## Томилов Ярослав. Отчет о выполнении индивидуального задания.

Я хотел создать игру в которой у игрока будет возможность построить свой собственный корабль из готовых модулей и кастомизировать интерфейс полученной системы.

Также я хотел добавить возможность разрушать 3D элементы окружения без привязки окружения к 3D сетке или же блокам, но к сожалению я не смог достичь этой цели. Сущность проблемы и рассмотренные мною варианты решения а также альтернативные методы реализации были рассмотрены мною в соответствующей главе.

## Цели:

* Создать инструменты для ручной постройки космического корабля, а именно свободную камеру, добавление, удаление и поворот деталей.
* Игрок должен иметь возможность редактировать свой интерфейс и назначать функционал кнопок и слайдеров так как ему захочется.
* Космический корабль должен иметь возможность перемещаться притягивать к себе предметы.
* Во время миссии игрок слеп и не может видеть окружающие корабль объекты. Чтобы ориентироваться в пространстве ему нужно облепить корабль камерами, которые проецируют картинку на интерфейс.
* Назначить игроку простую простое к выполнению задание, для демонстрации функционала корабля.
* Также проанализировать данные случайной генерации, но это будет в практическом задании, не здесь.

## 1.1 Создание булевых операций

### 1.1.1 Модификация 3D моделей. Binary Space Partitioning в 3D

Binary Space Partitioning - это метод, на каждом шаге которого некое пространство делится ровно на две половины, наиболее известное его применение это поиск значения внутри одномерного списка, но алгоритм не ограничен любым количеством измерений.

Деление пространства на две половинки будет происходить относительно первого полигона в списке, я не буду применять какие-либо способы поиска наиболее оптимальной плоскости для деления.

Каждый полигон имеет объект плоскости, для разделения полигонов создана функция SplitPolygon(), принадлежащая классу плоскости. Функция принимает 5 значений: полигон, который нужно разделить этой плоскостью, и 4 списка, coplanarFront, coplanarBack, Front и Back. Полигоны отправятся в массив Front если они находятся спереди полигона, в Back если находятся сзади. Если полигон лежит на плоскости то он отправляется в coplanarFront или coplanarBack в зависимости от того совпадают ли у них направления.

Довольно часто полигоны пересекают плоскость, и поэтому, прежде чем отправиться в какую-то из групп их необходимо разделить. Для этого у плоскости рассчитывается нормаль, перпендикулярный вектор, направленный вперед относительно плоскости. Потом берется скалярное произведение между каждой вершиной полигона и нормалью. Если результат положительный, то точка находится перед плоскостью, если отрицательный, то сзади. Ноль означает что точка лежит на плоскости.

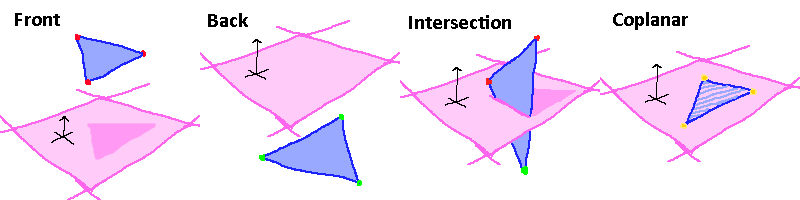


Рисунок 2: Возможные конфигурации

Теперь можно посмотреть на некоторые возможные конфигурации, которые могут получиться в результате такой проверки.

* Если для каждой вершины скаляр равен нулю, то полигон лежит на плоскости. Нужно лишь найти скаляр между нормалью делимого полигона и этой плоскостью, если он положителен, то полигон отправится в CoplanarFront, если отрицателен, то в CoplanarBack.
* Если есть хотя-бы один Front, но нет ни одного Back, то полигон находится спереди, и отправляется в соответствующий список.
* Если есть хотя-бы один Back, но нет ни одного Front, то полигон находится за плоскостью, и отправляется в соответствующий список.
* Если же есть хотя-бы один Front и хотя-бы один Back, то полигон пересекает плоскость, а значит должен быть разделен.

Используя эту функцию, каждый 3D объект можно будет превратить в BSP дерево, каждая нода которого является плоскостью, которая в свою очередь делит всю остальную модель на две части: ту которая находится спереди полигона, и ту которая за полигоном.

Полигоны, которые оказываются пересечены плоскостью должны быть поделены на два новых полигона, каждый из которых отправится на противоположные стороны дерева.



Рисунок 3: Функция Build() и DeleteInsides()

В итоге у меня получилась рекурсивная функция, которая каждым делением делит пространство на две части, и для каждой из этих частей вызывает саму себя.

Аналогично создается функция DeleteInsides(), эта функция также получает список полигонов, но в отличии от функции Build она избавляется от всех полигонов, которые находятся внутри первой 3D модели. Схему того как может быть исполнена эта команда в 2D можно увидеть на схеме ниже.

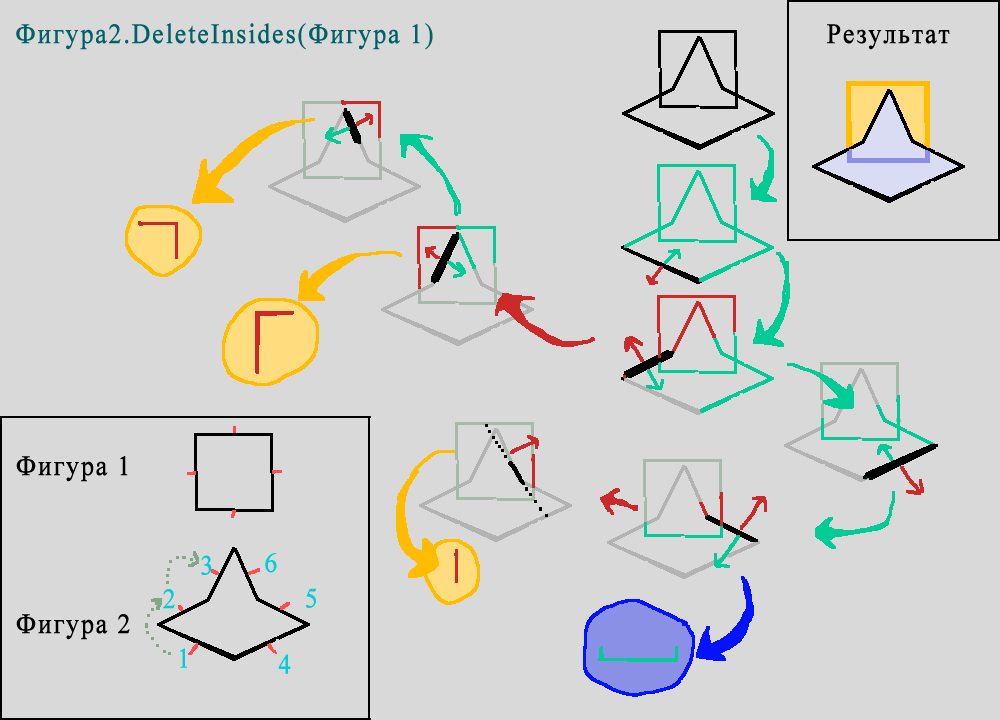


Рисунок 4. Пример исполнения команды DeleteInsides()

Вызвав команду из модели 1 над моделью 2 код вернет список полигонов принадлежащих второй модели, которые не находятся внутри модели 1. На рисунке 4 изображена аналогия исполнения этого кода, но в 2D пространстве. Вместо того чтобы делить пространство плоскостью полигона, пространство разделяется на половинки гранями фигур. Когда перед гранью есть грани Фигуры 2, но у этой грани нет граней спереди, тогда грани фигуры 2 возвращаются. Аналогично, когда сзади грани есть грани Фигуры 2, но сзади нет граней фигуры 1, которые могут быть нодой BSP дерева, в этом случае грани Фигуры 2 уничтожаются.

### 1.1.2 Булевы операции над объемами

После создания функции удаляющей внутренние полигоны можно с легкостью создать самые базовые операции над объемами.

**Union** - самая простая операция, она подразумевает объединение двух моделей. Для ее реализации нужно удалить полигоны модели 1 что внутри модели 2, и удалить полигоны модели 2 внутри модели 1.

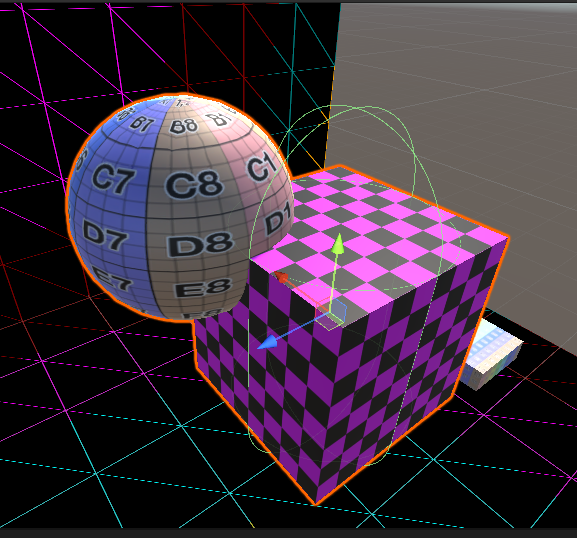


Рисунок 5: Команда Union и пример результата ее исполнения.

Substract - чуть более сложная булева операция. Для ее реализации нужно инвертировать Модель 1, тоесть перевернуть все ее полигоны. Инвертированная модель будет определять полигоны внутри нее как находящиеся снаружи, а те что находятся снаружи как те что внутри. Из модели 2 удаляются полигоны находящиеся внутри инвертированной модели 1, а из модели 1 удаляются полигоны внутри модели 2.

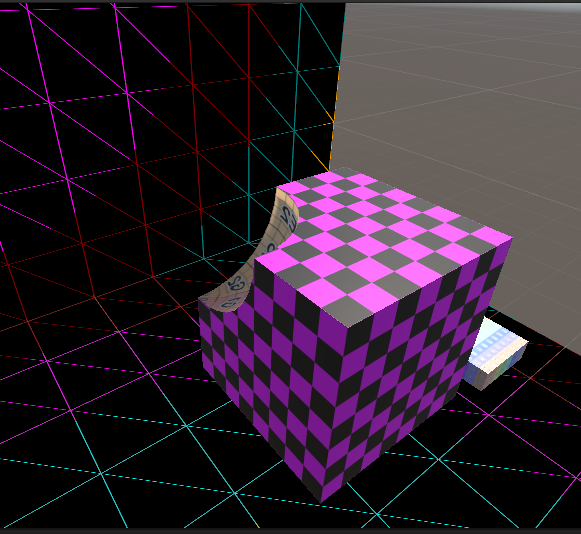


Рисунок 6: Команда Substract и пример результата ее исполнения.

Intersect - команда находящее пересечение двух объемов. Эта команда похожа по структуре на Substract, но в отличии от нее Intersect инвертирует оба объема.

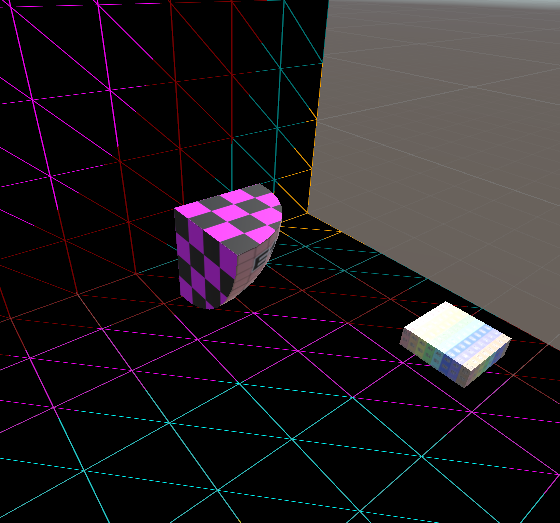


Рисунок 7: Команда Intersect и пример результата ее исполнения.

### 1.1.3 Причины отказа от системы CSG

Изначально я планировал применить Constructive Solid Geometry для выполнения базовых операций над различными объемами, но у меня было несколько причин отказаться от этой идеи.

## 

Рисунок 7: Распространенный и экстремальный аретефакты.

Данная система прекрасно справляется с обработкой выпуклых фигур, но она сталкивается с проблемами когда дело доходит до обработки объемов обладающих впадинами, также когда фигуры обладают большим количеством полигонов в результате обработки получается объект состоящий из еще большего количества полигонов, из-за того что каждое деление делит вообще все полигоны, даже те которые пересекаются с оригинальным полигоном.

Проблему с количеством полигонов решить не сложно, так как в коде все копланарные полигоны отправляются в одну BSP ноду. Нужно найти все границы полигонов и удалить из них границы разделяемые двумя полигонами, после чего триангулировать полученные полигоны, но из-за проблемы номер один эта операция не имеет смысла.

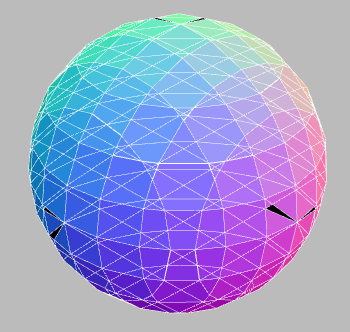
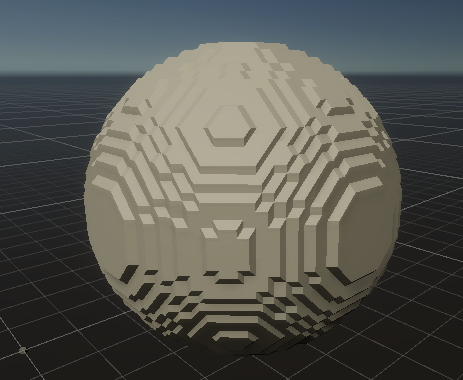
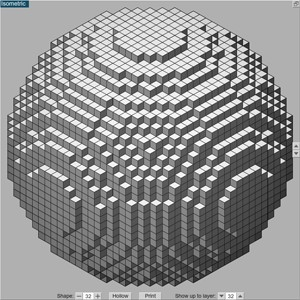
Первая проблема является более комплексной чем вторая и не имеет единственного решения. Причины проблемы понятны, она вызвана Binary Space Partitioning которое производят излишнее деление 3D модели и тем как определяются полигоны снаружи и внутри, а также точностью Float.

Сейчас я вижу три возможных варианта решения проблемы. Первый вариант это поиск артефактов-полигонов используя сложную систему данных. Каждая грань корректной 3D модели принадлежит двум полигонам, поэтому если определить полигоны имеющие такие грани и рекурсивно их удалять то можно избавиться от артефактов. Этот способ очень медленный, поэтому у меня остается лишь один вариант, переписать сам алгоритм.

Второй вариант это переписать алгоритм так чтобы он не использовал BSP, а применял другой алгоритм, например как в видео по ссылке ниже. Алгоритм довольно сложный и вероятнее всего медленный на этапе поиска точек принадлежащих обеим фигурам, но он не создает полигоны за пределами основной модели, и соответственно не оставляет артефакты.

[Geometry in Milliseconds: Real-Time Constructive Solid Geometry](https://youtu.be/Iqmg4gblreo)

Третий вариант это сделать вид что разрушается 3D модель, а на самом деле отображать разрушаемые объекты как 3D матрицу, как в майнкрафт. Используя алгоритм Marching Cubes чтобы замаскировать блочную структуру.



Я решил отложить работу с разрушаемостью до лучших времен, и вместо этого сфокусироваться на других частях проекта.

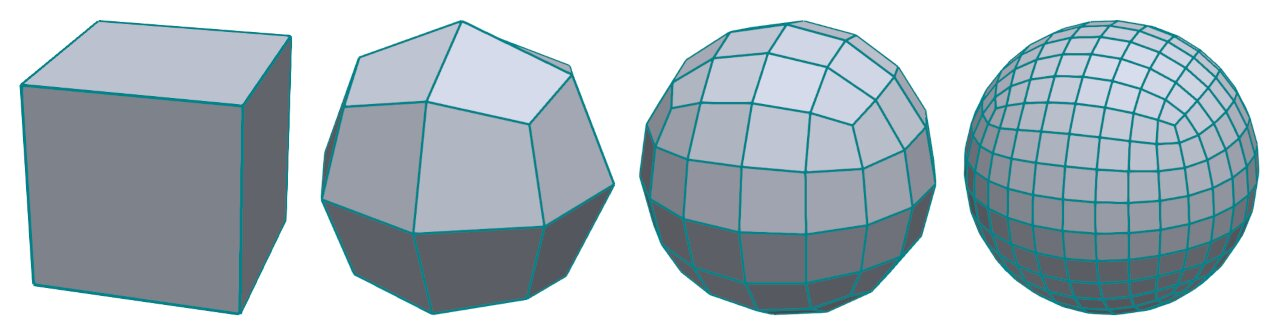
## 1.2 Генерация простых фигур, а также Астероидов

### 1.2.1 Платонические тела. Куб. Икосаэдр. Сфера.

Рисунок 8. Платонические тела.

Правильные многогранные 3D объекты, или же платонические тела - это примитивы, комбинируя или модифицируя объемы можно получить множество самых различных фигур. Для создания астероидов мне потребуется две фигуры: Куб и икосаэдр, они будут использованы для процедурной генерации сферы.

Чтобы процедурно сгенерировать сферу необходимо взять некую фигуру, а после рекурсивно разделить каждую из сторон, а после спроецировать каждую вершину на поверхность сферы.



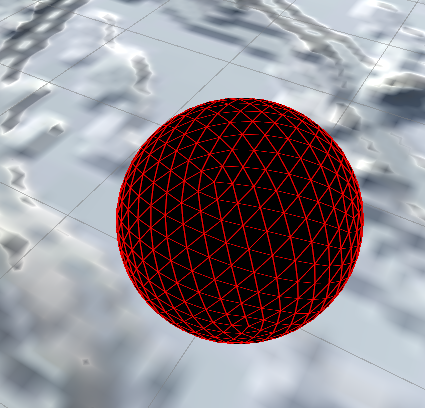
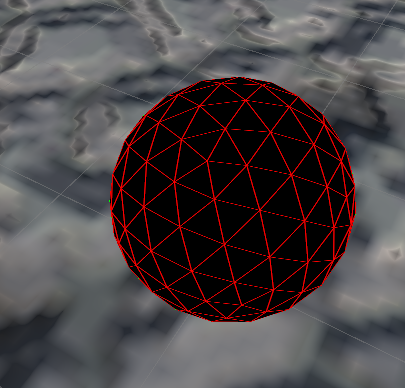
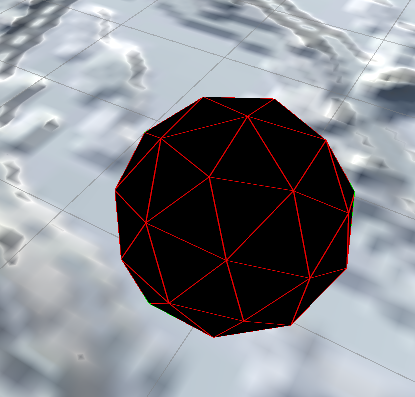
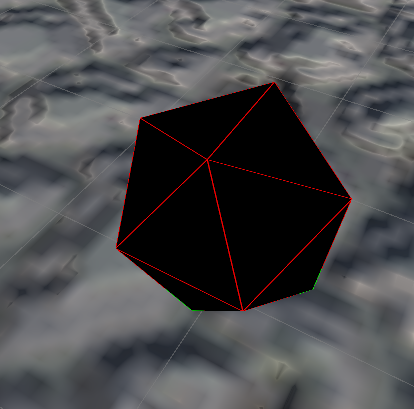


Рисунок 9. Трансформация куба и икосаэдра в сферу

Сфера, построенная на основе икосаэдра была выбрана мной потому что вершины 3D модели расположены равномерно по всей поверхности сферы. Этого нельзя сказать о сфере на основе куба. Очевидно, что точки вершин сферы расположены все плотнее друг к другу по мере приближения к точкам, которые ранее были углами куба.

Стоит заметить, что сфера, предоставляемая Unity, является сферой построенной на основе куба, поэтому она не является оптимальной для дальнейший манипуляций.

Также, для более тонкой обработки я создал команды Move() и Resize(), которые передвигают и изменяют размер модели соответственно.

### 1.2.2 Форма астероида. Диаграммы Вороного. Деформация сферы.

Сферическая форма не является самой естественной формой для однородного куска камня. Масса астероида не должна быть распределена равномерно относительно центра, должны быть выпуклости, впадины. При это поверхность астероида не должна иметь острых углов или резких переходов, она должна быть относительно гладкой.

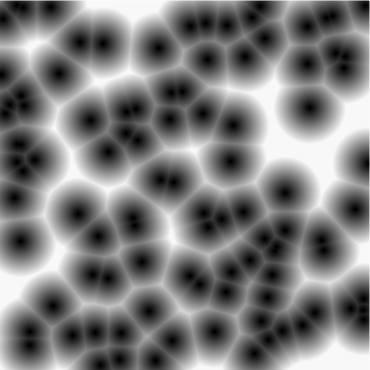
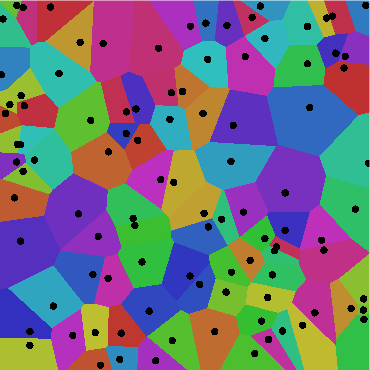


Рисунок 10: Диаграммы Вороного и диаграмма дистанций.

Диаграмма Вороного для группы точек S в двух измерениях предполагает разбиение плоскости, на области, каждая из которых образует множество точек, наиболее близких к одному из элементов множества S, чем к любому другому элементу множества.

В результате визуализации диаграммы Вороного получаются органичные фигуры, своим видом напоминающие клетки организма или же пузырики. Эти диаграммы довольно популярны в сфере компьютерной графики для создания текстур, которые можно описать как “органические” или “естественные”, например для создания текстуры лавы, булыжника, среза бревна. Черепаший панцирь, кожа рептилий или пятна на шкуре жирафа также представляют собой естественную диаграмму Вороного.





Рисунок 11. Код искажающий сферу, результат и схема

Для создания 3D модели астероида сначала будет создана процедурно-генерируемая сфера на основе икосаэдра. В пределах этой сферы я создаю произвольное количество точек, каждая из которых имеет случайные X, Y и Z координаты. Далее для каждой вершины сферы будет найдена ближайшая точка из списка случайных точек в пространстве, а затем будет найдено расстояние от вершины до этой точки.

В результате получаются натурально выглядящие объемы, форма которых очень схожа с формой речных камней. Прекрасная форма для астероида.

## 1.2 Редактор космического корабля

Игрок должен иметь возможность строить и изменять свой космический корабль, под этим подразумевается создание трехмерной сетки, в каждом узле которой может находиться компонент космического корабля, например стена. Компоненты должны иметь разный размер, начиная от куба со стороной в одну игровую единицу, и заканчивая компонентами чуть более сложных размеров, например параллелепипедов размерами в 3 единицы в ширину, 2 в длину, и 8 в высоту.

Также система должна быть достаточно гибкой и модульной. Эти два пункта необходимы для получения возможности быстро и просто добавлять новые компоненты обладающие схожими свойствами но разнящимися характеристиками, например химический двигатель и ионный двигатель, которые отличаются мощностью и размерами.

Игрок должен иметь удобный интерфейс для выбора деталей и для произведения простых операций над деталями, а именно добавление, удаление, поворот.

### 1.2.1 Создание трехмерного массива

Сначала будет создан объект космического корабля, к нему будет привязана матрица фиксированного размера. В центре матрицы будет создан блок ядра корабля, или же рубка, к которой могут крепиться компоненты. У каждого модуля будет множество параметров и функций, но сейчас важны только координаты и размер. Размеры рубки будут представлять собой куб 3 на 3 на 3 игровых единицы.

Добавление нового компонента может показаться довольно тривиальной задачей, но это предположение правильно только в ситуации когда элемент занимает лишь одну клетку.

### 1.2.2 Добавление элемента

При нажатии на кнопку мыши игра выпустит из камеры луч используя встроенную в физический движок Unity функцию Raycast(), которая возвращает первый коллайдер который пересек луч. Если такой коллайдер имеется, далее код получит из него точку пересечения с лучом и вызовет функцию GetAdjacentSpace().



Рисунок 1: Функция выпускающая луч сверху, GetAdjacentSpace снизу.

Функция GetAdjacentSpace() принимает на вход одну точку координат, потом находит такой узел матрицы, угол между которым и полученной точкой относительно центра кубоида меньше всех остальных.

Более простой способ реализации этой функции - это округление координаты полученной точки до ближайшего значения, но этот вариант очень сильно зависит от положения космического корабля в пространстве а также от того в какую сторону будет округлено значение, что не всегда удается предсказать. Во время тестирования я часто натыкался на эту проблему, и поэтому отправился искать другое решение.

Также, если моделька какого-либо элемента не является кубом, а например сферой, то в большом количестве ситуаций округление точки перенесет ее на уже занятую клетку.

Этот векторный метод очевидно более медленный чем округление, но у него нету перечисленных выше недостатков. На данный момент я использую самый наивный способ реализации, а значит алгоритм можно оптимизировать, как вариант, можно сначала можно воспользоваться округлением, затем найти соседей округленной клетки и выполнить векторную проверку.

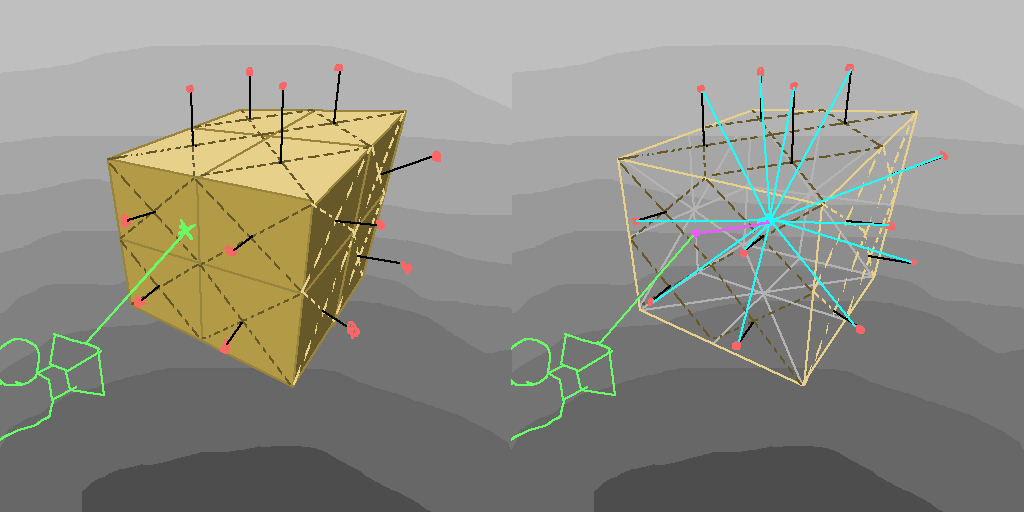


Рисунок 2: Схема наивного векторного метода

Теперь, когда у игрока есть возможность выпустить луч, определить точку его попадания и найти ближайший узел матрицы, можно добавить элемент в эту клетку. Сначала нужно определить свободно ли место в выбранной игроком позиции, напомню, строительные блоки в этой игре могут занимать более одной клетки.



Рисунок 3: Код функции добавления и поиска свободного места

Для определения наличия свободного места я использую функцию naiveVacantSpaceSearch(), эта функция принимает точку в которую нужно установить объект, она проверяет можно ли установить объект функцией CanPlaceBig(), которая проверяет каждый узел матрицы которой должен занять объект, как только она находит другой объект в одном из узлов функция возвращает False. Если функция naiveVacantSpaceSearch не смогла разместить объект, то она смещает его и пробует снова.

Как и в случае с алгоритмом поиска ноды на которую смотрит игрок, у этого алгоритма еще много возможностей для оптимизации, поэтому я и назвал его наивным.

Самый очевидный способ оптимизации алгоритма это предоставление ему личной матрицы, или модификация границ циклов For(){} каждый раз когда проверка натыкается на уже занятую клетку. Также можно использовать “отталкивающий алгоритм”, который перемещает клетку в противоположную сторону от занятой клетки.

Теперь, когда все эти проблемы решены, я, как игрок, могу добавлять элементы космического корабля.



Рисунок 4: Кораблик

### 1.2.3 Удаление элемента

Эта операция в разы, в разы проще чем добавление. Используя ранее названную функцию RayCast я вместо точки и объекта нахожу лишь объект и удаляю его, перед удалением не забыв обнулить занимаемые им места в матрице.

### 1.2.4 Перемещение элементов

Перемещение детали это тоже самое что добавление, только не нужно добавлять новый элемент. Для перемещения детали я реализовал несколько новых функций. Сначала деталь вызывает функцию DeleteMyFootprint() которая очищает все занятые деталью узлы.

Затем деталь пытается занять место находящееся на расстоянии одного узла в направлении которое выбрал игрок, всего направлений 6, каждое привязано к определенной оси координат.

### 1.2.5 Вращение деталей

Вращение деталей это довольно важная операция для строительной игры, что особенно важно в моем случае, так как разные компоненты будут иметь разные измерения и функционал в зависимости от ориентации. Самым главным примером такого элемента является простой двигатель, который в обычном состоянии имеет размеры 1х2х1. При вращении его измерения будут меняться на 2х1х1, 1х1х2 и обратно на 1х2х1, а его функционал будет изменяться в соответствии с направлением, направление в котором он прилагает силу будет меняться.

Unity имеет встроенный инструмент для определения вращений - quaternion, но в моем случае он не является оптимальным. Первая проблема это факт того что он может возвращать отрицательные значения, а размеры объекта не могут быть отрицательными. Вторая это то что он может возвращать значения типа float, что иногда случается из-за неидеальности алгоритмов и вполне нормально для стандартных игровых объектов, но разрушительно для моих блоков которые должны занимать целое количество узлов, блок не может иметь размеры 0.998х2.002х1.002 узла. Также их довольно сложно вращать не имея точки отсчета. Мне нужно сделать аналогичную систему, но работающую на целочисленных переменных.

Также стоит заметить, что направление двигателя может быть отрицательным, но не его размеры. А отображение модели двигателя зависит от предоставляемых Unity кватернионов и тех сложных правил вращения которым они подчиняются.



Рисунок 5: Код отвечающий за вращение деталей

На рисунке 5 описаны действия нужные для вращения объекта. Размеры объекта внутри массива изменяются при помощи команды SillyQuaternion() так как она возвращает только положительные значения. Вектор силы двигателя (если есть) меняется при помощи команды dumbQuaternion(), эта команда поддерживает негативные значения и корректно конвертирует значения в зависимости от направления вращения. 3D модель объекта поворачивается при помощи стандартной команды предоставляемой движком, я использую команду RotateAround для вращения модели объекта относительно выбранной пользователем оси координат.

После поворота объекта и всех его значимых характеристик на 90 градусов осталось лишь провести действия, аналогичные тем которые проводятся при перемещении объекта. Нужно очистить уже занятые объектом клетки и затем проверить умещается ли объект в своем текущем положении.

### 1.2.6 Крепление

Если деталь висит в воздухе и ничего не касается - значит она должна быть либо удалена, либо вычтена из вычислений. Для того чтобы определить прикреплена ли деталь нужно проверить касается ли она основы корабля, которой является рубка, или же касается ли она детали о которой известен факт ее закрепленности.

При добавлении детали она проверяет своих соседей, и если она находит закрепленную деталь или основание то она закрепляется. При удалении, перемещении и повороте нужен полный перерасчет всего корабля.

В будущем я планирую реализовать систему креплений при помощи структуры дерева. Такая структура не только ускорит расчеты, но и упростит работу с креплениями.

### 1.2.7 Функциональные компоненты

Я решил что для начала нужно создать три функциональных компонента, это двигатели, камеры и луч притяжения. Этих деталей достаточно для создания самого базового геймплея.

Для функциональных компонентов мною был создан отдельный абстрактный класс являющийся потомком обычного, нефункционального компонента. Классы потомки этого нового класса будут обладать ссылками на свое отображение в списке деталей а также функцией ComponentAction(), отвечающей за выполнение некоего действия.

Активация компонентов будет производиться через игровой интерфейс нажатием кнопок.

### 1.2.8 Сохранение космического корабля

При сохранении космического корабля каждая его деталь будет представлена в виде своего типа, координаты и градуса поворота. Эти значения будут записаны в класс сохранения, а после конвертированы в JSON. Встроенные инструменты Unity и C# не имеют продвинутого функционала работы с этим типом файла. Например, они не способны превратить ничего сложнее списка в Json файл. Список классов, или же словарь нельзя конвертировать с их помощью.

Я мог установить библиотеку Newtonsolt Json.Net, но решил отказаться от этой идеи и ограничиться работой со списками.

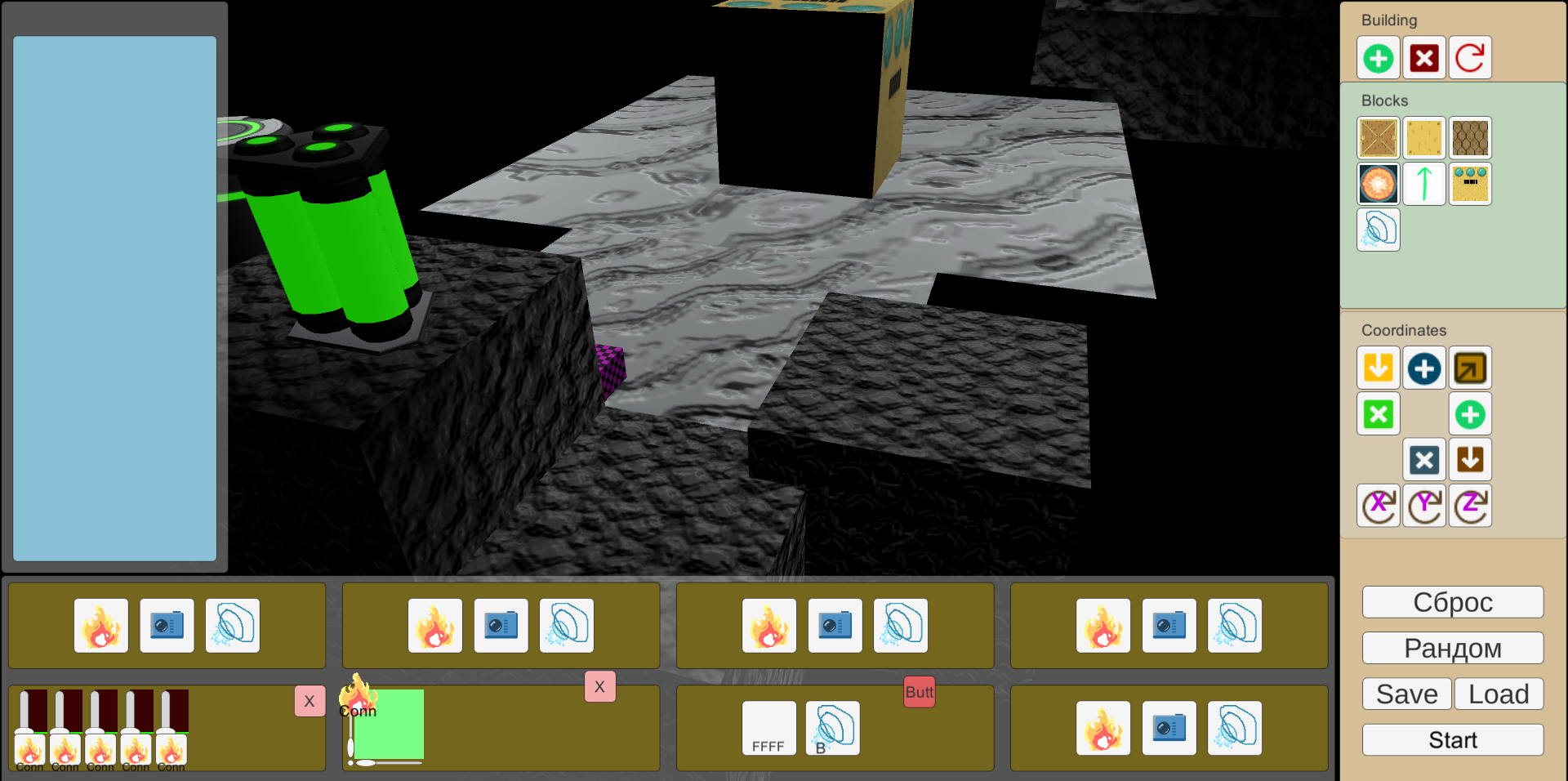
Как только приходит время загрузить файл сохранения игра стирает текущую постройку корабля и конфигурацию его интерфейса и заменяет на то что было обнаружено в Json, поочередно заново генерируя каждый строительный блок корабля.

Похожая операция будет производиться над динамическим интерфейсом, но вместо того чтобы работать с 3D матрицей процесс будет происходить в 2D. Проблема осложняется тем что элементы интерфейса должны сохранять воспоминания о том с какими компонентами корабля они связаны. Для этого каждый элемент будет сохранен в список типов, и к каждому элементу этого списка будет привязано 5 интерактивных объектов (которые могут быть null). При загрузке объекта, что является вторичным созданием, эти 5 элементов отправятся в соответствующие места.

Это накладывает искусственное ограничение в 5 элементов, но в целом является неплохим быстрым решением.

## 1.3 Интерфейс.

### 1.3.1 Статичный интерфейс



Пользовательский интерфейс будет предоставлять доступ к базовым операциям работы с блочной структурой, а также позволять сохранять и загружать полученные конфигурации.

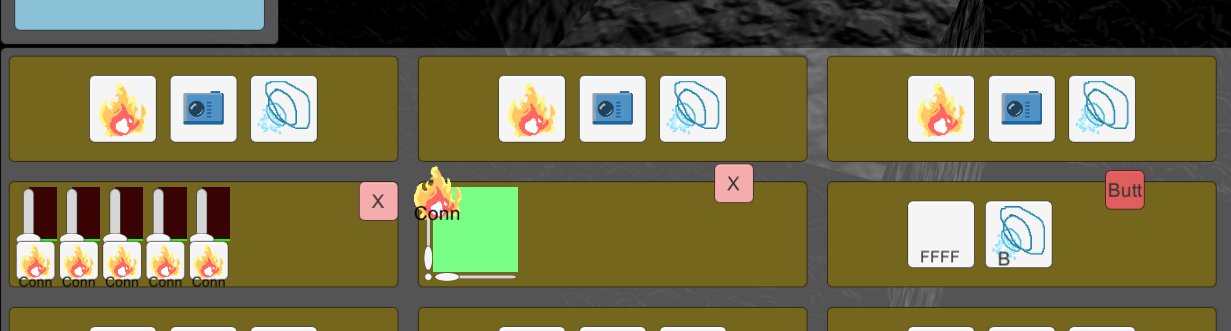
Если посмотреть на правую сторону, то можно увидеть, по порядку сверху вниз:

* Кнопки добавления, удаления и обновления;
* Зеленоватое поле с палитрой доступных блоков;
* Поле “Coordinates” позволяющее выбрать инструмент выборки блока нажатием на желтую левую верхнюю кнопку. Все остальные кнопки проводят операции над выбранным блоком, это передвижение компонента на 1 клетку по некоей оси координат и поворот также относительно некоей оси.
* Кнопки с текстом производят массовые действия. Очевидно что “сброс ” обнуляет весь прогресс постройки игрока, кнопка “Рандом” использует алгоритм Wave Function Collapse для создания 3D модели корабля, эта функция также использовалась мной в практическом отчете. Функции Save и Load очевидно позволяют сохранить текущую конфигурацию космического корабля и интерфейса, но сейчас они хранят лишь одну конфигурацию.
* Кнопка ниже (не видна на картинке) с надписью “начать игру” отвечает за начало игры. Она разблокирует возможность корабля к передвижению и расширяет интерфейс на весь экран, не давая игроку видеть.
* Также мною были созданы кнопки Quit, которая закрывает приложение, и кнопка BulkGeneration, которая генерирует установленное количество уникальных моделей для космического корабля. Она была незаменима при выполнении практической работы.

### 1.3.2 Динамический интерфейс

Интерфейс будет состоять из сетки 4 на 3 клетки, в каждую клетку будет возможность назначить какой-либо пресет нажатием кнопки. Данные пресеты можно будет заполнять данными о компонентах.

На картенке ниже можно увидеть как это будет выглядеть без назначенных компонентов. Верхний ряд это клетки которые еще не были определены игроком, нижний ряд это клетки выбравшие каждый из трех доступных на данный момент вариантов.

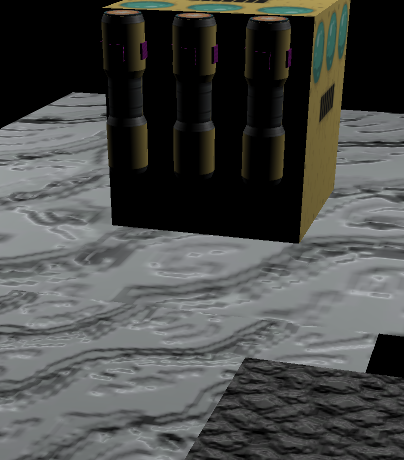
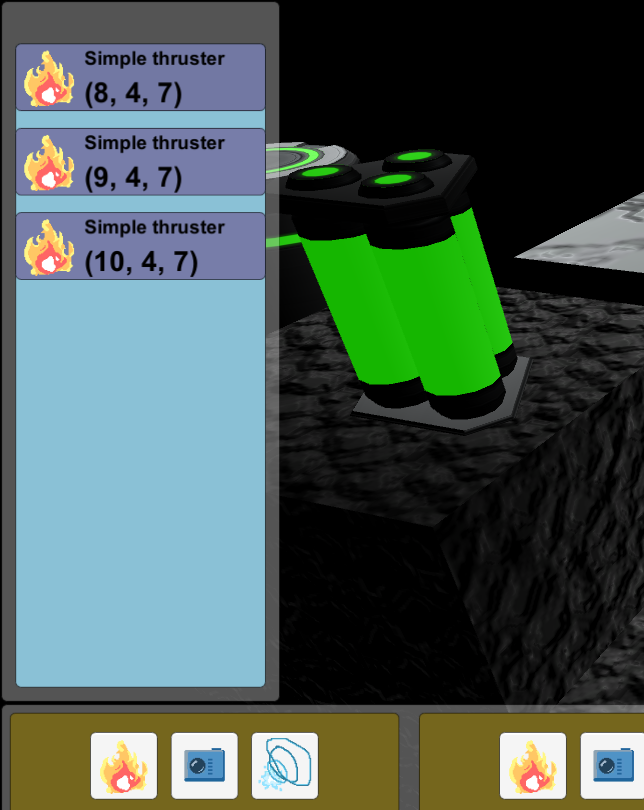


Самая левая заполненная клетка позволяет игроку управлять мощностью до 5 двигателей, клетка посередине обладает небольшим экранчиком для отображения видео с камеры а самый правый компонент является универсальным и позволяет назначить некий объект кнопке, которая активирует специальную функцию объекта. Это довольно бесполезно для двигателей, но может применяться для включения и выключения фонариков или же обновления луча притягивания.

Сейчас интерфейс маленький, но он будет расширен до более приятных глазу значений как только игрок закончит редактировать корабль.

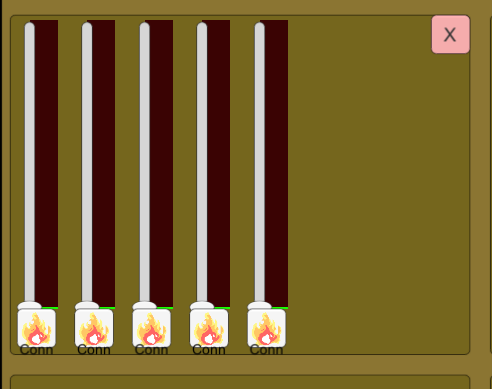
### 1.3.3 Динамический список

В левой части экрана расположено голубое окошко, оно необходимо для отображения интерактивных объектов. Все объекты обладающие функционалом будут отображаться в нем, сейчас в нем можно увидеть двигатели, камеры и луч притяжения.



При нажатии на элементы списка будет подсвечен соответствующий ему объект космического корабля.

### 1.3.4 Динамический функционал



У большинства функциональных объектов интерфейса будет иметься свой элемент-коннектор. На картинке выше он выглядит как белый квадратик с огнем, под каждым из слайдеров. При нажатии на него курсор игрока выделит элемент а сам элемент будет ожидать следующего клика, и в зависимости от результата нажатия назначит соответствующий элемент определенному слайдеру.

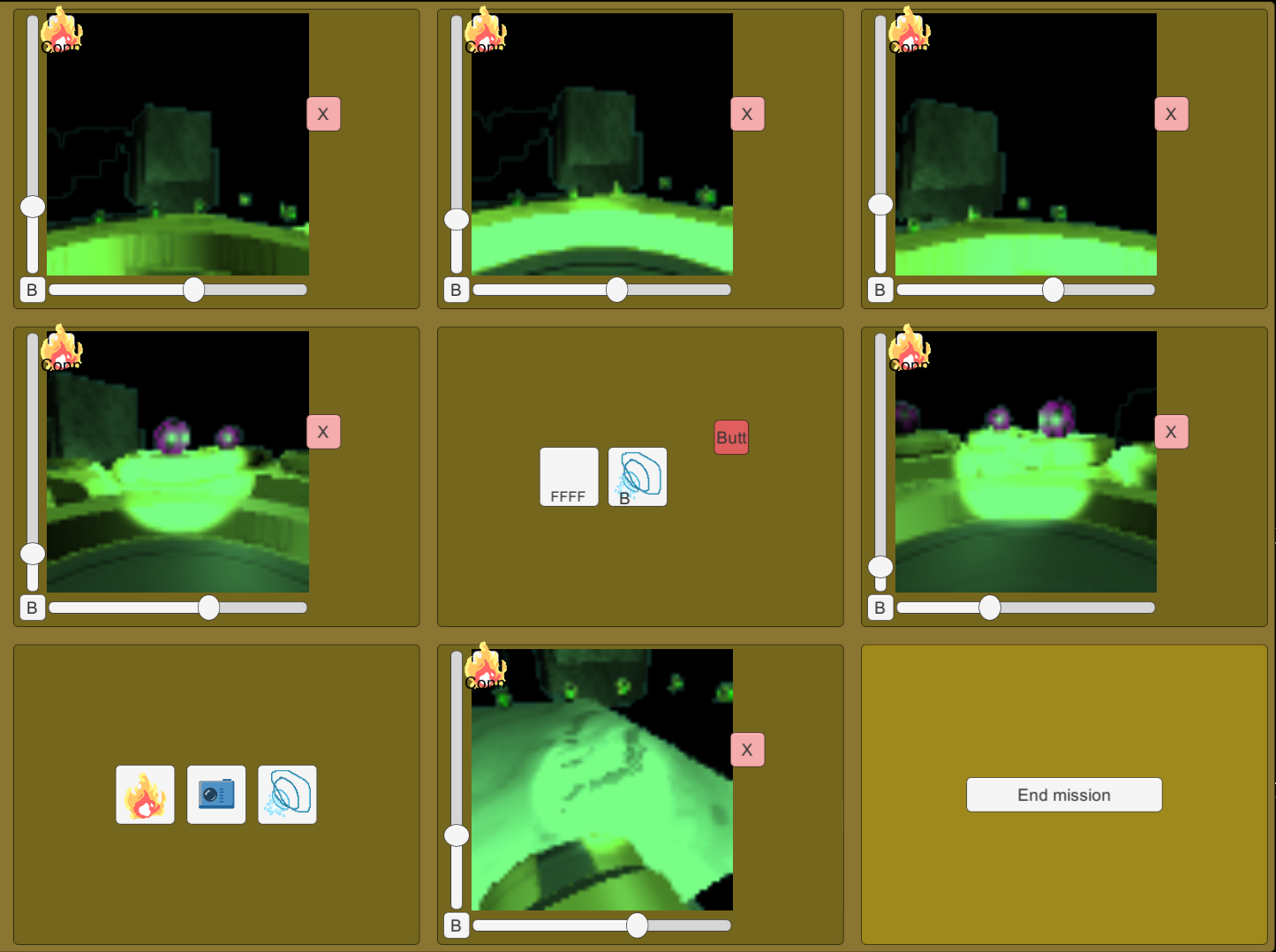
Всего я смог придумать 4 различных результата нажатия на элемент-коннектор:

* Первое нажатие: Назначает элемент камере игрока, ждет следующего клика игрока, следующий клик обязательно принадлежит к следующим трем категориям.
* Второе нажатие: Игрок вторично нажал на элемент, в этом случае я считаю это за желание обнулить значение элемента.
* Потенциальная выборка: элемент на который нажал игрок потенциально можно присоединить. В этом случае выполняются проверки на совместимость элемента.
* Нажатие в пустоту: Игрок нажал на пустое поле не имеющее кнопок, вероятнее всего игрок передумал о своих действиях, мои действия похожи на второй вариант.

Если действие игрока оказалось выборкой то элемент-коннектор оповещает элемент-активатор (слайдер, кнопку) и компонент космического корабля. В некоторых ситуациях он становится связующим звеном между элементами, а иногда лишь тем кто сводит два элемента вместе.

### 1.3.5 Взаимодействие отображения и объектов

Через созданную коннектером связь происходит общение между интерфейсом и деталями корабля. Так например при изменении слайдера скорости двигателя тот начнет постепенно увеличивать свою скорость чтобы соответствовать установленной скорости, но одновременно с этим он будет “заходить” внутрь элемента интерфейса и менять отображение своей текущей скорости на спидометре. На левой картинке можно увидеть ручку слайдера, отвечающую за желанную скорость, и полосы разных цветов отображающие текущую скорость двигателя.



А вот так вот выглядит интерфейс камер. Вертикальные слайдеры отвечают за подъем камеры, а горизонтальные за поворот. Всего на этом корабле установлено 6 камер видеонаблюдения. Три верхние камеры смотрят на один и тот же объект, камеры посередине смотрят на другие камеры, самая нижняя камера смотрит на притягивающий луч, кнопка активации которого находятся в середине интерфейса.

### 1.3.5 Интерфейс заключение

По моему мнению интерфейс получился классным. Он обладает всем функционалом который я ожидал от него увидеть, и исполняет все функции которые я запланировал. Это довольно необычный стиль для интерфейса, который может стать потенциальной “изюминкой” успешной игры.

## Заключение

Я смог выполнить большую часть поставленных мной целей, и в итоге получил в хорошем смысле медленную, медитативную игру с необычным визуальным стилем и огромным пространством для творчества. Благодаря тому как я организовал системы функциональных элементов и модульного интерфейса игровой контент будет нетрудно расширить большим количеством деталей с альтернативными характеристиками.

На данный момент в проекте точно найдется некоторое количество багов и недоработок, но в ближайшем времени я их выслежу и починю.

# **Список использованной литературы**

1. Документация языка C# - <https://learn.microsoft.com/en-us/dotnet>
2. Документация Unity - <https://docs.unity3d.com/Manual>
3. YouTube канал GDC - <https://www.youtube.com/@Gdconf>
   1. Отличное объяснение геометрических функций / CSG - <https://youtu.be/Iqmg4gblreo>
4. YouTube канал Dave / GameDevelopment - <https://www.youtube.com/@davegamedevelopment>
5. YouTube канал ConsidetaCode - <https://www.youtube.com/@ConsideraCore>
6. YouTube канал Brackeys - <https://www.youtube.com/@Brackeys>
7. YouTube канал Jimmy Vegas - https://www.youtube.com/@JimmyVegasUnity
8. Ссылка на мой GitHub репозиторий - <https://github.com/JleHuBbluKoT/3D-Meshes>
9. Ссылка на оригинал документа в Google Docs - [Томилов Ярослав экзаминационная](https://docs.google.com/document/d/1_B4GBB-crKUIc4-RFQrv72AfPxB6Y_QBH2-Kp-BaAsM/edit?usp=sharing)
10. Не один раз выручивший меня сайт - <https://stackoverflow.com/>