочерними объектами ImageTarget считается оверлеем и отображается только при обнаружении соответствующего маркера. Поэтому помещаем нашу 3D-модель Земли (EarthHigh) в объект ImageTarget.

После проделанных действий на сцене должен отобразится сам маркер и Земля. Настраиваем их размеры и положение относительно друг друга.



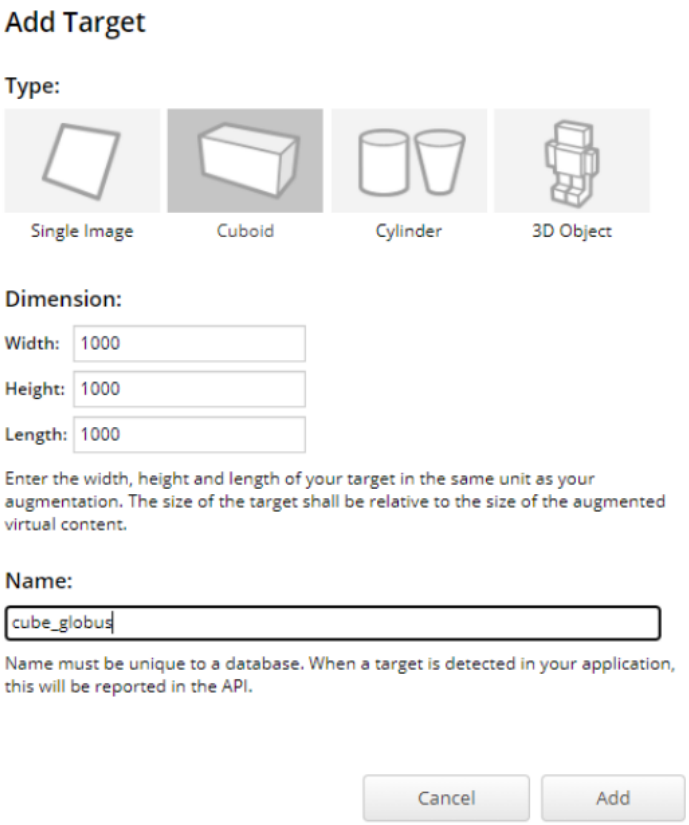
При запуске проекта наш объект EarthHigh отобразится с настройками, которые задали выше, при обнаружении маркера.

Привязка к кубическому маркеру

В Vuforia также доступна привязка к прямоугольнику или кубу. Для реализации такого таргета потребуется 6 маркеров по числу сторон куба.

Если использовать QR-коды на всех сторонах куба, то Vuforia будет путать их между собой, поэтому необходимо стилизовать QR-коды, например, поместить в центр различные надписи/картинки. Или вместо QR-кодов можно использовать обычные картинки с четкими границами.

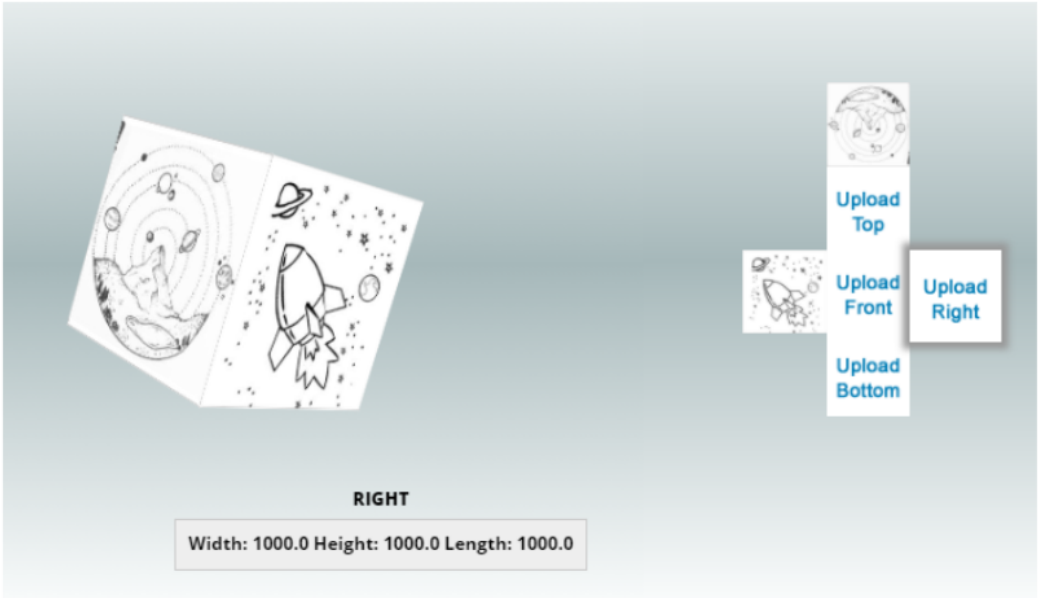
Создаем в Vuforia новый маркер. В Type нужно выбрать кубический маркер. А в размерах указать одну величину, так как нам нужен куб и соотношение сторон у него одинаковое. Оптимально будет указать значение 1000 и загружать картинки размером 1000 на 1000 пикселей. Тогда изображение будет четкое, а маркер сразу достаточно большим.



Теперь в Vuforia открывается страница превью куба, где можно загрузить картинки.

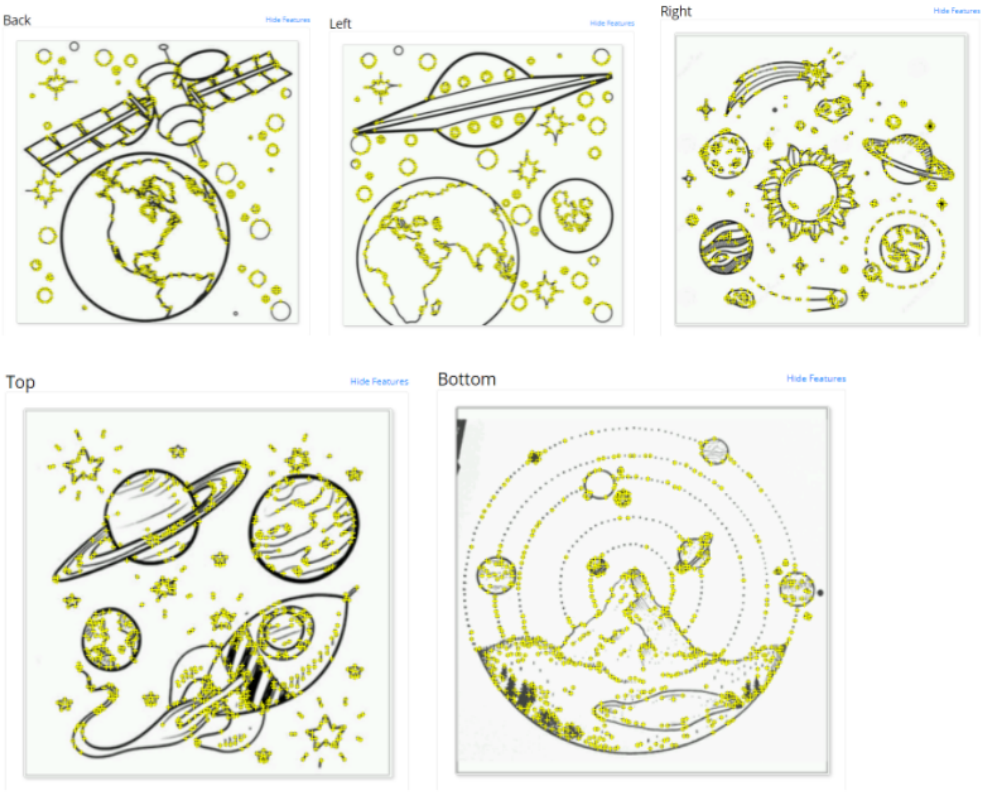
Картинки должны быть размером 1000 на 1000 пикселов и в формате 8 или 24 bit PNG или JPG. Чтобы не возникало ошибок при загрузке можно воспользоваться онлайн конвертером, например, https://online-converting.ru/image/convert-to-png/ или любым графическим редактором.

Подготовленные картинки подгружаем в куб.



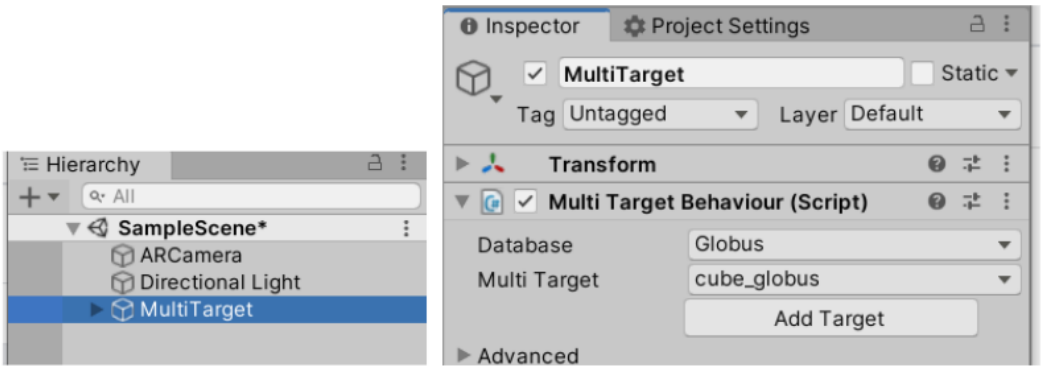
После загрузки всех сторон можно нажать на куб дважды, для открытия просмотра каждой стороны в отдельности и ее рейтинг.

Выбирая справа стороны, смотрим оценки маркеров. В нашем случае все маркеры оценены на 5 звезд. Ниже представлены все стороны созданного куба с отображением ключевых точек.



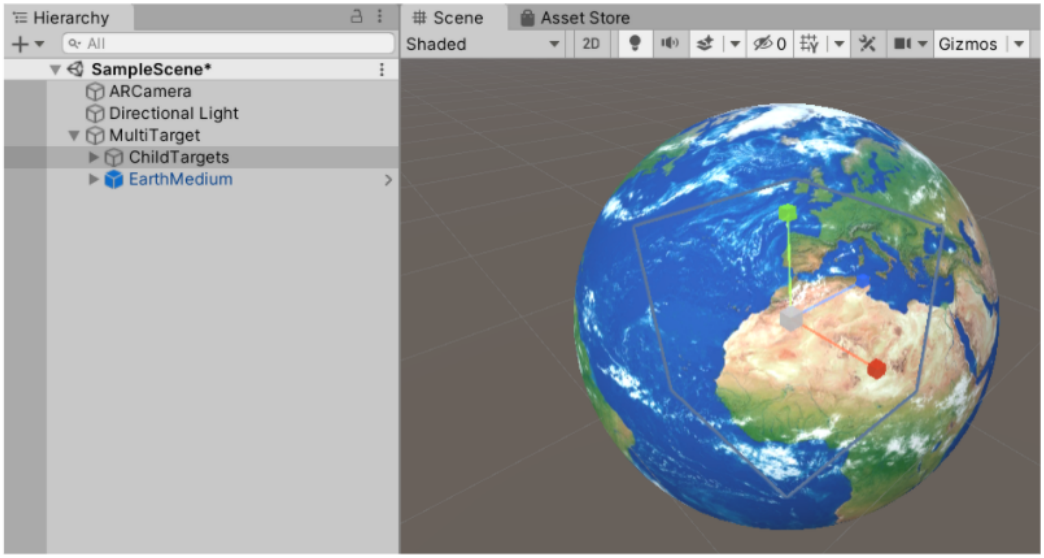
Теперь скачиваем созданный маркер-куб, так же, как и для одиночного маркера. И импортируем unitypackage в проект.

Добавляем на сцену MultiTarget, который соответствует кубическому типу маркеров. И в его настройках указываем нашу базу и название нашего маркера.



Если все верно, то на сцене отображается куб со всеми нашими маркерами. Теперь настраиваем отображение Земли так, чтобы она полностью покрывала маркер-куб. Опять же, 3D-модель должна быть дочерним объектом таргета.

На картинке можно увидеть куб в виде серых полос.

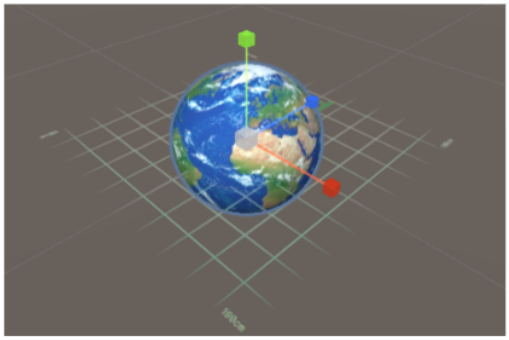


Теперь при запуске проекта куб будет полностью перекрыт на экране отображением Земли и вращать планету можно простым поворотом куба.

Безмаркерная дополненная реальность

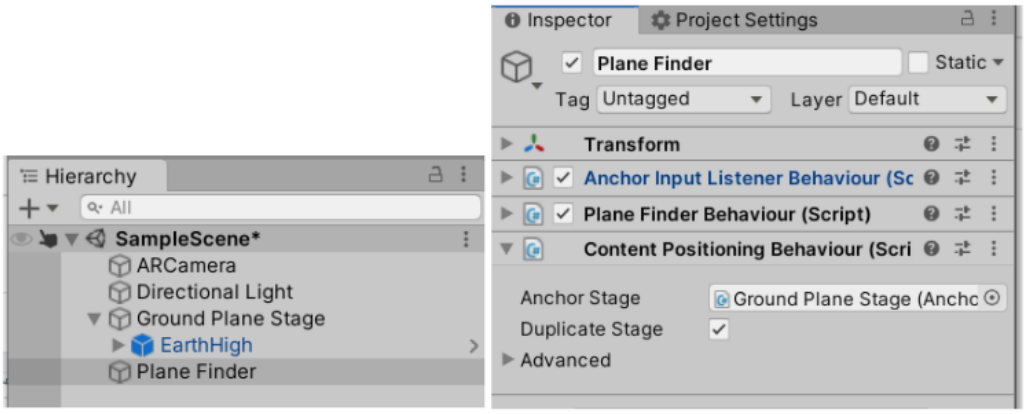
В данном виде технологии AR объект позиционируется на ровной поверхности изображения в камере.

Для его создания добавляем на сцену объект Ground Plane Stage — компонент безмаркерной технологии в Vuforia, который должен стать родительским для всех отображаемых объектов. Настраиваем отображение Земли.

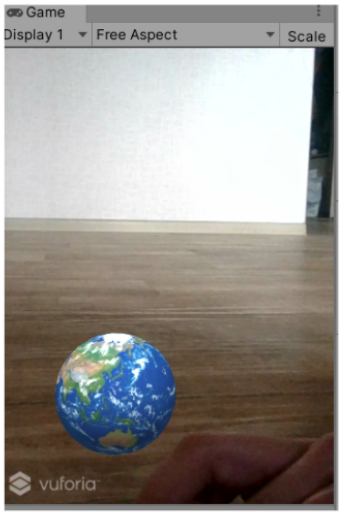


И нам понадобится второй компонент Plane Finder, который отвечает за настройку нахождения прямых поверхностей и добавления на них ранее созданного Ground Plane Stage.

Привязываем этот объект в последнем компоненте Plane Finder, который называется Content Positioning Behaviour.



Теперь при запуске проекта на устройстве у нас автоматически определится ровная поверхность, куда разместиться наш объект.



Задачи программиста-аналитика

В задачи программиста-аналитика входит получение данных об ионосфере и спутниках, их обработка и визуализация, а также итоговая сборка всех частей проекта. Рассмотрим все подзадачи по порядку.

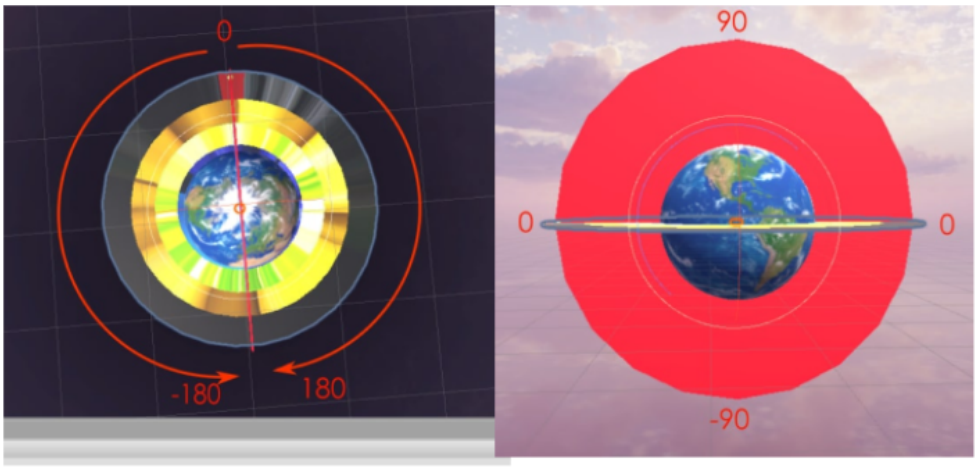
Для реализации данного проекта была выбрана платформа Unity, описываемые решения будут представлены для нее.

Написание клиента для подключения к серверу и сбора данных 4D-Space

Для начала реализации задачи необходимо получить данные с сервера.

Данные ионосферы представлены в следующем формате: latitude, longitude, height, plasma.

Все данные привязаны соответственно к земному шару и представлены на картинке ниже



С сервера можно сделать два типа запроса для получения данных по ионосфере:

1. на получение данных ионосферы вокруг всей планеты;
2. получение сегмента данных, соответствующих взгляду пользователя.

1. Получение всех данных об ионосфере на текущий момент с сервера происходит по запросу

URL: 178.250.159.250:8000/plasma

Ответом сервера json:

{ longitude: [23.5, 12.56, -34, ...],

latitude: [34.5, 56.4, -12...],

height: [12, , 90, 78, ...],

plasma: [1213, 45456, 45342, ...] }

Ограничения по значениям параметров:

longitude: [-180, 180]

latitude: [-90, 90]

height: [90, 1000]

4D-Space данные не статичны и постоянно меняются с течением времени. Дополнительной подзадачей является отображение данных по запросу с определенного времени. Эти данные можно запросить, указав соответствующие параметры запроса (опционно):

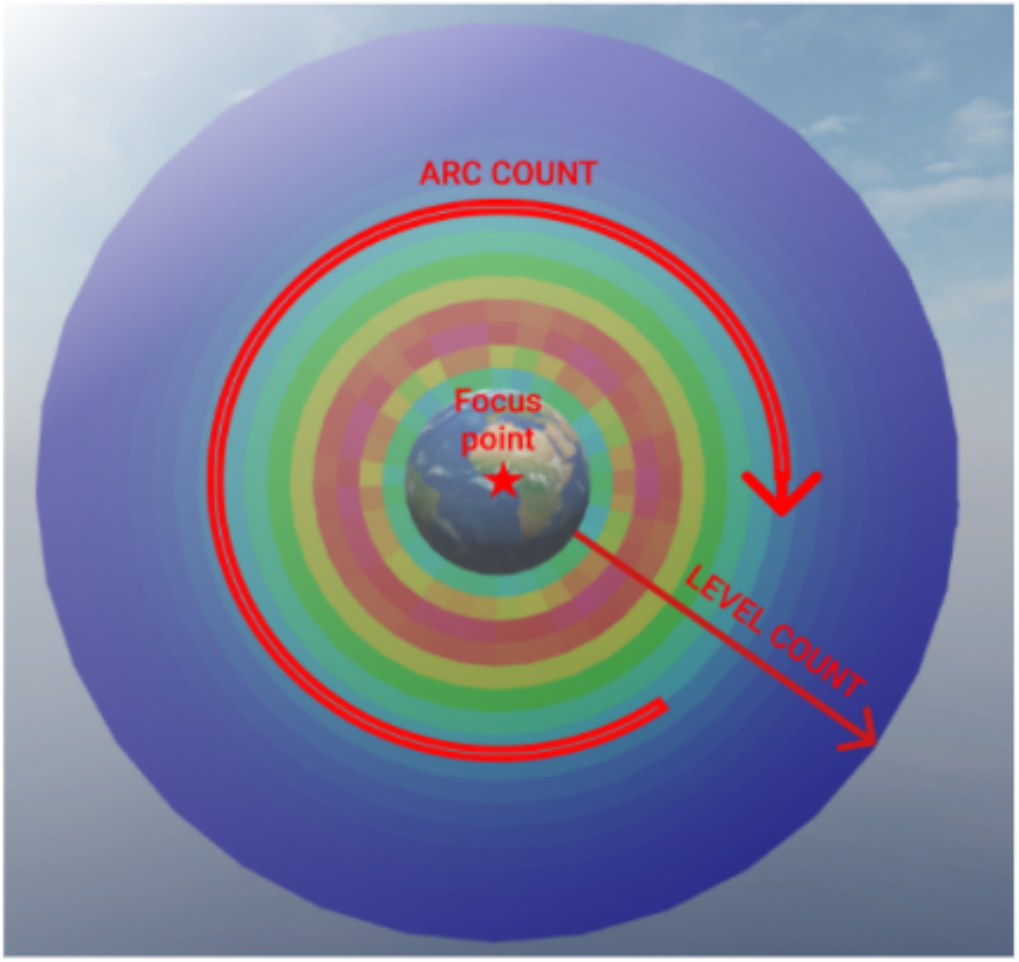
hour (int)[0:24] — время в часах;

day (int)[1,365] — день года;

2. Получение данных об ионосфере по фокусу взгляда пользователя.

Для получения не всех данных ионосферы, а конкретного сегмента, нам необходимо знать:

- Focus point — это точка центра фокуса пользователя и ее необходимо указывать координатой.
- Level count — количество сегментов по высоте.
- ARC count — количество дуг, на которые разделена окружность.



Запросить данные сервера по фокусу взгляда пользователя можно запросом:

URL: 178.250.159.250:8000/segment

В параметры запроса необходимо указать:

heights (int) — количество уровней данных по высоте;

n_segment (int) — количество дуг на которые должны быть разбиты данные;

lat, long (float) — координата, направление взгляда пользователя, фокус взгляда пользователя.

Пример запроса:

`http://83.136.232.74:8000/segment?height=16&n_segment=16&lat=16&long=16`

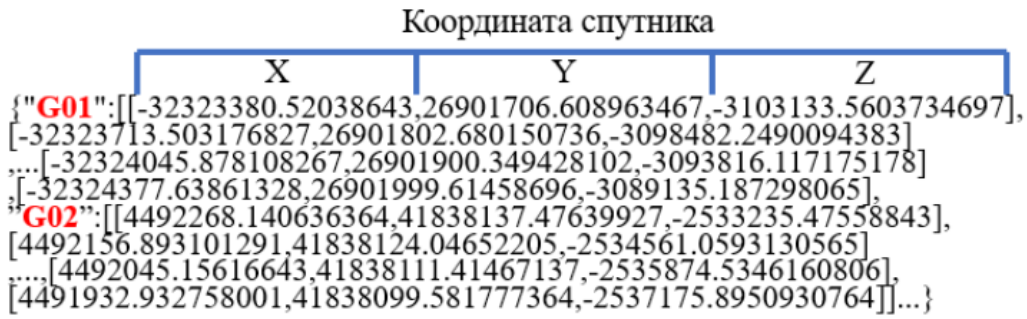
Примечание: Данный запрос разработчики советуют использовать для сравнения полученного самостоятельно сегмента из цельных данных. Так как вызов данных по сегменту в определенный промежуток времени невозможен.

3. Также необходимо получить данных о спутниках.

Получение спутников происходит по запросу:

URL: 178.250.159.250:8000/sats

Пример ответа сервера:



Сервер присылает словарь, внутри которого содержатся данные о маршруте некоторого количества спутников. Необходимо найти среди спутников системы G (Glonas), их 32 и названия в словаре соответствуют системе и номеру, например "G01". Данные других спутников не нужно отображать.

Массив соответствующего спутника содержит координаты последовательного изменения положения этого спутника каждые 15 минут.

Решение

Для реализации запросов к серверу и ожидания ответа на них в Unity используем Coroutine. Корутины представляют собой простые C# итераторы, возвращающие IEnumerator и использующие ключевое слово yield для ожидания ответа по определенному условию.

А для хранения данных будет использоваться встроенный класс Unity PlayerPrefs, который позволяет сохранять определенные данные пользователя и передавать их между сценами или внутри одной сцены между различными элементами.

Создаем скрипт Connect_Server().

```
using System; //добавляем для работы со временем
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking; //подключаем класс для работы с сервером

public class Connect_Server: MonoBehaviour
{
    //для реализации запросов к серверу используем Coroutine и одну функцию на все
    ↪  запросы
    //передаем Url запроса и название переменной, в которую будет записываться ответ
    ↪  сервера
    private IEnumerator getRequest(string url, string type)
    {
        //создаем поток связи с сервером
        using (UnityWebRequest req = UnityWebRequest.Get(url))
        {
            //чтобы продолжить, дожидаемся ответа сервера
            yield return req.SendWebRequest();

```

```
            //проверяем, не возникло ли ошибок
            if (req.isDone && !req.isNetworkError && !req.isHttpError)
            {
                //если ответ от сервера получен
                //записываем в реестр соответствующую переменную
                PlayerPrefs.SetString(type, req.downloadHandler.text);
                PlayerPrefs.Save();
            }
            else
            {
                //действия при ошибке: пока только отображаем в консоле
                Debug.Log("error_connect_server");
            }
        }
    }
```

Теперь пишем код к функциям, которые и будут вызывать getRequest с соответствующими параметрами.

Первой функцией будет получение значений ионосферы, с различной вариацией запросов.

```
  //значение плазмы можно получать во времени, поэтому должна быть модификация запроса
  ↪  и на время
    //в функцию будем передавать переменную time, в которой записана информация о
    ↪  времени в следующем формате ?hour=16&day=16
    public void get_plasma(string time = null)
    {
        string type = "plasma_all"; //значение переменной type будет в дальнейшем
        ↪  использовано как название переменной в которой будут хранится полученные
        ↪  данные
        string PlasmaUrl = "http://178.250.159.250:8000/plasma";

        //если информацию о времени передали, то записываем параметры времени запрос
        if (time != null) {
            PlasmaUrl += time;
        }

        Debug.Log("Comment: Получение значений плазмы по всей Земле");
        StartCoroutine(getRequest(PlasmaUrl, type));//вызываем корутину по
        ↪  получившемуся запросу
    }
```

Далее создаем по тому же принципу функции для получения других данных.

```
public void get_sputniks()
    {
        string type = "sputniks";
        string SatsUrl = "http://178.250.159.250:8000/sats";
        Debug.Log("Comment: Получение информации о спутниках");
        StartCoroutine(getRequest(SatsUrl, type));
    }

    //функция получения данных по сегменту будет принимать данные сегмента
    public void get_segment(int height_v, int n_segment_v, int lat_v, int long_v)
    {
        string type = "plasma_segments";
        string SegmentUrl = "http://178.250.159.250:8000/segment?height="+
        ↪  height_v+"&n_segment="+ n_segment_v + "&lat="+ lat_v + "&long="+ long_v;
```

```
14          Debug.Log("Comment: Получение данных плазмы по сегменту");
15          StartCoroutine(getRequest(SegmentUrl, type));
16      }
17  }
```

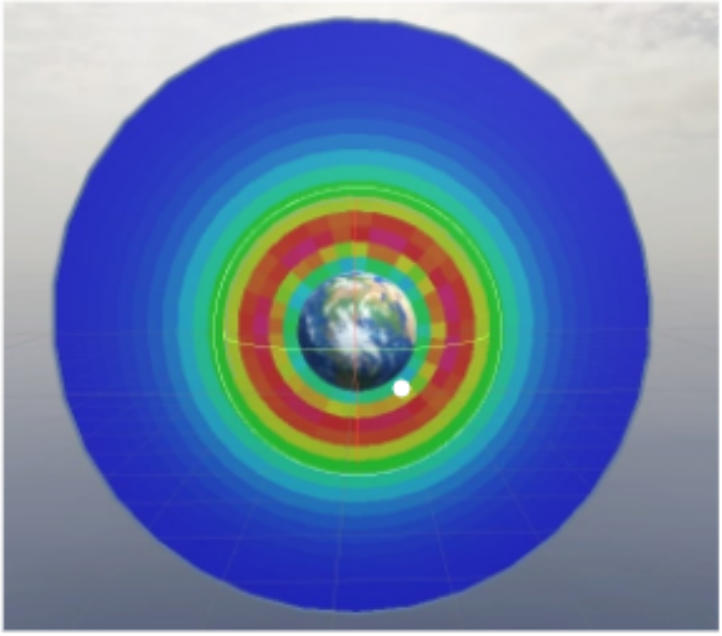
В дальнейшем для использования полученных данных нужно использовать следующий метод:

```
1  PlayerPrefs.GetString(type); // где type - название переменной
2            //"plasma_all" - данные по всей ионосфере
3            //"plasma_segments" - данные ионосферы по сегменту
4            //"sputniks" - данные спутников
```

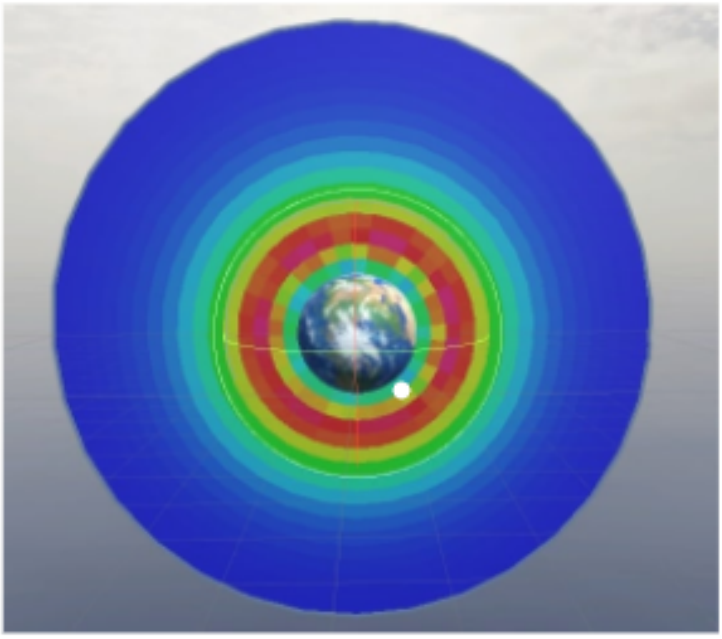
Расчет и разработка алгоритма для визуализации данных о состоянии ионосферы

После получения данных, необходимо их обработать и отобразить.

У ионосферы отображается только часть данных: срез шара центр которого находится в точке взгляда пользователя (направления камеры), и проходящий через центр Земли. Пример отображения ионосферы представлен на картинке ниже.



Предполагаемый окрас данных реализован в тепловой схеме от синего (наименьшее значение данных) к красному (наибольшее значение данных)



Для отображения данных также возможно использование цветовой модели Grayscale (серая шкала). Но данный метод не рекомендуется, так как он будет менее информативен.

Если будет использована другая шкала цветов, то ее выбор необходимо обосновать.

Решение

Данную задачу следует разбить на несколько шагов:

- сериализация полученных данных по ионосфере;
- сопоставление систем координат данных;
- создание цветовой схемы отображения данных;
- генерация поля ионосферы (среза данных);
- изменение данных ионосферы при повороте среза данных.

Обработка полученных данных об ионосфере

Для удобной дальнейшей работы с данными по ионосфере создадим соответствующий класс (скрипт Data) с пометкой о сериализации, таким образом мы сможем быстро преобразовать полученные данные в отдельные массивы.

```
 //создаем для ионосферы класс
[Serializable]
class Data
{
    //публичные переменные для данных с сервера
    public float[] latitude;
    public float[] longitude;
    public float[] height;
    public float[] plasma;
}
```

Так как данные с сервера приходят в формате JSON, необходимо его сериализовать на массивы с помощью служебной функции JsonUtility. Вызывать функцию будет необходимо после получение данных с сервера, поэтому запишем все в отдельную функцию в новом файле, который назовем Data_Init.

```
public Data info_plasma;//переменная класса Data в которой и будут хранится данные

public void data_processing()
{
    Debug.Log("Comment: получение данных разбивка на массивы");
    info_plasma = JsonUtility.FromJson<Data>(PlayerPrefs.GetString("plasma_all"));
    ↪  //PlayerPrefs.GetString("plasma_all") - берет информацию по плазме из
    ↪  пространства имен PlayerPrefs
}
```

Теперь с данными будет удобно работать и мы можем приступить к следующему шагу.

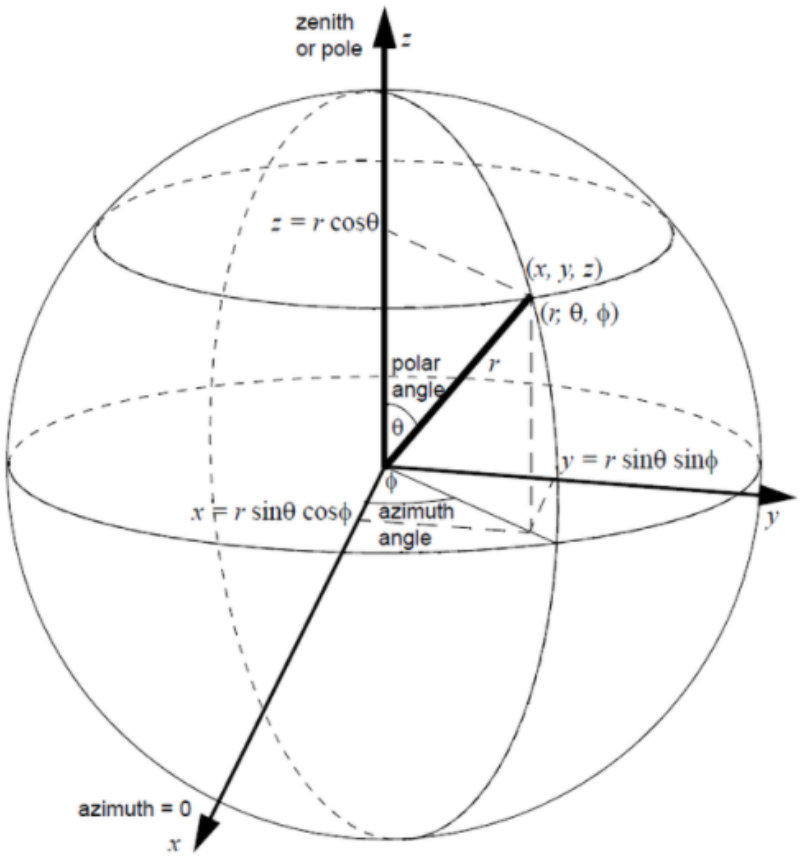
Но для дальнейшей работы так же, будет удобнее временно записать данные с сервера в файл, чтобы каждый раз при отладке не дожидаться ответа сервера. В таком случае код по сериализации Data будет выглядеть так:

```
  void data_processing()
    {
        //указываем ссылку на файл с данными
        string path = "Assets/Resources/json/data.json";

        //читаем данные из файла в текстовую переменную
        StreamReader reader = new StreamReader(path);
        string data = reader.ReadToEnd();
        reader.Close();

        //сереализуем данные в класс Data
        info_plasma = JsonUtility.FromJson<Data>(data);
    }
```

Сопоставление систем координат у данных с сервера с системой координат в Unity

При представлении местоположения объектов в трехмерных измерениях существует несколько различных типов систем координат, которые можно использовать для представления местоположения относительно некоторой исходной точки.

Данные, которые мы получаем с сервера приходят в сферической системе координат. Сферическая система координат использует три параметра для представления точки в пространстве, радиального расстояния (r — радиус земли или в нашем случае высота значений ионосферы) от исходной точки до точки в пространстве, зенитный угол (Θ — широта) от положительной оси z и азимутальный угол (Ф — долгота) от положительной оси x, все отсчеты начинаются в центре Земли.



Декартова система координат — это стандартная система координат x, y и z, которая и используется в Unity.

На рисунке выше дается графическое представление сферической системы координат. Он также указывает на эквивалентные размеры в декартовой системе координат.

Глядя на фигуру в сферических координатах, мы можем вычислить компоненты x, y и z декартовой системы координат.

Произведем математические выкладки для дальнейших вычислений.

Мы начнем с компонента z, потому что его легче всего вычислить. Поскольку компонент z находится в той же плоскости, что и r и угол Θ, и эта плоскость перпендикулярна плоскости xy, в которой находится угол Φ, мы можем видеть, что компонента z эквивалентна смежной стороне треугольника, и r эквивалентно гипотенузе. Принимая во внимание эту информацию, мы можем вычислить компонент z следующим образом:

$$z = rcos\Theta$$

Чтобы вычислить компоненты x и y, нам нужно сначала вычислить проекцию линии радиуса на плоскость xy. Эта проекция была бы эквивалентна гипотенузе двух треугольников в плоскости xy. Следующая формула даст использовать длину проекции r:

$$r_{projection} = rsin\Theta$$

Компонента x равна проекции проекции r на плоскость xz. Мы можем вычислить

компонент x, используя следующую формулу:

$$x = r_{projection} cos\varphi = rcos\varphi sin\Theta$$

Компонента y равна проекции проекции r на плоскость yz. Мы можем вычислить компонент y по следующей формуле:

$$y = r_{projection} sin\varphi = rsin\varphi sin\Theta$$

Теперь, когда мы можем преобразовать из сферических координат в декартовы, было бы полезно сделать обратное. Используя теорему Пифагора, мы можем вычислить длину r.

$$r = \sqrt{x^2 + y^2 + z^2}$$

Используя формулу для компонента z, мы можем вывести формулу для вычисления угла Θ.

$$z = rcos\Theta \Rightarrow \Theta cos^{-1}\frac{z}{r}$$

$$\Theta = cos^{-1}\frac{x}{\sqrt{x^2 + y^2 + z^2}}$$

Подставляя формулы для компонентов x и y, мы можем решить значение угла Φ.

$$\frac{y}{rsin\varphi} = sin\Theta = \frac{x}{rcos\varphi}$$

$$\frac{y}{x} = \frac{sin\varphi}{cos\varphi} = tan\varphi$$

$$\varphi = tan^{-1}\frac{y}{x}$$

При работе с тригонометрическими функциями единицы измерения должны быть в радианах, а не в градусах. Радианы — это стандартная единица измерения угла. Радиан — это угол, который выметается, когда радиус окружности и длина начерченной дуги равны по длине.

$$\Theta_{radians} = \Theta_{degrees} \cdot \frac{\pi}{180}$$

Полный круг 360° эквивалентен 2π радианам. Таким образом, преобразование из градусов в радианы выглядит следующим образом:

$$\Theta_{degrees} = \Theta_{radians} \cdot \frac{180}{\pi}$$

Преобразование радианов в градусы выглядит следующим образом:

Теперь переходим к программированию. Лучшим вариантом будет возврат данных в формате Vector3 для того, чтобы отобразить в данной точке объект модели данных (сегмент сектора, который будет представлен 3D или 2D объектом).

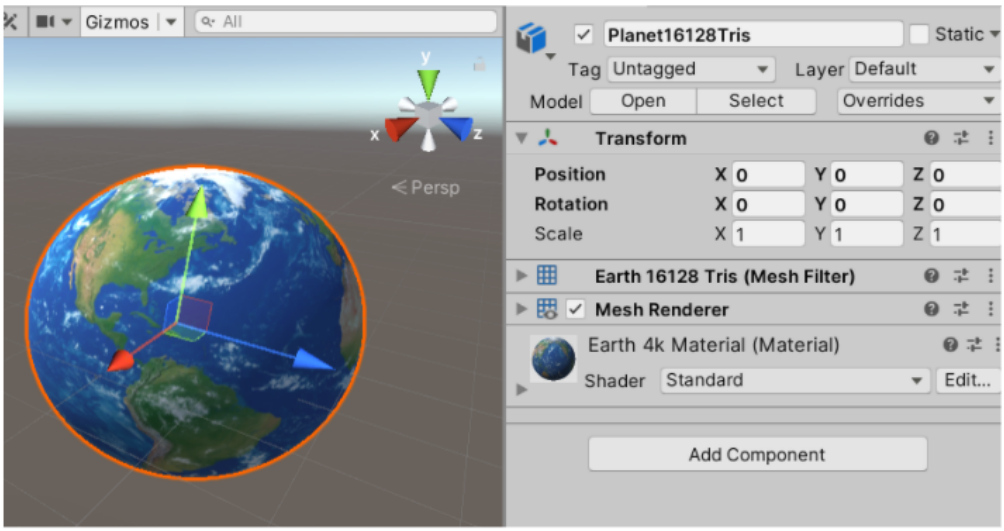
```
Vector3 getXYZ(float lat, float lon, float height)
   {
       lat = lat * Mathf.PI / 180;
       lon = lon * Mathf.PI / 180;

       float x = height * Mathf.Cos(lat) * Mathf.Cos(lon);
       float y = height * Mathf.Sin(lat);
       float z = height * Mathf.Cos(lat) * Mathf.Sin(lon);

       return new Vector3(x, y, z);
   }
```

И нам понадобится функция для возврата обратных величин при определении взгляда пользователя в географической системе координат.

```
//перевод координат xyz в latlong
    float[] getLatLng(Vector3 pointXYZ)
    {
        float r = Mathf.Sqrt(pointXYZ.x * pointXYZ.x + pointXYZ.y * pointXYZ.y +
        ↪  pointXYZ.z * pointXYZ.z);
        float lat = Mathf.Asin(pointXYZ.z / r) * 180 / Mathf.PI;
        float lon = Mathf.Atan2(pointXYZ.y, pointXYZ.x) * 180 / Mathf.PI;
        float[] latlong = { lat, lon, r };
        return latlong;
    }
```

Для наглядности понимания расположения данных отобразим Землю и на месте точек с данными белые шары, чтобы проверить, что все работает и посмотреть совместимость единиц измерений, которые получаются при использовании getXYZ().

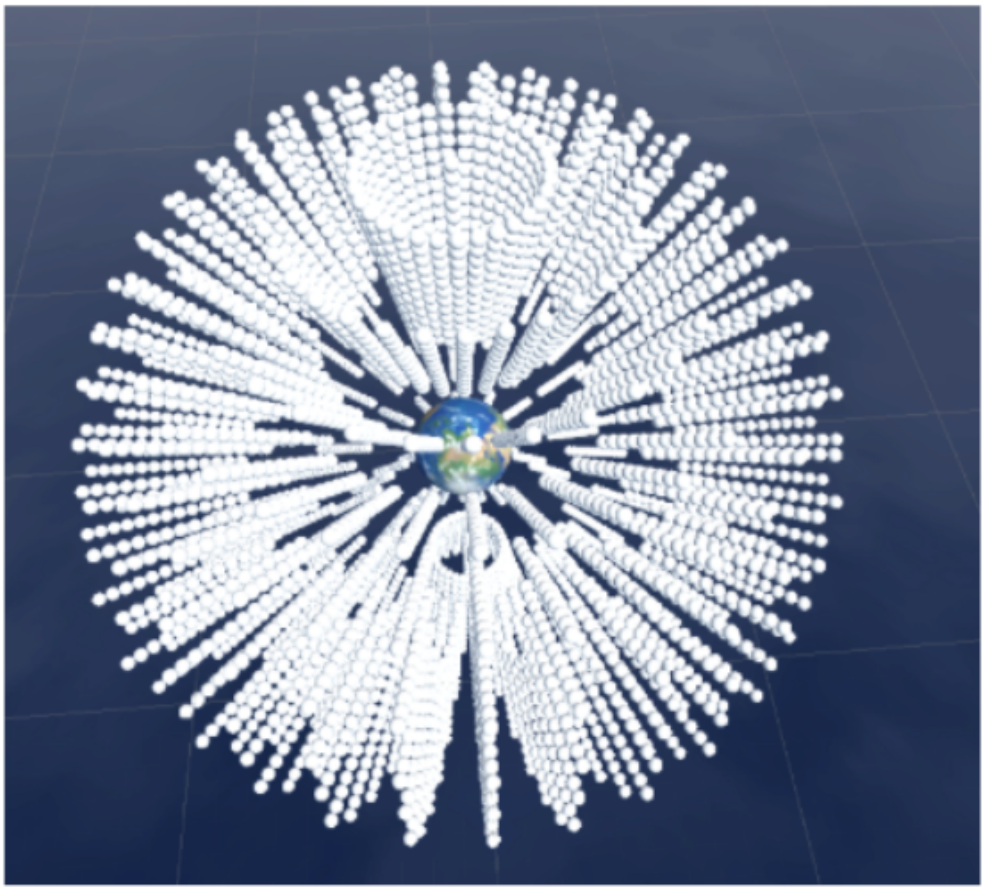
Изначально добавим на сцену 3D-модель Земли в точке (0,0,0).



Теперь напишем скрипт для создания новых объектов на сцене, а конкретно типовых шаров в точках, которые получены с сервера. Сам шар мы временно создадим из стандартной фигуры Shape в Unity и сохраним как префаб, для дальнейшего прикрепления в скрипте.

```
public Transform prefab;//временный 3D-объект для присоединения дефолтного шара на
↪  месте полученных данных

public void Start()
{
    data_processing();//сериализация класса Data
      int lenght = info_plasma.latitude.Length;
    for (int i = 0; i < lenght; i++)//перебор всех данных из Data
    {
          Instantiate(prefab, getXYZ(info_plasma.latitude[i],
          ↪  info_plasma.longitude[i], info_plasma.heigth[i]),
          ↪  Quaternion.identity);//спавн шара в пространстве Unity
    }
}
```

Добавим написанное в ранее созданный скрипт Data_Init и запустим проект. Результат представлен на рисунке ниже.



Для ее наглядности длина цикла перебора данных была искусственно уменьшена (length заменено на 7000), тогда как сервер возвращает более 100 тысяч значений.

Создание цветовой схемы отображения данных

Необходимо создать функцию, возвращающую цвет, в зависимости от значения plasma.

Для реализации тепловой схемы будем использовать формат HSV и его конвертацию в RGB формат, так как HSV можно вычислить в процентном соотношении от максимального значения plasma.

В данном формате используется три параметра: H (оттенок), S (насыщенность) и V (яркость). Все параметры должны быть в интервале [0...1]. S и V следует сделать постоянными величинами, для понятной цветовой схемы. Насыщенность возьмем максимальную (1), а яркость чуть выше среднего (0.7).

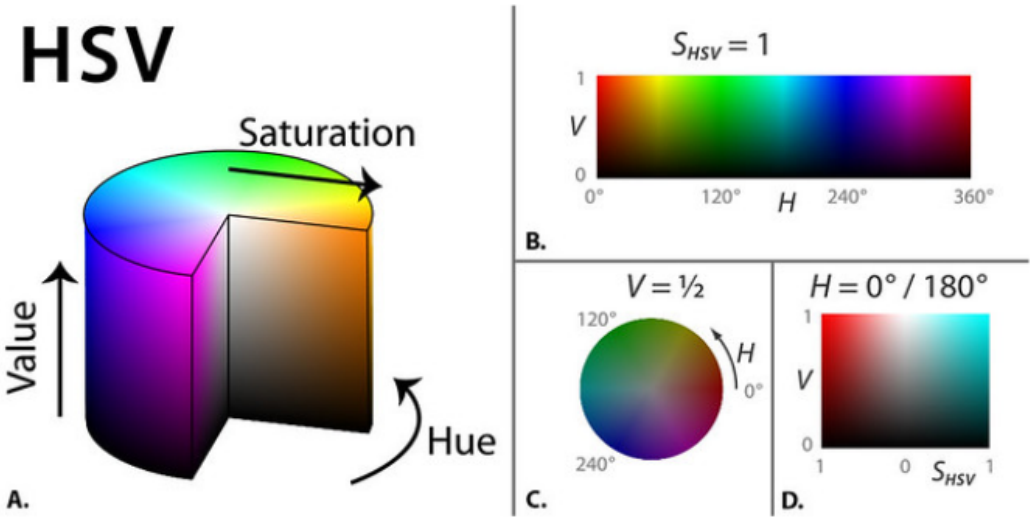
Теперь необходимо определить оттенок, то есть наш основной цвет. Для этого необходимо знать максимальное значение plasma. Дописываем в класс Data строки по их нахождению.

```
public float max_value;
```

```
    public float min_value;

    public void start_connect()
    {
        //присваиваем максимуму начальное значения плазмы
        max_value = plasma[0];

        //вычисляем максимальное значение в плазме перебирая и сравнивая значения
        for (int i = 0; i < plasma.Length; i++)
        {
            if (plasma[i] > max_value) max_value = plasma[i];
         }
    }
```

Нахождение max_value вынесено в отдельную функцию, которую нужно запускать после получения данных с сервера.

На картинке представлены значения и формулы вычисления HSV.



После получения максимального значения plasma, вычисляем интервал изменения цвета. Так как по условию необходимо наибольшее значение делать красным, а наименьшее — синим, то нам нужные не все цветовые значения (холодные цвета от синего к красному нужно исключить), поэтому вводим дополнительный коэффициент

Вставим в функцию start_connect() в скрипте Data определение цвета всех значений, чтобы в дальнейшем сразу посмотреть результат.

```
        float step = 180 / (max_value * 0.4f); //вычисляем интервал изменения цвета
        color_val = new Color[len];//устанавливаем длину массива цветов

//вычисляем цвета для всех точек для примера
        for (int i = 0; i < plasma.Length; i++)
        {
            float color = step * plasma[i] / 360; //вычисляем получившееся значение на
            ↪  полном цветовом диапазоне HSV
            color_val[i] = Color.HSVToRGB(color, 1, 0.7f);
        }
```

Функция HSVToRGB() в Unity возвращает уже RGB формат, который можно в дальнейшем использовать для окраски данных.

Перепишем функцию Start в Data_Init так, чтобы у объектов (шаров) появился цвет данных.

```
public Transform prefab;//временный 3D-объект для присоединения дефолтного шара на
↪  месте полученных данных

public void Start()
{
    data_processing();//сериализация класса Data
    int lenght = info_plasma.latitude.Length;
    for (int i = 0; i < lenght; i++)//перебор всех данных из Data
    {
            var myNewObject = Instantiate(prefab, getXYZ(info_plasma.latitude[i],
            ↪  info_plasma.longitude[i], info_plasma.heigth[i]),
            ↪  Quaternion.identity);//спавн шара в пространстве Unity
            myNewObject.GetComponent<Renderer>().material.color = new
            ↪  Color(info_plasma.color_val[i].b, info_plasma.color_val[i].g,
            ↪  info_plasma.color_val[i].r); //присваиваем значение цвета
    }
}
```

Для того, чтобы максимальное значение отображалось именно красным цветом, как указано в условии, а минимальное — синим, при назначении цвета шару нужно поменять местами значения red и blue. Тогда данные будут отображаться верно.

По условию задачи отображение данных может быть реализовано в оттенках серого.

Для данной схемы нам кроме максимального значения плазмы понадобится вычислить его минимальное значение. Дописываем класс Data в месте получения максимального значения.

```
public void start_connect()
    {
        //присваиваем максимуму и минимуму начальные значения плазмы
        max_value = plasma[0];
        min_value = plasma[0];
        //вычисляем максимальное и минимальное значение в плазме перебирая и сравнивая
        ↪  значения
        for (int i = 0; i < plasma.Length; i++)
        {
            if (plasma[i] > max_value) max_value = plasma[i];
            if (plasma[i] < min_value) min_value = plasma[i];
        }
    }
```

В эту функцию дописываем получение значения цвета в черно-белом формате. Используем для этого

```
       //вычисляем цвета для всех точек для примера
       for (int i = 0; i < plasma.Length; i++)
        {
            float value = map(plasma[i], min_value, max_value,0,1); //вычисляем
            ↪  значение цвета по словарю
            color_val[i] = new Color(value, value, value);//переводим численное
            ↪  значение в цветовую схему грескейл
        }
```

Теперь переписываем функцию Start в Data_Init для задания цвета объекту

```
public void Start()
{
    data_processing();//сериализация класса Data
    int lenght = info_plasma.latitude.Length;
    for (int i = 0; i < lenght; i++)//перебор всех данных из Data
    {
            var myNewObject = Instantiate(prefab, getXYZ(info_plasma.latitude[i],
            ↪  info_plasma.longitude[i], info_plasma.heigth[i]),
            ↪  Quaternion.identity);//спавн шара в пространстве Unity
            myNewObject.GetComponent<Renderer>().material.color = new
            ↪  Color(info_plasma.color_val[i].r, info_plasma.color_val[i].g,
            ↪  info_plasma.color_val[i].b); //присваиваем значение цвета
    }
```

На данный момент отображение данных приведет к сложному для восприятия результату, так как выводятся все данные, поэтому перейдем к реализации отображения среза ионосферы и уже на ее примерах отобразим данные в двух цветовых схемах.

Генерация поля ионосферы и отслеживание поворота ионосферы

Здесь возможны несколько вариантов для реализации:

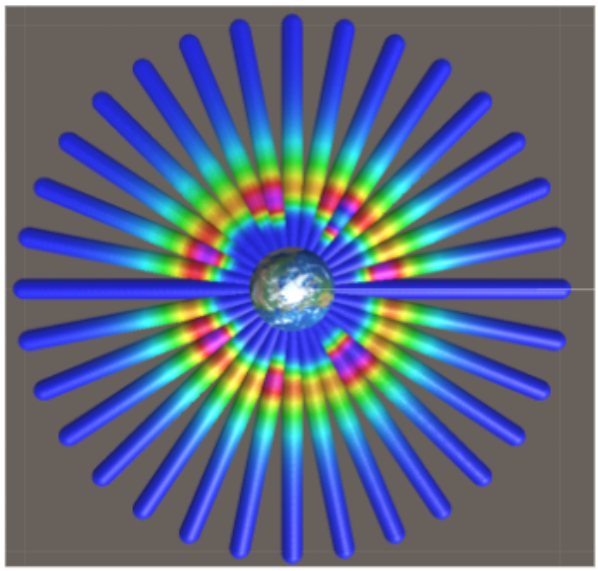
- вывод данных не кругами, а объектом (кубы/сферы и другие);
- реализация ионосферы через сферические треугольники;
- готовая 3D-модель.

Реализация методом дубликатов одиночного объекта Для этого варианта у нас уже был код (void Start() в Data.cs) , который нужно видоизменить так, чтобы отображалась только одна плоскость (срез) данных. Для этого добавим условие перед спавном шаров.

```
public Data info_plasma; //данные по ионосфере
    public Transform parent; //пустой объект в котором будут все части ионосферы
    public Transform prefab; //3D объект, которым будет отображаться точка с данными
    private int count=0; //подсчет объектов в срезе

    /*для отладки закомментировали код по получению данных с сервера
    public void data_processing()
    {
        Debug.Log("Comment: получение данных разбивка на массивы");
        info_plasma = JsonUtility.FromJson<Data>(PlayerPrefs.GetString("plasma_all"));
    }*/

    void data_processing()
    {
        string path = "Assets/Resources/json/series_nolat.json";

        StreamReader reader = new StreamReader(path);//Чтение json из файла data.json
        ↪  для отладки
        string data = reader.ReadToEnd();
        reader.Close();

        info_plasma = JsonUtility.FromJson<Data>(data);
        info_plasma.start_connect(); //вычисление максимальной плазмы и цветов данных
```

```
}


void Start()
{
    data_processing(); //проводим социализацию данных с сервера и запускаем
    ↪  подсчеты других величин

    for (int i = 0; i < info_plasma.len; i++)//перебираем все значения
    {
        if (info_plasma.longitude[i] == 0 || info_plasma.longitude[i] == -180)
        ↪  //условие для отображения данных именно по срезу, а не всех.
        ↪  Изначально выведем данные по 0 долготе
        {
            var myNewObject = Instantiate(prefab, getXYZ(info_plasma.latitude[i],
            ↪  info_plasma.longitude[i], info_plasma.heigth[i]),
            ↪  Quaternion.identity, parent);
            myNewObject.GetComponent<Renderer>().material.color = new
            ↪  Color(info_plasma.color_val[i].b, info_plasma.color_val[i].g,
            ↪  info_plasma.color_val[i].r); // задаем цвет объекту
            myNewObject.transform.name = count.ToString();//даем имя объекту
            ↪  равнозначное его порядковому номеру при выводе среза

            myNewObject.GetComponent<Get_Color>().level =
            ↪  (Convert.ToInt32(info_plasma.heigth[i]) - 90) / 5; //создаем
            ↪  переменную level для подсчета удаленности объекта от земли -
            ↪  понадобится позже
            count += 1; //считаем количество точек с нуля
        }

    }
}
```

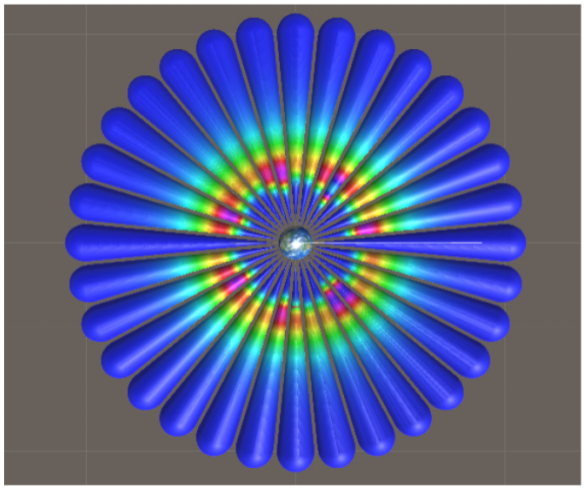
Можно немного увеличить шар и получим следующий результат. Как видно на картинке, цветовая схема работает без ошибок и отображает данные в тепловом виде от синего к красному.



В дальнейшем можно изменять размеры prefab для улучшения качества визуализации. А так же, можно программно скалировать объекты в зависимости от удаленности от Земли. Например, добавив строчку *myNewObject.transform.localScale* =

new Vector3(1f(level+1), 1f * (level + 1), 1f * (level + 1));* в код создания объектов уже можно получить решение, позволяющее управлять при построении визуализации размером промежутков.

Результаты выполнения такого видоизмененного кода представлены на картинке ниже.



Теперь сделаем отслеживание данных на такой модели при повороте ионосферы. Для этого напишем скрипт Get_Color(), который при повороте ионосферы будет обновлять данные о цвете.

```
public class Get_Color : MonoBehaviour
{
    public Data info_plasma; //для данных об ионосфере

    public int level; //указание уровня на котором находится объект
    public Color color; //цвет данных

    private int ii=0; //счетчик кадров

    public void Update()
    {
        if (ii == 20) info_plasma =
        ↪  transform.parent.gameObject.GetComponent<Data_Init>().info_plasma; //при
        ↪  запуске извлекаем данные об ионосфере
        if (ii % 10 == 0 && ii > 50) //через 50 кадров от старта с интервалов в 10
        ↪  кадров будет происходить обновление данных
        {
             color = GetValue(); //получение значения цвета данных
             transform.GetComponent<Renderer>().material.color = new Color(color.b,
             ↪  color.g, color.r); //установка цвета
        }
        ii += 1;//прибавляем кадры
    }


    Color GetValue()
    {
        float[] nn = getLatLng(transform.position);//получаем текущий longitude и
        ↪  latitude, чтобы в дальнейшем сравнивать по этим значениям
        color = new Color(0,0,0); //назначаем черный цвет, который будет означать, что
        ↪  данные не получены
```

```
        for (int i = 0; i < info_plasma.len; i++) //перебираем весь массив значений
        ↪  ионосферы
        {
            //сопоставляем текущий longitude и latitude с значениями сервера
            //11.25f - интервал с которыми приходят текущие данные и который нам
            ↪  необходимо указать, если текущая позиция окажется между данными
            //высота приходит с интервалом в 5еденц и начинается с 90, поэтому ее
            ↪  можно вычислить зная уровень объекта, который мы задаем при генерации
            //уровень никогда не поменяется, так как объкт не может перемещаться по
            ↪  высоте
            if (info_plasma.longitude[i] >= nn[1] - (11.25f/2) &&
            ↪  info_plasma.longitude[i] <= nn[1] + (11.25f / 2) &&
                info_plasma.latitude[i] >= nn[0] - (11.25f / 2) &&
                ↪  info_plasma.latitude[i] <= nn[0] + (11.25f / 2) &&
                info_plasma.heigth[i] == 90 + (5 *
                ↪  transform.GetComponent<Get_Color>().level))
            {
                color = info_plasma.color_val[i]; //назначаем цвет
                break;//завершаем цикл, так как точка с данными одна и смысла
                ↪  перебирать значения дальше нет
            }
        }
        return color;
    }
    //перевод координат xyz в latlong
    float[] getLatLng(Vector3 pointXYZ)
    {

        float r = Mathf.Sqrt(pointXYZ.x * pointXYZ.x + pointXYZ.y * pointXYZ.y +
        ↪  pointXYZ.z * pointXYZ.z);
        float lat = Mathf.Asin(pointXYZ.z / r) * 180 / Mathf.PI;
        float lon = Mathf.Atan2(pointXYZ.y, pointXYZ.x) * 180 / Mathf.PI;
        float[] latlong = { lat, lon, r };
        return latlong;
    }
}
```

Закрепляем данный скрипт на объекте точки данных ионосферы. И теперь при повороте ионосферы значения меняются в зависимости от ее положения.

Реализация генеративным методом

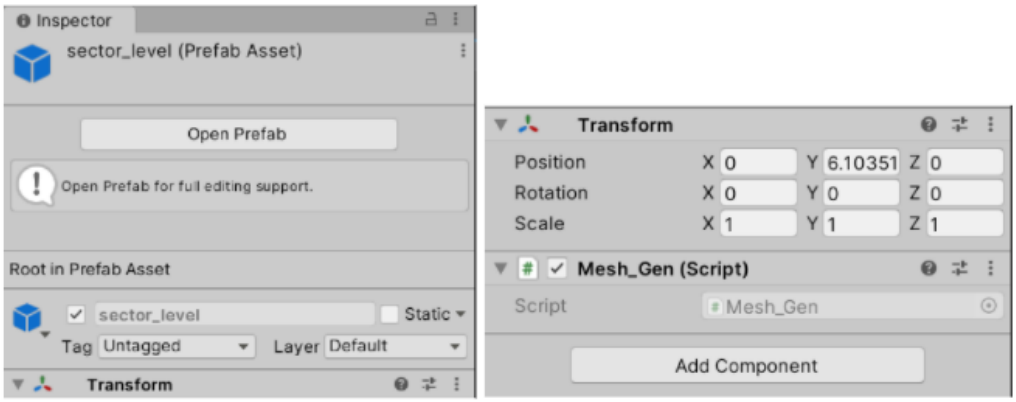
Можно реализовать ионосферу сегментированием и подсчетом значений внутри сегментов. Сегменты можно генерировать с помощью треугольников, исходящих из ядра земли и с небольшим смещением по вертикали, для правильного отображения. Таким методом можно давать пользователю возможность вводить значения количества сегментов в срезе и делений внутри каждого сегмента.

В данном случае, нам понадобится следующий код для генерации треугольников. Можно взять стандартный скрипт для генерации кубов и переделать его под наши цели. Назовем данный скрипт Mesh_Gen().

```
public class Mesh_Gen : MonoBehaviour
{
 //треугольники будут равнобедренные, но разного размера
    //поэтому требуется передавать его высоту и длину нижней части
    public void Gen(float R=1, float height=2)
    {
        //Создание меша, прикрепление к нему дефолтного материала
        MeshRenderer meshRenderer = gameObject.AddComponent<MeshRenderer>();
```

```
        meshRenderer.sharedMaterial = new Material(Shader.Find("Standard"));

        MeshFilter meshFilter = gameObject.AddComponent<MeshFilter>();

        Mesh mesh = new Mesh();

        //задаем сетку координат, две две вершины будут занулены
        Vector3[] vertices = new Vector3[4]
        {
            new Vector3(0, 0, 0),
            new Vector3(0, 0, 0),
            new Vector3(R, height/2,  0),
            new Vector3(R, -height/2,  0)
        };
        mesh.vertices = vertices;

        int[] tris = new int[6]
        {
            // lower left triangle
            0, 2, 1,
            // upper right triangle
            2, 3, 1
        };
        mesh.triangles = tris;

        //создаем сетку нормалей
        Vector3[] normals = new Vector3[4]
        {
            -Vector3.forward,
            -Vector3.forward,
            -Vector3.forward,
            -Vector3.forward
        };
        mesh.normals = normals;

        //указываем базовые координаты текстуры меша
        Vector2[] uv = new Vector2[4]
        {
            new Vector2(0, 0),
            new Vector2(1, 0),
            new Vector2(0, 1),
            new Vector2(1, 1)
        };
        mesh.uv = uv;

        meshFilter.mesh = mesh;
    }
}
```

Создадим Prefab из пустого объекта (Create Empty) и добавим в компоненты созданный скрипт Mesh_Gen(). Конвертируем созданный объект в префаб.

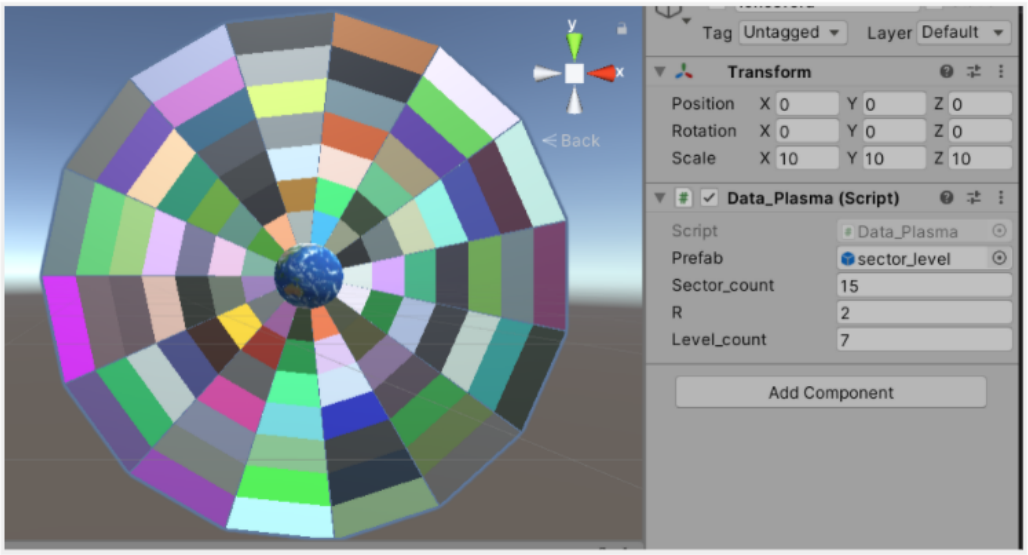


Создадим скрипт Data_Plasma(), который будет генерировать окружность из треугольников.

```
public class Data_Plasma : MonoBehaviour
{
    public GameObject prefab;//поместим сюда пустой объект(selector_level) с
    ↪  привязанным скриптом генерации меша

    public int sector_count = 32; //количество секторов
    public int R = 5; //величина сектора
    public int level_count = 10; //деления внутри сектора

    float trig_base = 0;
    float rot_step=12;


    public void Start()
    {
        rot_step = 360 / sector_count; //рассчитываем расстояние между секторами
        trig_base = (R * Mathf.PI * 2) / sector_count; //величина ширины треугоника
        ↪  по основанию
        generator();
        transform.localScale = new Vector3(10, 10, 10);
    }


    void generator()
    {
        //генерируем все кусочки среза ионосферы
        for (int j = 1; j <= level_count+1; j++)
        {
            for (int i = 0; i <= sector_count; i++)
            {

                GameObject new_obj = Instantiate(prefab); //создаем пустой объект
                ↪  (будущая часть сектора)
                new_obj.transform.SetParent(transform); //назначаем ему родительский
                ↪  объект чтобы все кусочки ионосферы были в одном большом объекте
                new_obj.transform.Rotate(0, 0, rot_step * i); //поворачиваем объект
                ↪  для того, чтобы треугольники образовывали окружность

                //на текущий момент все объекты лежат в одной плоскости и перекрывают
                ↪  друг друга
                //чтобы пользователю были видны все объекты немного сместим их друг
                ↪  под друга
```

```
                float z = new_obj.transform.position.z + 0.001f * j;
                new_obj.transform.position = new Vector3(0, 0, z);

              new_obj.name = (level_count-j).ToString(); //даем имя объекту по его
               ↪  номеру сегмента
            new_obj.GetComponent<Get_Color>().sector = i; //создаем переменную level
             ↪  для подсчета удаленности объекта от земли
                new_obj.GetComponent<Get_Color>().level = j - 1; //создаем переменную
                 ↪  level для подсчета удаленности объекта от земли


                new_obj.GetComponent<Mesh_Gen>().Gen(R * j, trig_base * j);//задаем
                 ↪  объекту форму треугольника

                //временно дает рандомный цвет каждому кусочку, чтобы была видна
                 ↪  сегментация
                new_obj.GetComponent<Renderer>().material.color =
                 ↪  UnityEngine.Random.ColorHSV();
            }
        }
    }
}
```

Пример сгенерированной ионосферы данным скриптом ниже.



Получение цвета сектора должно теперь происходить по среднему значению точек данных, которые попали в сектор. Но так как теперь у нас не одиночные объекты, а сектор, то его позицию будем вычислять через инструмент Matrix4x4. Матрица преобразования может выполнять произвольные линейные трехмерные преобразования (например, перенос, поворот, масштаб, сдвиг и т. д.) И перспективные преобразования с использованием однородных координат.

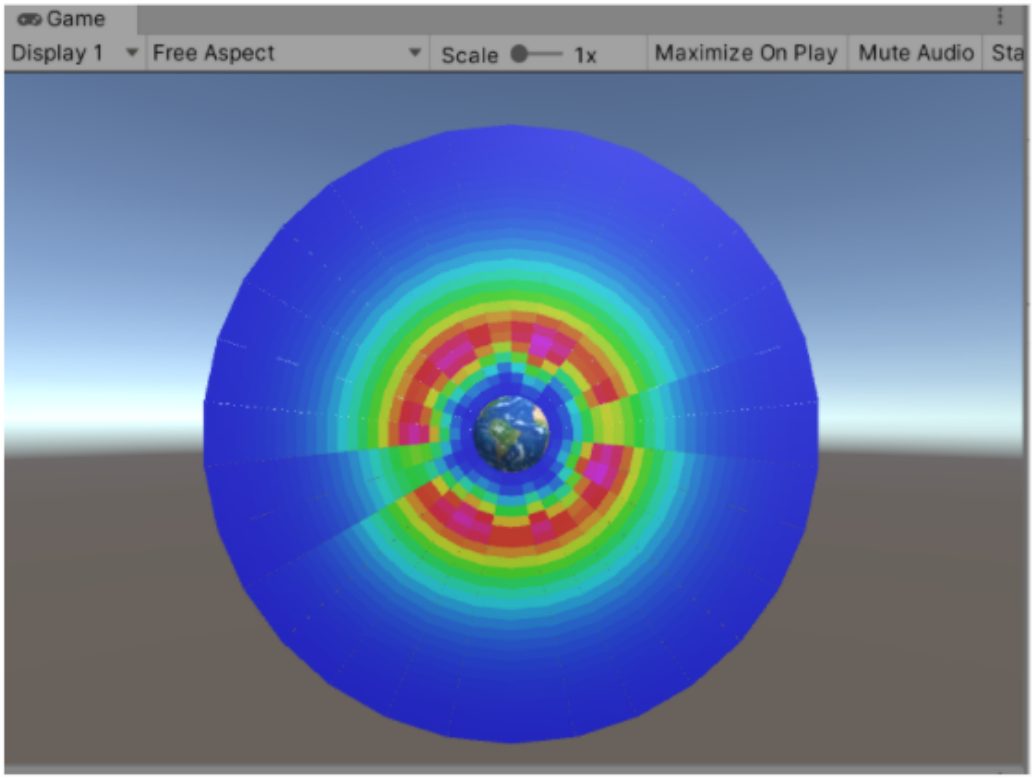
```
public class Get_Color: MonoBehaviour
{

    private GameObject center_figure; //родительский объект для всех кусочкоув
     ↪  ионосферы
    public Data info_plasma; //значения ионосферы с сервера/файла
```

```
int counter = 0; //счетчик кадров
static float max_value;//максимальное значение plasma

public int sector_count;//общее значение секторов в ионосфере
public int level_count;//общее значение уровней в секторе

public int sector; //текущий сектор
public int level; //текущий уровень в секторе

//начальные значения
public float latitude;
public float longitude;
private Vector3 position;

//текщие значения
public float new_latitude;
public float new_longitude;



void Start()
{
    center_figure = transform.parent.gameObject; //получаем объект на котором
    ↪  задаются параметры ионосферы
    info_plasma = center_figure.GetComponent<Data_Plasma>().info_plasma;
    ↪  //записываем данные по плазме в переменную
    level_count = center_figure.GetComponent<Data_Plasma>().level_count; //число
    ↪  делений в секторе
    sector_count = center_figure.GetComponent<Data_Plasma>().sector_count; //число
    ↪  секторов в окружности


    //1000 - это максимальная высота по условию, а 90 - минимальная
    level = ((1000-90) / level_count) * (level-1);//вычисляем шаг по высоте и
    ↪  находим текущее значение

    //вычисляем стартовую позицию
    latitude = 0;
    longitude = (360 / sector_count) * sector;
    position = getXYZ(latitude, longitude);

    max_value = center_figure.GetComponent<Data_Plasma>().info_plasma.max_value;
}



float GetValue()
{
    int count = 0; //для количества точек попавших в сегмент
    float value = 0; //для суммы значений plasma в сегменте

    for (int i = 0; i < info_plasma.len; i++)
    {
        //находим точки попавшие в сегмент используя интервалы
        //11.25 - это интервал с которым приходят данные с сервера
        if ((info_plasma.longitude[i] >= new_longitude - 11.25f) &&
            (info_plasma.longitude[i] <= new_longitude + 11.25f) &&
            (info_plasma.latitude[i]  >= new_latitude - 11.25f) &&
            (info_plasma.latitude[i] <= new_latitude + 11.25f) &&
            ((info_plasma.heigth[i] - 90) >= level - ((1000 - 90) / level_count))
            ↪  &&
```

```
            ((info_plasma.heigth[i] - 90) <= level + ((1000 - 90) / level_count)))
        {
            value += info_plasma.plasma[i];
            count += 1;
        }
    }
    return value / count;
}

void Update()
{

    if ((counter + sector) % 100 == 0) //обновление сектора целиком
    {
        float value = 0; //среднее значение плазмы в сегменте

        //для определения смещения ионосферы используем Matrix4x4
        Vector3 angle = center_figure.transform.eulerAngles;
        Quaternion rotation = Quaternion.Euler(angle.x, angle.y, angle.z);
        Matrix4x4 m = Matrix4x4.Rotate(rotation);
        Vector3 new_position = m.MultiplyPoint3x4(position);

        //вычисляем текущие latitude и longitude
        float[] latlong = getLatLng(new_position);
        new_latitude = latlong[0];
        new_longitude = latlong[1];

        //определяем среднее значение плазмы
        value = GetValue();

        //определяем цвет для среднего значения plasma
        Color color = getColor(value);
        //задем объекту полученный цвет
        transform.GetComponent<Renderer>().material.color = new Color(color.b,
        ↪  color.g, color.r);
    }

    counter += 1;//счетчик кадров
}

//для определения координат в xyz по latlong
Vector3 getXYZ(float lat, float lon, float radius = 100)
{
    lat = lat * Mathf.PI / 180;
    lon = lon * Mathf.PI / 180;

    float x = radius * Mathf.Cos(lat) * Mathf.Cos(lon);
    float z = radius * Mathf.Sin(lat);
    float y = radius * Mathf.Cos(lat) * Mathf.Sin(lon);

    return new Vector3(x, y, z);
}

//для определения latlong по xyz
float[] getLatLng(Vector3 pointXYZ)
{
    float r = Mathf.Sqrt(pointXYZ.x * pointXYZ.x + pointXYZ.y * pointXYZ.y +
    ↪  pointXYZ.z * pointXYZ.z);
    float lat = Mathf.Asin(pointXYZ.z / r) * 180 / Mathf.PI;
    float lon = Mathf.Atan2(pointXYZ.y, pointXYZ.x) * 180 / Mathf.PI;
```

```
        float[] latlong = { lat, lon, r };
        return latlong;
    }

    //определение цвета для значениея plasma
    Color getColor(float value)
    {
        float step = 180 / (float)(max_value * 0.4);
        float color = (step * value)/360;
        return Color.HSVToRGB(color, 1, 0.7f);
    }

}
```

При запуске проекта получим результат, представленный на картинке ниже, где значения ионосферы обновляются в зависимости от ее поворота.



Реализация методом цельной 3d-модели ионосферы

Создание ионосферы по готовой 3D-модели окружности, которая разбита по кусочкам на сегменты и уровни заранее.

В данном методе будет использоваться тот же скрипт, что и в предыдущем, для определения точек данных внутри области, подсчета среднего значения и вывода цвета на фигуру. Только в этот раз необходимо изначально задать фиксированные sector_count и level_count, по их числу в созданной 3D модели.

Реализация визуализации данных ионосферы в зависимости от угла наблюдения пользователя

Расчет и реализация алгоритма визуализации данных в зависимости от угла наблюдения пользователя — от того, на какой сектор геокоординат наведена в данный момент камера мобильного устройства.

Переводя камеру или прокручивая глобус мы меняем сектор геокоординат, поскольку данные об ионосфере привязаны к определенным геокоординатам, то и визуализация при смене точки наблюдения должна меняться.

Решение

Пересчет значений ионосферы уже был реализован выше, он был привязан к вращению ионосферы, осталось только закрепить ионосферу к камере.

В Unity существует довольно простое решение данной задачи. Создаем скрипт Rotation, где будет публичная переменная для отслеживаемого объекта. И для отслеживания угла просмотра будем использовать функцию LookAt(), которая позволяет автоматически определить с какой точки смотрит на объект камера.

```
 public class Rotation : MonoBehaviour
{
    public Transform target;//привязываем ионосферу со сцены

    void Update()
    {
        // Rotate the camera every frame so it keeps looking at the target
        transform.LookAt(target); //получаем угол наблюдения пользователя
        float xangle = transform.rotation.x;
        float yangle = transform.rotation.y;
        float zangle = transform.rotation.z;
```

```
        transform.Rotate(xangle+90, yangle, zangle);//Разворачиваем объект
        ↪ относительно угла наблюдения пользователя

    }
}
```

Теперь ионосфера привязана к Земле и углу поворота камеры. Что позволяет нам постоянно автоматически получать значения позиции для обновления цвета всех частей ионосферы.

Расчет и разработка алгоритма визуализации полета спутников

Из данных с сервера необходимо извлечь координаты полета и текущее местоположение спутников системы G. Соотнести системы координат земли и спутников и настроить интерполированный плавный полет спутников.

Сделать возможность просмотра истории полета каждого спутника в виде дуг, а не прямых отрезков.

Решение

Первым шагом мы сделаем десериализацию полученных данных о спутниках в словарь в Unity. Для этого нам понадобится расширение Newtonsoft.Json, которое можно установить через пакетный менеджер. Находим в нем JSON.NET for Unity и импортируем себе в проект.

Для десериализации создадим новый скрипт Stat_Init. Так же, как и ранее с данными об атмосфере, поместим данные с сервера в файл stat.json для более быстрой

отладки. Данные по маршруту движения спутников будем сохранять в отдельный массив и переводить в формат понятный Unity, а именно Vector3.

```
using Newtonsoft.Json;//используем расширение, которое установили

public class Stat_Init : MonoBehaviour
{
    public Dictionary<string, List<List<float>>> Stat;//словарь для хранения данных с
    ↪  сервера

    public GameObject prefab;//объект для спутника

    //сразу десериализовать объект в вектора не выйдет, поэтому создаем отдельный
    ↪  массив Vector3
    //в нем будем хранить путь для каждого спутника
    public Vector3[] value_route;


    void Start()
    {
        //временно читаем данные из файла, для отладки
        string path = "Assets/Resources_js/json/stat.json";
        StreamReader reader = new StreamReader(path);
        string data = reader.ReadToEnd();
        reader.Close();

        //проводим десериализацию полученных данных в ранее созданный словарь
        Stat = JsonConvert.DeserializeObject<Dictionary<string,
        ↪  List<List<float>>>>(data);

        //для всех значений из словаря,т. е. для всех спутников
        foreach (KeyValuePair<string, List<List<float>>> sputnik in Stat)
        {
            //по условию задачи все спутники не нужны, а только системы Glonas
            if (sputnik.Key.ToString().IndexOf('G') != -1)//ищем систему G в названии
            ↪  спутников
            {
                value_route = new Vector3[96];//длина маршрута с сервера всегда равна
                ↪  96 точкам, поэтому сразу указываем размерность

                //для нужных спутников перебираем значения пути
                for (int i = 0; i < 96; i++)
                {
                    //переводим их из float[] в Vector3[] для дальнейшего
                    ↪  использования
                    //100000 в нашем случае для сопоставления наших условных единиц в
                    ↪  Unity и пришедших у.е. с сервера
                    value_route[i] = new Vector3(sputnik.Value[i][0] / 1000000,
                    ↪  sputnik.Value[i][1] / 1000000, sputnik.Value[i][2] / 1000000);
                }

                GameObject new_obj = Instantiate(prefab, value_route[0],
                ↪  Quaternion.identity); //создаем спутник на сцене в первой его
                ↪  точке
                new_obj.name = sputnik.Key.ToString(); //присваеваем спутнику его имя
                ↪  из данных с сервера
                new_obj.GetComponent<Stat>().value_route = value_route; //записываем
                ↪  его маршрут движения

            }
```

```
        }
    }
}
```

Теперь настроим отображение пути спутника. Для этого добавим в скрипт еще одну переменную, которая будет хранить объект отображения пути.


```
 public GameObject point; //объект для маршрута спутника

//И в функции Start() после передачи пути объекту настроим объект LineRenderer,
↪  который в Unity отвечает за отрисовку линий
//.....
                new_obj.GetComponent<Stat>().value_route = value_route; //записываем
                ↪  его маршрут движения

 //LineRenderer lineRenderer = point.GetComponent<LineRenderer>(); //для отрисовки
 ↪  маршрута используем LineRenderer
                 point.GetComponent<LineRenderer>().SetPositions(value_route);
                 ↪  //передаем значения векторов в маршрут для его отрисовки
                 GameObject my_obj = Instantiate(point, value_route[0],
                 ↪  Quaternion.identity, new_obj.transform);//создаем объект маршрута
                 ↪  движения спутника
...... //
```

Теперь необходимо создать префабы для спутника и его пути.

Реализуем пока отображение одной модели спутника во всех местах. Это может быть элементарный цилиндр или другая подобная фигура. К которой необходимо прикрепить скрипт, для хранения данных о пути.

Поэтому создаем скрипт, который необходимо привязать к модели спутника. Назовем его Stat.

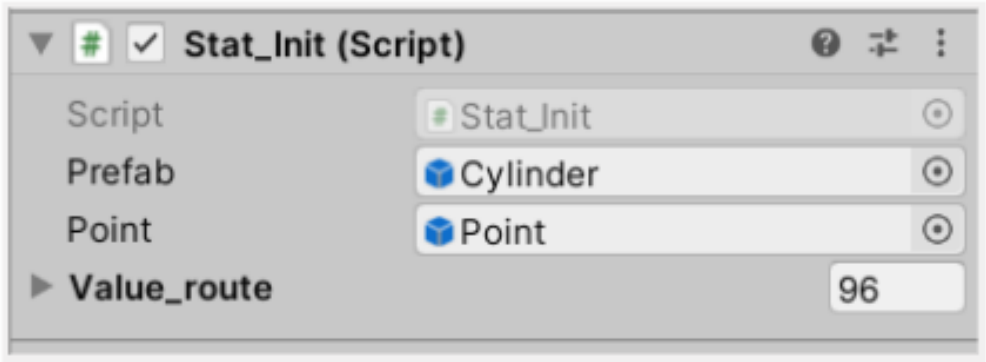
```
 public class Stat : MonoBehaviour
{
    public Vector3[] value_route;//значения пути спутника
}
```

Пока он короткий, но в дальнейшем мы расширим его для создания анимации движения.

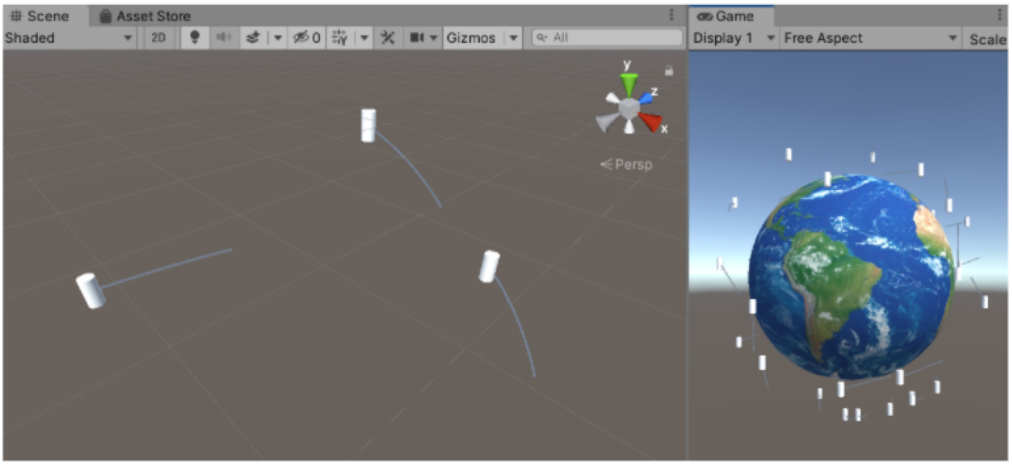
Теперь создаем объект для отображения пути спутника. Для пути создадим пустой объект (Create Empty) и прикрепим к нему Line Render, внутри которого укажем будущую ширину линий и прикрепим материал для визуализации пути.

Материал, следуя документации Unity о Line Render следует прикреплять из систем частиц, поэтому выбираем Default Terrain Specular. Назовем созданный объект «Point» и конвертируем в префаб.

Закрепляем созданные префабы в настройках скрипта, который прикрепляем к любому объекту на сцене



Теперь можно запустить проект и на сцене отобразится 32 одинаковых спутника с их траекториями.



Добавим анимацию движения спутников.

Для плавной анимации будем использовать метод FixedUpdate(), который выполняется по стандарту 50 раз в секунду, тогда как Update() зависит от производительности устройства пользователя и может меняться в зависимости от нагрузки. И MoveTowards(), который осуществляет плавную анимацию от одной позиции к другой.

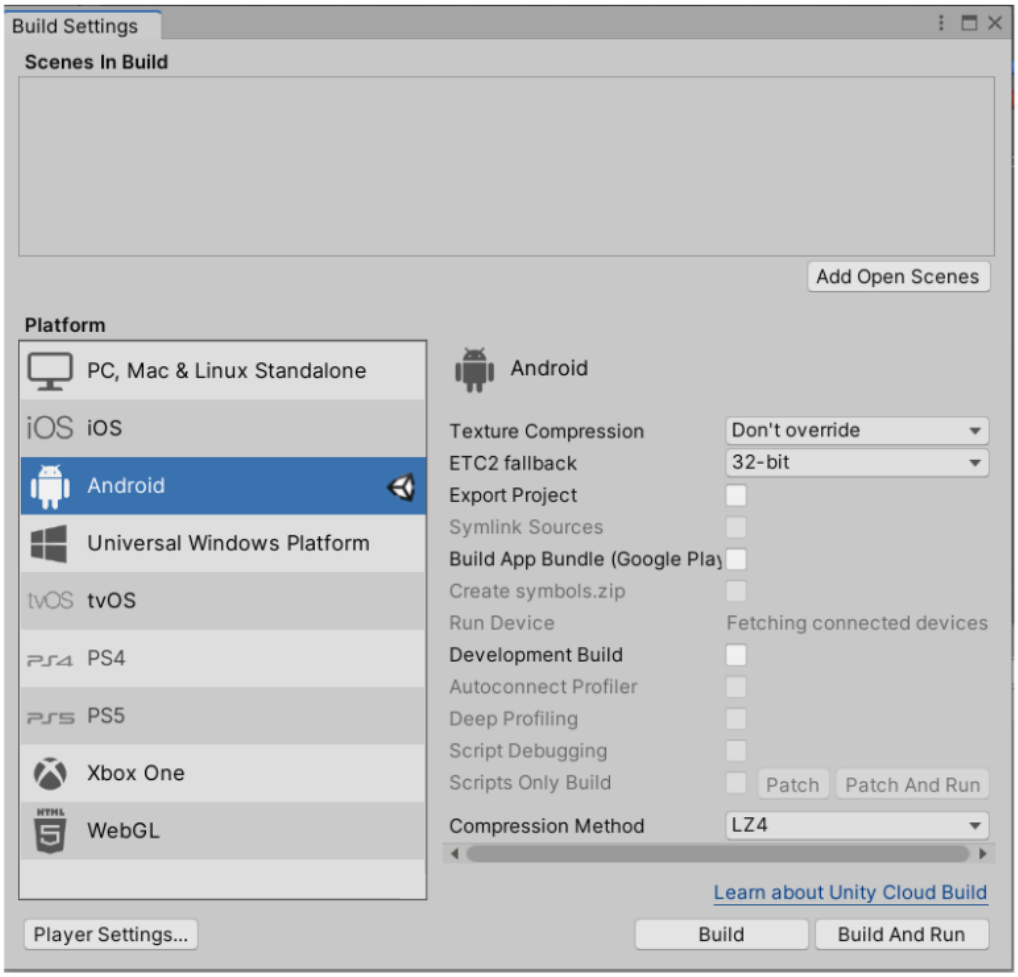
В Stat() добавим метод FixedUpdate() и счетчик для плавной анимации.

```
 public class Stat : MonoBehaviour
{
    public Vector3[] value_route;//значения пути спутника
    private int point=0;//счетчик обновлений

    private void FixedUpdate()
    {
        transform.position = Vector3.MoveTowards(value_route[point % 95],
        ↪  value_route[point % 95 + 1], 1); //создаем движение от первой точки к
        ↪  последней путем перемещения по массиву
        point += 1;//счетчик кадров для нахождения текущей позиции
    }
}
```

В результате выполнения данного кода модели спутников будут двигаться по пути с 0 точки в 95, достигая конца перемещаться обратно на 0 позицию и вновь двигаться к конечной точке по маршруту.

Сборка всех частей приложения в исполняемый файл

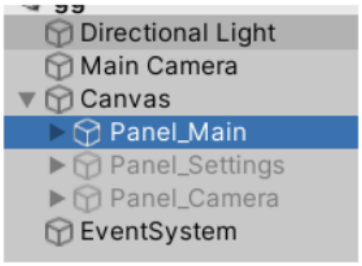
Требований по целевому устройству нет, поэтому можно собрать приложение под любую платформу. Для примера приложение собиралось под Android. Для этого в Unity меняем платформу на Android.



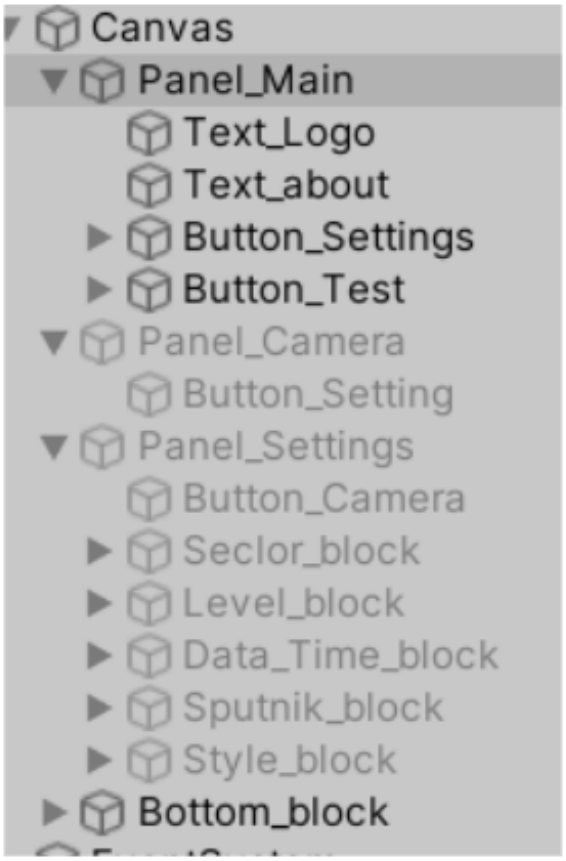
Сразу добавим дополненную реальность в проект (пункт 2.3), чтобы производить отладку с использованием координат относительно используемого маркера.

Приложение будет вертикальной формы, поэтому зададим в настройках дисплея размерность 1920 на 1080. Переходим к созданию интерфейса.

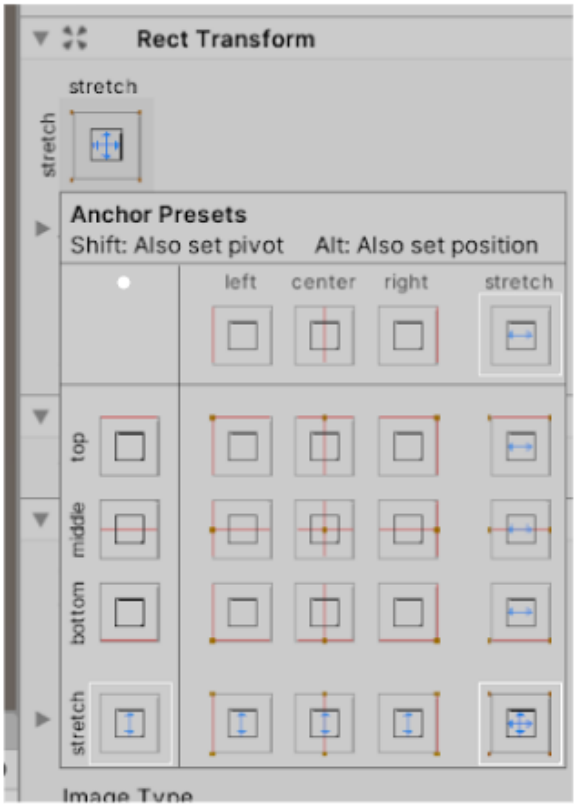
Создаем для UI Canvas и отрисовываем несколько Panel по вайфреймам от дизайнера. Даем им названия соответствующие содержимому.



Внутри каждой панели создаем элементы UI: тексты, кнопки, слайдеры и блоки для ввода данных. Строку состояния вынесем в отдельный блок и программно будем ей менять текст и цвет.

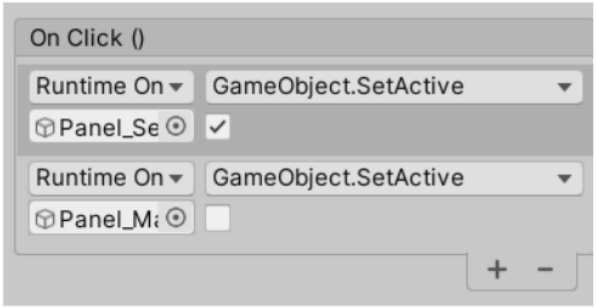


Для адаптивности приложения необходимо правильно настаивать привязывания всех компонентов UI относительно родительских объектов.



После создания всех кнопок настраиваем переходы между панелями. Для этого не будем использовать скрипт, а воспользуемся настройкой свойств.

У каждой кнопки есть событие OnClick(), которое срабатывает при нажатии на кнопку. Создаем внутри два блока, где указываем первым панель, которую необходимо отобразить, а вторым — текущую панель, на которой и находится кнопка. Например, кнопка настройки параметров ионосферы, должна отобразить Panel_Settings и скрыть Panel_Main.



Проводим такие действия со всеми кнопками на сцене.

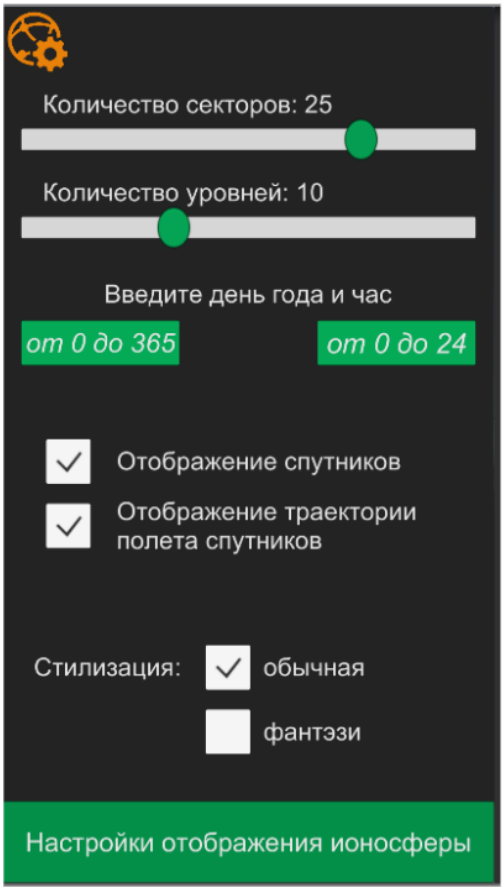
Теперь напишем скрипт получения информации, которую будет вводить пользователь на странице Panel_Settings.

Для ввода информации пользователем используются объекты InputField, Slider и Toggle. Создаем скрипт Get_Settings(), в котором обращаемся ко всем таким объектам для считывания их значений.

```
//создаем переменные для всех параметров
public int sector_count = 32; //количество секторов в ионосфере
    public int level_count = 10; //количество уровней в секторе


    //переменные для запроса данных с сервера за определенное время
    public int day = 0; //день года от 1 до 365
    public int hour = 0; //час от 1 до 24

    //переменные спутников
    //public bool sputniks = true; //отображение моделей спутников
    //public bool s_traectory = true; //отображение траектории спутников

    public string style;//для типа стилизации


    //переменные для привязки объектов
    public InputField day_p;
    public InputField hour_p;
    public Slider count_sector;
    public Text text_count_sector;
    public Slider count_level;
    public Text text_count_level;
    public Toggle sputniks_p;
    public Toggle s_traectory_p;
    public ToggleGroup toggleGroup_style_p;

    //строка состояния внизу окна
    public Text status;
```

```
//при старте задаем базовые  параметры
void Start()
{
    //задаем параметры размерности массивам моделей
    Default_model = new GameObject[32];
    Fantasy_model = new GameObject[32];

    //ищем модели спутников в папках
    var go1 = Resources.LoadAll("Model1/", typeof(GameObject));
    var go2 = Resources.LoadAll("Model2/", typeof(GameObject));

    //записываем в массив найденные объекты
    for (int i = 0; i < 32; i++)
    {
        Default_model[i] = (GameObject)go1[i];
        Fantasy_model[i] = (GameObject)go2[i];
    }

    style = "0";//всегда при старте будет обычная стилизация
}


//создаем функцию считывания введе'нных пользователем данных
public void get_value()
{
    //проверка дня
    if (day_p.text!="" && Int32.Parse(day_p.text) != day &&
    ↪  Int32.Parse(day_p.text) <= 365 && Int32.Parse(day_p.text) > 0)
    {
        day = Int32.Parse(day_p.text);
        Up_plasma_data = true;
    }
    else day = 0;
    //проверка часа
    if (hour_p.text != "" && Int32.Parse(hour_p.text) != hour &&
    ↪  Int32.Parse(hour_p.text) <= 24 && Int32.Parse(hour_p.text) > 0)
    {
        hour = Int32.Parse(hour_p.text);
    }
    else hour = 0;
}
```

Теперь на экране можно взаимодействовать со всеми объектами и их значения записываются в переменные.



Далее добавляем все сконструированные 3D-модели в проект и вместо загрузки модели одного спутника, настраиваем каждому спутнику его модель. Реализуем переключение стилизации.

В скрипт Get_Settings() добавляем переменные и меняем функцию Start().

```
//для стилизации находим все объекты на разные стили
public GameObject[] Default_model;
    public GameObject[] Fantasy_model;


    void Start()
    {
        //задаем параметры размерности массивам моделей
        Default_model = new GameObject[32];
        Fantasy_model = new GameObject[32];

        //ищем модели спутников в папках
        var go1 = Resources.LoadAll("Model1/", typeof(GameObject));
        var go2 = Resources.LoadAll("Model2/", typeof(GameObject));

        //записываем в массив найденные объекты
        for (int i = 0; i < 32; i++)
        {
            Default_model[i] = (GameObject)go1[i];
            Fantasy_model[i] = (GameObject)go2[i];
        }

        style = "default";//всегда при старте будет обычная стилизация
        transform.GetComponent<Stat_Init>().Start_Sput(Default_model);
```

```
        Planet = Default_planet;
    }
```

Теперь меняем скрипт обработки данных по спутникам и их визуализацию Stat_Init(). В Stat_Init() в public void Init_Sput(bool error) было добавлено условие на проверку пришедших ошибок с сервера, если данные не удалось получить, то считываем их из записанного файла. Таким образом пользователь в любом случае сможет увидеть как именно работает приложение. Закрепим данный скрипт на объекте EventSystem.

```
using System.Collections.Generic;
using UnityEngine;
using Newtonsoft.Json;
using System.IO;

public class Stat_Init : MonoBehaviour
{
    public Dictionary<string, List<List<float>>> Stat;//словарь для хранения данных с
    ↪  сервера

    public GameObject point; //объект для маршрута спутника


    //сразу десериализовать объект в вектора не выйдет, поэтому создаем отдельный
    ↪  массив Vector3
    //в нем будем хранить путь для каждого спутника
    public Vector3[] value_route;

    //инициализация данных
    public void Init_Sput( bool error)
    {
        if (error)//не удалось получить данные
        {
            //чтение данных из файла, так как с сервера идут ошибки
            string path = "Assets/Resources_js/json/stat.json";
            StreamReader reader = new StreamReader(path);
            string data = reader.ReadToEnd();
            reader.Close();

            //проводим десериализацию полученных данных в ранее созданный словарь
            Stat = JsonConvert.DeserializeObject<Dictionary<string,
            ↪  List<List<float>>>>(data);
        }
        else //данные получены с сервера
        {
            //проводим десериализацию полученных данных в ранее созданный словарь
            Stat = JsonConvert.DeserializeObject<Dictionary<string,
            ↪  List<List<float>>>>(PlayerPrefs.GetString("sputniks"));
        }

        //создадим пустой объект на сцене, как родительский для всех спутников
        parent = Instantiate(new GameObject(),
        ↪  GameObject.Find("EventSystem").GetComponent<Get_Settings>().Planet.transform);

        //parent.transform.SetParent(GameObject.Find("EventSystem"
        ↪  ).GetComponent<Get_Settings>().Planet.transform);

        parent.name = "parent_sputniks";
        parent.tag = "parent_sputniks";
```

```
    }

    public GameObject parent;

    public void Start_Sput(GameObject[] prefabs) {

        int i_obj = 0; //счетчик отображенных спутников

        //для всех значений из словаря,т. е. для всех спутников
        foreach (KeyValuePair<string, List<List<float>>> sputnik in Stat)
        {
            //по условию задачи все спутники не нужны, а только системы Glonas
            if (sputnik.Key.ToString().IndexOf('G') != -1)//ищем систему G в названии
            ↪  спутников
            {
                value_route = new Vector3[96];//длина маршрута с сервера всегда равна
                ↪  96 точкам, поэтому сразууказываем размерность

                //для нужных спутников перебираем значения пути
                for (int i = 0; i < 96; i++)
                {
                    //переводим их из float[] в Vector3[] для дальнейшего
                    ↪  использования
                    //100000000 в нашем случае для сопоставления наших условных единиц
                    ↪  в Unity и пришедших у.е. с сервера
                    value_route[i] = new Vector3(sputnik.Value[i][0] / 100000000,
                    ↪  sputnik.Value[i][1] / 100000000, sputnik.Value[i][2] /
                    ↪  100000000);

                }

                Vector3 locPos =
                ↪  GameObject.Find("MultiTarget").transform.localPosition;

                GameObject new_obj = Instantiate(prefabs[i_obj], value_route[0]+
                ↪  locPos, Quaternion.identity,parent.transform); //создаем спутник
                ↪  на сцене в первой его точке
                new_obj.transform.localPosition = value_route[0];
                new_obj.name = sputnik.Key.ToString(); //присваеваем спутнику его имя
                ↪  из данных с сервера
                new_obj.AddComponent<Stat>().value_route = value_route; //записываем
                ↪  его маршрут движения
                new_obj.tag = "sput";//задаем тег спутнику

                point.GetComponent<LineRenderer>().SetPositions(value_route);
                ↪  //передаем значения векторов в маршрут для его отрисовки
                GameObject my_obj = Instantiate(point, value_route[0],
                ↪  new_obj.transform.localRotation, new_obj.transform);//создаем
                ↪  объект маршрута движения спутника
                i_obj +=1;
            }
        }
    }
}
```

Меняем скрипт Stat(), который прикреплен ко всем спутникам (добавляется программно), для нахождения актуальной позиции при анимации.

```
using UnityEngine;
```

```
public class Stat : MonoBehaviour
{
    public Vector3[] value_route;
    private int point = 0;

    //анимация спутников и их перемещение относительно координат маркера AR
    private void FixedUpdate()
    {
        //ищем позицию маркера, чтобы сместить спутник относительно нее
        Vector3 locPos = GameObject.Find("MultiTarget").transform.localPosition;

        transform.position = Vector3.MoveTowards(value_route[point % 95] + locPos,
        ↪  value_route[point % 95 + 1] + locPos, 1);//анимируем движение спутника
        point += 1;//счетчик кадров для нахождения текущей позиции
    }
}
```

Теперь настраиваем загрузку ионосферы по данным указанным пользователем в настройках. Для этого изменим скрипт Data_Plasma(). В данном скрипте была создана дополнительная функция на получение данных из файла. Прикрепляется данный скрипт на родительский объект ионосферы (Ionosfera).

```
using System.IO;
using UnityEngine;

public class Data_Plasma : MonoBehaviour
{
    public Data info_plasma; //данные по ионосфере

    public GameObject prefab;//поместим сюда пустой объект с привязанным скриптом
    ↪  генерации меша

    public int sector_count; //количество секторов
    public int R = 5; //величина сектора
    public int level_count; //деления внутри сектора

    float trig_base;
    float rot_step;

    //получаем данные с сервера
    public void data_processing()
    {
        Debug.Log("Comment: получение данных разбивка на массивы");
        info_plasma = JsonUtility.FromJson<Data>(PlayerPrefs.GetString("plasma_all"));
        info_plasma.start_connect();
    }

    //функция с чтением из файла по тестовым данным
    public void data_processing_test()
    {
        string path = "Assets/Resources_js/json/series_nolat.json";

        StreamReader reader = new StreamReader(path);//Чтение json из файла data.json
        ↪  для отладки
        string data = reader.ReadToEnd();
        reader.Close();

        info_plasma = JsonUtility.FromJson<Data>(data);
        info_plasma.start_connect();
    }
```

```
public void Start_Plasma(int s_count, int l_count)
{
    sector_count = s_count;
    level_count = l_count;
    rot_step = 360 / sector_count; //рассчитываем расстояние между секторами
    trig_base = (R * Mathf.PI * 2) / sector_count; //величина ширины треугоника
    ↪  по основанию
    generator();

    //привязка сгенерированной ионосферы к земле
    GameObject.Find("Ionosfera").transform.localScale = new
    ↪  Vector3(0.05f,0.05f,0.05f);

}

void generator()
{
    //генерируем все кусочки среза ионосферы
    for (int j = 1; j <= level_count; j++)
    {
        for (int i = 0; i <= sector_count; i++)
        {

            GameObject new_obj = Instantiate(prefab, new Vector3(0,0,0),
            ↪  Quaternion.identity, transform); //создаем пустой объект (будущая
            ↪  часть сектора)
            //new_obj.transform.SetParent(transform); //назначаем ему родительский
            ↪  объект Inosfera
            Quaternion rotPos =
            ↪  GameObject.Find("MultiTarget").transform.localRotation;
            new_obj.transform.Rotate(rotPos.x, rotPos.y, rot_step * i+ rotPos.z);
            ↪  //поворачиваем объект для того, чтобы треугольники образовывали
            ↪  окружность

            //на текущий момент все объекты лежат в одной плоскости и перекрывают
            ↪  друг друга
            //чтобы пользователю были видны все объекты немного сместим их друг
            ↪  под друга

            float z = new_obj.transform.position.z + 0.001f * j;

            //отслеживаем объект в дополненной реальности
            Vector3 locPos =
            ↪  GameObject.Find("MultiTarget").transform.localPosition;
            new_obj.transform.position = locPos+ new Vector3(0, 0, z);


            new_obj.name = (level_count-j).ToString(); //даем имя объекту по его
            ↪  номеру сегмента
            new_obj.GetComponent<Get_Color>().sector = i; //создаем переменную
            ↪  level для подсчета удаленности объекта от земли
            new_obj.GetComponent<Get_Color>().level = j - 1; //создаем переменную
            ↪  level для подсчета удаленности объекта от земли


            new_obj.GetComponent<Mesh_Gen>().Gen(R * j, trig_base * j);//задаем
            ↪  объекту форму треугольника

            //временно дает рандомный цвет каждому кусочку, чтобы была видна
            ↪  сегментация
```

```
                new_obj.GetComponent<Renderer>().material.color =
                ↪  UnityEngine.Random.ColorHSV();
            }
        }
    }
}
```

Скрипт Data() с хранящимися данными по ионосфере не изменялся. Скрипт не прикрепляется к объектам и служит для описания класса данных по ионосфере.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[Serializable]
public class Data
{
    //публичные переменные для данных с сервера
    public float[] latitude;
    public float[] longitude;
    public float[] heigth;
    public float[] plasma;
    public int len;

    public float max_value;

    public void start_connect()
    {
        len = plasma.Length; //присвоим длину в отдельную переменную

        //вычисляем максимальное значение в плазме
        max_value = plasma[0];

        for (int i = 0; i < len; i++)
        {
            if (plasma[i] > max_value) max_value = plasma[i];
        }
    }
}
```

Mesh_Gen() генерирующий сектора окружности ионосферы также, остался без изменений. Добавляется к prefab sector_level.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Mesh_Gen : MonoBehaviour
{

    public void Gen(float R = 1, float height = 2)
    {

        MeshRenderer meshRenderer = gameObject.AddComponent<MeshRenderer>();
        meshRenderer.sharedMaterial = new Material(Shader.Find("Standard"));

        MeshFilter meshFilter = gameObject.AddComponent<MeshFilter>();
```

```
        Mesh mesh = new Mesh();

        Vector3[] vertices = new Vector3[4]
        {
            new Vector3(0, 0, 0),
            new Vector3(0, 0, 0),
            new Vector3(R, height/2,  0),
            new Vector3(R, -height/2,  0)
        };
        mesh.vertices = vertices;

        int[] tris = new int[6]
        {
            // lower left triangle
            0, 2, 1,
            // upper right triangle
            2, 3, 1
        };
        mesh.triangles = tris;

        Vector3[] normals = new Vector3[4]
        {
            -Vector3.forward,
            -Vector3.forward,
            -Vector3.forward,
            -Vector3.forward
        };
        mesh.normals = normals;

        Vector2[] uv = new Vector2[4]
        {
            new Vector2(0, 0),
            new Vector2(1, 0),
            new Vector2(0, 1),
            new Vector2(1, 1)
        };
        mesh.uv = uv;

        meshFilter.mesh = mesh;
    }

    public void Start()
    {
        if (transform.name == "sector_level(Clone)")
        {
            Gen();
        }
    }
}
```

Get_Color(), который окрашивает сегменты ионосферы тоже не изменяем и по прежнему прикрепляем к prefab sector_level.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Get_Color: MonoBehaviour
{
```

```
private GameObject center_figure; //родительский объект для всех кусочков
↪  ионосферы
public Data info_plasma; //значения ионосферы с сервера/файла

int counter = 0; //счетчик кадров
static float max_value;//максимальное значение plasma

public int sector_count;//общее значение секторов в ионосфере
public int level_count;//общее значение уровней в секторе

public int sector; //текущий сектор
public int level; //текущий уровень в секторе

//начальные значения
public float latitude;
public float longitude;
private Vector3 position;

//текщие значения
public float new_latitude;
public float new_longitude;



void Start()
{
    center_figure = transform.parent.gameObject; //получаем объект на котором
    ↪  задаются параметры ионосферы
    info_plasma = center_figure.GetComponent<Data_Plasma>().info_plasma;
    ↪  //записываем данные по плазме в переменную
    level_count = center_figure.GetComponent<Data_Plasma>().level_count; //число
    ↪  делений в секторе
    sector_count = center_figure.GetComponent<Data_Plasma>().sector_count; //число
    ↪  секторов в окружности


    //1000 - это максимальная высота по условию, а 90 - минимальная
    level = ((1000-90) / level_count) * (level-1);//вычисляем шаг по высоте и
    ↪  находим текущее значение

    //вычисляем стартовую позицию
    latitude = 0;
    longitude = (360 / sector_count) * sector;
    position = getXYZ(latitude, longitude);

    max_value = center_figure.GetComponent<Data_Plasma>().info_plasma.max_value;
}



float GetValue()
{
    int count = 0; //для количества точек попавших в сегмент
    float value = 0; //для суммы значений plasma в сегменте

    for (int i = 0; i < info_plasma.len; i++)
    {
        //находим точки попавшие в сегмент
        //11.25 - это интервал с которым приходят данные с сервера
```

```
        if (((info_plasma.longitude[i]) >= (new_longitude ) - 11.25f) &&
            ((info_plasma.longitude[i] ) <= (new_longitude) + 11.25f) &&
            ((info_plasma.latitude[i] ) >= (new_latitude) - 11.25f) &&
            ((info_plasma.latitude[i] ) <= (new_latitude) + 11.25f) &&
            ((info_plasma.heigth[i] - 90) >= level - ((1000 - 90) / level_count))
            ↪  &&
            ((info_plasma.heigth[i] - 90) <= level + ((1000 - 90) / level_count)))
        {
            value += info_plasma.plasma[i];
            count += 1;
        }
    }
    return value / count;
}

void Update()
{

    if ((counter + sector) % 100 == 0) //обновление сектора целиком
    {
        float value = 0; //среднее значение плазмы в сегменте

        //для определения смещения ионосферы используем Matrix4x4
        Vector3 angle = center_figure.transform.eulerAngles;
        Quaternion rotation = Quaternion.Euler(angle.x, angle.y, angle.z);
        Matrix4x4 m = Matrix4x4.Rotate(rotation);
        Vector3 new_position = m.MultiplyPoint3x4(position);

        //вычисляем текущие latitude и longitude
        float[] latlong = getLatLng(new_position);
        new_latitude = latlong[0];
        new_longitude = latlong[1];

        //определяем среднее значение плазмы
        value = GetValue();

        //определяем цвет для среднего значения plasma
        Color color = getColor(value);
        //задем объекту полученный цвет
        transform.GetComponent<Renderer>().material.color = new Color(color.b,
        ↪  color.g, color.r);
    }

    counter += 1;//счетчик кадров
}

//для определения координат в xyz по latlong
Vector3 getXYZ(float lat, float lon, float radius = 100)
{
    lat = lat * Mathf.PI / 180;
    lon = lon * Mathf.PI / 180;

    float x = radius * Mathf.Cos(lat) * Mathf.Cos(lon);
    float z = radius * Mathf.Sin(lat);
    float y = radius * Mathf.Cos(lat) * Mathf.Sin(lon);

    return new Vector3(x, y, z);
}

//для определения latlong по xyz
```

```
    float[] getLatLng(Vector3 pointXYZ)
    {
        float r = Mathf.Sqrt(pointXYZ.x * pointXYZ.x + pointXYZ.y * pointXYZ.y +
        ↪  pointXYZ.z * pointXYZ.z);
        float lat = Mathf.Asin(pointXYZ.z / r) * 180 / Mathf.PI;
        float lon = Mathf.Atan2(pointXYZ.y, pointXYZ.x) * 180 / Mathf.PI;
        float[] latlong = { lat, lon, r };
        return latlong;
    }

    //определение цвета для значениея plasma
    Color getColor(float value)
    {
        float step = 180 / (float)(max_value * 0.4);
        float color = (step * value)/360;
        return Color.HSVToRGB(color, 1, 0.7f);
    }

}
```

Скрипт Rotation(), который необходим для закрепления ионосферы перед взглядом пользователя, не меняем. Прикрепляется данный скрипт на родительский объект ионосферы (Ionosfera).

```
using UnityEngine;

public class Rotation : MonoBehaviour
{
    public Transform target;//прикрепляем камеру для слежения за углом поворота

    void Update()
    {
        // Rotate the camera every frame so it keeps looking at the target
        transform.LookAt(target);
        float xangle = transform.rotation.x;
        float yangle = transform.rotation.y;
        float zangle = transform.rotation.z;

        transform.Rotate(xangle+90, yangle, zangle);

    }
}
```

Скрипт подключения к серверу Connect_Server(), необходимо дополнить передачей статусов в нижнюю панель интерфейса и передачей ошибки, в случае ее возникновения. Добавляем данный компонент на объект EventSystem.

```
using System; //добавляем для работы со врememенем
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking; //подключаем класс для работы с сервером

public class Connect_Server : MonoBehaviour
{

    //для реализации запросов к серверу используем Coroutine и одну функцию на все
    ↪  запросы
```

```
    //передаем Url запроса и название переменной, в которую будет записываться ответ
    ↪  сервера
    public IEnumerator getRequest(string url, string type)
    {
        //создаем поток связи с сервером
        using (UnityWebRequest req = UnityWebRequest.Get(url))
        {
            //чтобы продолжить, дожидаемся ответа сервера
            yield return req.SendWebRequest();

            //проверяем, не возникло ли ошибок
            if (req.isDone && !req.isNetworkError && !req.isHttpError)
            {
                Debug.Log("Comment: Ответ от сервера получен - type = " + type);
                //если ответ от сервера получен записываем в реестр соответвующую
                ↪  переменную
                PlayerPrefs.SetString(type, req.downloadHandler.text);
                PlayerPrefs.Save();

                ↪  GameObject.Find("EventSystem").GetComponent<Get_Settings>().status.text
                ↪  = "Данные " + type + " получены";
            }
            else
            {
                //действия при ошибке
                Debug.Log("error_connect_server");

                ↪  GameObject.Find("EventSystem").GetComponent<Get_Settings>().status.text
                ↪  = "Ошибки подключения к серверу. Запуск тестовых данных";
            }

             ↪  GameObject.Find("EventSystem").GetComponent<Get_Settings>().Update_Status();
        }
    }

    //теперь создаем функции на вызов корутины

    //значение плазмы можно получать во времени, поэтому должна быть модификация
    ↪  запроса и на время
    //в функцию будем передавать переменную time, в которой записана информация о
    ↪  времени в следующем формате ?hour=16&day=16
    public void get_plasma(string time = null)
    {
        string type = "plasma_all";
        string PlasmaUrl = "http://178.250.159.250:8000/plasma";

        //если информацию о времени передали, то дозаписываем запрос
        if (time != null)
        {
            PlasmaUrl += time;
        }

        Debug.Log("Comment: Получение значений плазмы по всей Земле PlasmaUrl = " +
        ↪  PlasmaUrl);
        StartCoroutine(getRequest(PlasmaUrl, type));//вызываем корутину по
        ↪  получившемуся запросу
}

    public void get_sputniks()
    {
```

```
        string type = "sputniks";
        string SatsUrl = "http://178.250.159.250:8000/sats";
        Debug.Log("Comment: Получение информации о спутниках");
        StartCoroutine(getRequest(SatsUrl, type));
    }


    //функция получения данных по сегменту будет принимать данные сегмента
    public void get_segment(int height_v, int n_segment_v, int lat_v, int long_v)
    {
        string type = "plasma_segments";
        string SegmentUrl = "http://178.250.159.250:8000/segment?height=" + height_v +
        ↪  "&n_segment=" + n_segment_v + "&lat=" + lat_v + "&long=" + long_v;
        Debug.Log("Comment: Получение данных плазмы по сегменту");
        StartCoroutine(getRequest(SegmentUrl, type));
    }

    public void Start()
    {
        Debug.Log("Comment: Запуск приложения. Сразу получаем данные по спутникам и
        ↪  плазме");
        get_plasma();
        get_sputniks();
        GameObject.Find("EventSystem").GetComponent<Get_Settings>().status.text =
        ↪  "Получение информации с сервера";
    }
}
```

Дописываем теперь скрипт Get_Settings так, чтобы все данные обновлялись по настройкам и ответам сервера и выводился нужный статус в строке состояния. Добавляем данный компонент на объект EventSystem.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.Linq;
using System;

public class Get_Settings : MonoBehaviour
{
    //создаем переменные для всех параметров
    public int sector_count = 32; //количество секторов в ионосфере
    public int level_count = 10; //количество уровней в секторе


    //переменные для запроса данных с сервера за определенное время
    public int day = 0; //день года от 1 до 365
    public int hour = 0; //час от 1 до 24

    //переменные спутников
    //public bool sputniks = true; //отображение моделей спутников
    //public bool s_traectory = true; //отображение траектории спутников

    public string style;//для типа стилизации


    //переменные для привязки объектов
    public InputField day_p;
```

```
public InputField hour_p;
public Slider count_sector;
public Text text_count_sector;
public Slider count_level;
public Text text_count_level;
public Toggle sputniks_p;
public Toggle s_traectory_p;
public ToggleGroup toggleGroup_style_p;

//строка состояния внизу окна
public Text status;

//при старте задаем базовые  параметры
void Start()
{
    //задаем параметры размерности массивам моделей
    Default_model = new GameObject[32];
    Fantasy_model = new GameObject[32];

    //ищем модели спутников в папках
    var go1 = Resources.LoadAll("Model1/", typeof(GameObject));
    var go2 = Resources.LoadAll("Model2/", typeof(GameObject));

    //записываем в массив найденные объекты
    for (int i = 0; i < 32; i++)
    {
        Default_model[i] = (GameObject)go1[i];
        Fantasy_model[i] = (GameObject)go2[i];
    }

    style = "0";//всегда при старте будет обычная стилизация
}

//переменная для обновления данных ионосферы
private bool Up_plasma_data = false;
//создаем функцию считывания введенных пользователем данных
public void get_value()
{
    //проверка дня
    if (day_p.text!="" && Int32.Parse(day_p.text) != day &&
    ↪  Int32.Parse(day_p.text) <= 365 && Int32.Parse(day_p.text) > 0)
    {
        day = Int32.Parse(day_p.text);
        Up_plasma_data = true;
    }
    else day = 0;
    //прверка часа
    if (hour_p.text != "" && Int32.Parse(hour_p.text) != hour &&
    ↪  Int32.Parse(hour_p.text) <= 24 && Int32.Parse(hour_p.text) > 0)
    {
        hour = Int32.Parse(hour_p.text);
    }
    else hour = 0;


    ↪  Update_Style(toggleGroup_style_p.ActiveToggles().FirstOrDefault().name);//проверка
    ↪  обновления стилизации
    Update_Data_Plasma();//проверка обновления ионосферы
    Update_Plasma();//проверка обновления ионосферы
}
```

```
//переменная для обновления генерации ионосферы
private bool Up_plasma = false;
//функция изменения генерации секторов
public void Update_Plasma()
{
    if (Up_plasma)
    {
        GameObject obj = GameObject.Find("Ionosfera");
        var o = obj.GetComponentsInChildren<Transform>();
        for (int i=1; i < o.Length;i++) Destroy(o[i].transform.gameObject);
        obj.GetComponent<Data_Plasma>().Start_Plasma(sector_count, level_count);
        Up_plasma = false;

        status.text = "Изменения в сегментации ионосферы";

    }
}

public void Update_Data_Plasma()
{
    if (Up_plasma_data)
    {
        Debug.Log("делаем новый запрос к серверу = ? hour = 16 & day = 16");
        string time = null;
        if (hour!=0 && day != 0)
        {
            time = "?hour="+hour+"&day="+day;
        }
        start_count = 1;
        transform.GetComponent<Connect_Server>().get_plasma(time);
        Up_plasma_data = false;

        status.text = "Идет запрос иносферы по hour=" + hour + "&day=" + day ;

    }

}



//объекты для изменения области видимости
public GameObject parent_sputniks;
public GameObject[] traectory;

//функция для изменения видимости спутников
public void Change_sputniks()
{

    if (sputniks_p.isOn == true && s_traectory_p.isOn == true)
    {
        parent_sputniks.SetActive(true); //отображение спутников
        foreach (var g_obj in traectory)
        ↪  g_obj.SetActive(s_traectory_p.isOn);//отображение траектории
    }
    else if (s_traectory_p.isOn == false && sputniks_p.isOn == true )
    {
        parent_sputniks.SetActive(true); //отображение спутников
        foreach (var g_obj in traectory) g_obj.SetActive(false);//отображение
        ↪  траектории
```

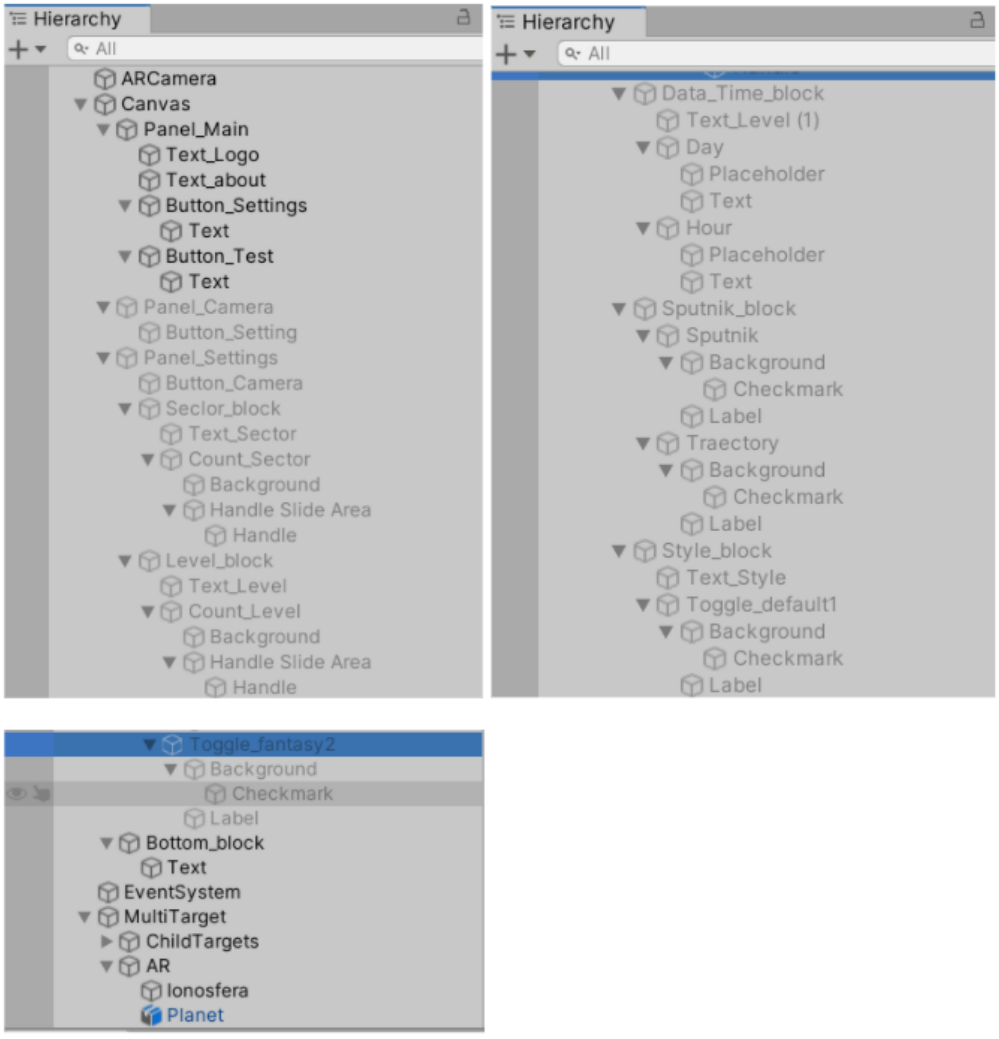
```
    }
    else if (sputniks_p.isOn == false)
    {
        parent_sputniks.SetActive(false); //отображение спутников
        //foreach (var g_obj in traectory) g_obj.SetActive(false);//отображение
        ↪  траектории
    }
    status.text = "Настройки отображения спутников изменены";

}




//функция обновления значений ползунков и текста к ним
public void Update_Text()
{
    if (count_sector.value != sector_count)
    {
        text_count_sector.text = "Количество секторов: "+ sector_count.ToString();
        sector_count = (int)count_sector.value;
    }

    if (count_level.value != level_count)
    {
        text_count_level.text = "Количество уровней: " + level_count.ToString();
        level_count = (int)count_level.value;
    }
    status.text = "Настройки сектора изменены";
    Up_plasma = true; //сообщаем, что нужно обновление ионосферы
}




//для стилизации находим все объекты на разные стили
public GameObject[] Default_model;
public GameObject[] Fantasy_model;
public Material Default_planet;
public Material Fantasy_planet;

//переменная с объектом Планеты
public GameObject Planet;

//при обновлении стилизации меняем объекты
public void Update_Style(String style_up)
{
    //проверка на изменение слилей
    if (!(style.Contains('1') && style_up.Contains('1')) && !(style.Contains('2')
    ↪  && style_up.Contains('2')))
    {

        if (style != "0") foreach (var go in
        ↪  GameObject.FindGameObjectsWithTag("sput")) Destroy(go);
        if (style != "0") foreach (var go in
        ↪  GameObject.FindGameObjectsWithTag("sputniks_traectory")) Destroy(go);

        style = style_up;
```

```
        if (style.IndexOf('1') != -1)
        {
            transform.GetComponent<Stat_Init>().Start_Sput(Default_model);
            Planet.transform.GetComponent<MeshRenderer>().material =
            ↪  Default_planet;
        }
        else
        {
            transform.GetComponent<Stat_Init>().Start_Sput(Fantasy_model);
            Planet.transform.GetComponent<MeshRenderer>().material =
            ↪  Fantasy_planet;
        }

        parent_sputniks = transform.GetComponent<Stat_Init>().parent;
        traectory = new GameObject[32];
        traectory = GameObject.FindGameObjectsWithTag("sputniks_traectory");

        status.text = "Стилизация изменена";
    }
}



public int start_count=0;

//обновление статусов при старте
public void Update_Status()
{
    if (start_count!=2)
    {
        if (status.text == "Данные sputniks получены")
        {
            transform.GetComponent<Stat_Init>().Init_Sput(false);
            Update_Style("1");

            status.text = "Данные по спутникам получены и визуализированы";
            start_count++;
        }
        if (status.text == "Данные plasma_all получены")
        {
            GameObject.Find("Ionosfera
            ↪  ").GetComponent<Data_Plasma>().data_processing();
            GameObject.Find("Ionosfera
            ↪  ").GetComponent<Data_Plasma>().Start_Plasma(sector_count,
            ↪  level_count);

            status.text = "Данные по ионосфере получены";
             start_count++;
        }
        if (status.text == "Ошибки подключения к серверу.Запуск тестовых данных")
        {
            Debug.Log("sss");
            transform.GetComponent<Stat_Init>().Init_Sput(true);
            Update_Style("1");

            GameObject.Find("Ionosfera
            ↪  ").GetComponent<Data_Plasma>().data_processing_test();
            GameObject.Find("Ionosfera
            ↪  ").GetComponent<Data_Plasma>().Start_Plasma(sector_count,
            ↪  level_count);
```

```
                status.text = "Ошибки подключения к серверу. Тестовые данные
                ↪  визуализированны";
                start_count = 2;
            }
        }
    }
}
```

После присоединения всех скриптов к соответствующим объектам производим корректировку масштабов Земли/Ионосферы/Спутников относительно друг друга и маркера. Далее компилируем проект в исполняемый файл и тестируем на целевом устройстве.

Итоговая иерархия проекта:



Итоговое демо-видео: https://youtu.be/xnWlPVmi81o

Тестирование корректности отображаемых данных

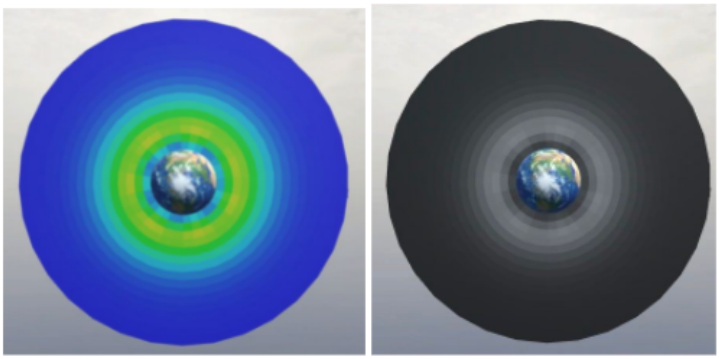
Тестирование корректности отображаемых данных

Для проверки корректности отображаемых данных дается файл с ответом на запрос вида 178.250.159.250:8000/plasma. Данные из этого файла требуется отобразить на созданной модели ионосферы.

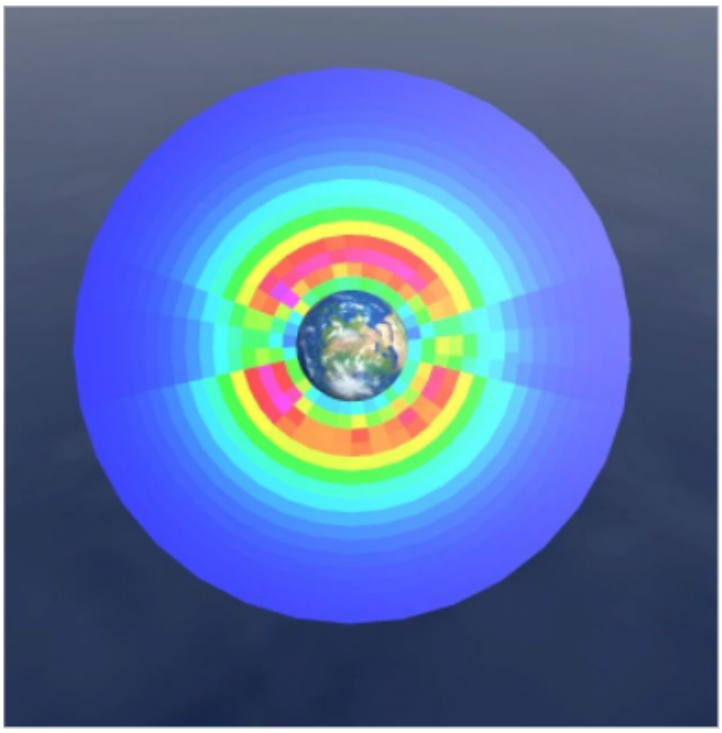
Для этого нам потребуется добавить файл в проект и вместо получения данных с сервера сделать чтение данных из файла, как это делалось при отладке.

```
void data_processing()
    {
        //указываем ссылку на файл с данными
        string path = "Assets/Resources/json/data.json";

        //читаем данные из файла в текстовую переменную
        StreamReader reader = new StreamReader(path);
        string data = reader.ReadToEnd();
        reader.Close();

        //сереализуем данные в класс Data
        info_plasma = JsonUtility.FromJson<Data>(data);
    }
```

После загрузки данных в созданные участниками финала олимпиады приложения, визуализация должна быть идентична приведенным ниже вариантам. При просмотре состояния ионосферы в районе северного полюса:



для нулевого меридиана изображение следующее:



Баллы за решения

Задачи 3Д-дизайнера

Создание 3D-модели земли (2 балла)

Разработка 3D-модели ионосферы (6 баллов)

Конструирование 3D-моделей спутников (6 баллов)

Разработка стилизованных 3D-моделей (6 баллов)

AR-разработчик (архитектор)

Создание вайрфрейма приложения (3 балла)

Отрисовка иконок для приложения (3 балла)

Отображение виртуальных объектов:

- Прикрепить виртуальные объекты к кубу (30 баллов)
- Прикрепить виртуальные объекты без маркера (пользователь может выбрать куда поставить виртуальных глобус) (10 баллов)
- Прикрепить виртуальные объекты к маркеру (4 баллов)

Программист

Получения данных с сервера об ионосфере:

- полные данные (3 балла)
- или срез данных по взгляду пользователя(3 балла)
- Получение данных о спутниках(2 балла)
- Слежение за взглядом пользователя (3 балла)
- Вычисление координат данных, которые требуется отобразить (15 баллов)

Генерация поля ионосферы с разной дискретизацией:

- несколько 3D-моделей (8 баллов)
- генеративный метод или его вариация дублирование одного объекта (12 баллов)

Визуализация данных на сфере:

- грейскейл схема (2 балла) — визуализация в оттенках серого
- тепловая схема (5 баллов)

Отрисовка полета спутников:

- если движение правильное (2 балла)
- если движение плавное, полет по дуге (5 баллов)
- Можно посмотреть историю полета каждого спутника (отрисовано дугой) (5 баллов)

Бонусные баллы к итоговой сборке.

Первый бонус (10 баллов):

- Наличие исполняемого файла, под целевое устройство
- Соответствие итогового приложения варфрейму от дизайнера
- Присутствуют комментарии в коде

Второй бонус (не засчитывается, если не выполнены пункты прошлого бонуса):

- В приложении реализованы все опции и работают корректно (4 балла)
- Адаптивная верстка приложения (2 балла)
- Отображение траектории полета спутников (2 балла)
- Разная стилизация виртуального мира (2 балла)

Третий бонус (не засчитывается, если не выполнены пункты прошлого бонуса):

- Присутствуют элементы интерфейса для изменения времени. Данные модели меняются в соответствии с ответом сервера (10 баллов)