# Игровая логика Mars Colony

### 

[**Игровая логика Mars Colony**](#_2lbnh07xh4r8) **1**

[Краткое описание GD, которого автор хотел достичь](#_vaz30ecjmujj) 1

[1. Настраиваем проекты](#_bjryduiqxrfb) 2

[2. Создаем ядро — базовые сооружения](#_6f81e5xo6tea) 3

[3. Добавляем и тестируем первые команды — создать карту и разместим объекты на карте](#_bubp9ke3b9kd) 7

[4. Выносим настройки строений и модулей в отдельный файл](#_gbz3tt9smwpj) 8

[5. Добавляем течение времени](#_ch3yr4ge8npu) 12

[6. Добавляем Constructible](#_93tdpbql01u) 14

[7. Добавляем ресурсы](#_29coleuqxf29) 15

[8. Добавляем цикл производства](#_pc935h437atb) 17

[9. Настройки в JSON](#_c01uo7fvv6az) 21

[**View**](#_v2zijspd6yzo) **22**

[10. Контроллер камеры](#_kuhnq2p2mj9n) 22

Нашел интересную статью на хабре, которая может быть очень полезна. Я уже писал про нее раньше, но в этот раз хочу все подробно расписать, т.к. считаю, что на базе этого можно построить игровую логику. Ссылки привожу ниже: <https://habr.com/ru/post/322258/> - Часть ½ <https://habr.com/ru/post/322268/> - Часть 2/2

Репозиторий проекта <https://github.com/Sudar1977/MarsColonyGameLogic>

В связи с выходом игры SpaceLab на GreenLight была выложена серию статея о разработке игры на C#/Unity. Она основана на реальном опыте её разработки и немного отличаться от стандартных гайдов для новичков.

<https://steamcommunity.com/sharedfiles/filedetails/?id=868755125>

$ git push -u origin main

$ git remote add origin https://github.com/Sudar1977/StrategyGB.git

$ git commit -m "first commit"

$ git add .

Автор шаг за шагом рассказывает о создании движка, на котором будет работать игровая логика нашей экономической стратегии. Есть ссылка на репозиторий <https://github.com/theshock/GameLogicArticle> , где каждый пункт добавлен отдельным коммитом.

Есть ссылка на видео и бесплатное скачивание.

Автор сразу предупреждает, что у него нету цели идеально применить огромное количество паттернов или описать подход к методологии TTD. В статье он старается писать читабельный, поддерживаемый и безбажный код, как он писался бы в жизни. Возможно, людям имеющим огромный скилл в C# и написании игр данная статья покажется очевидной. Тем не менее, вопрос о том, как писать гейм-логику возникает довольно часто.

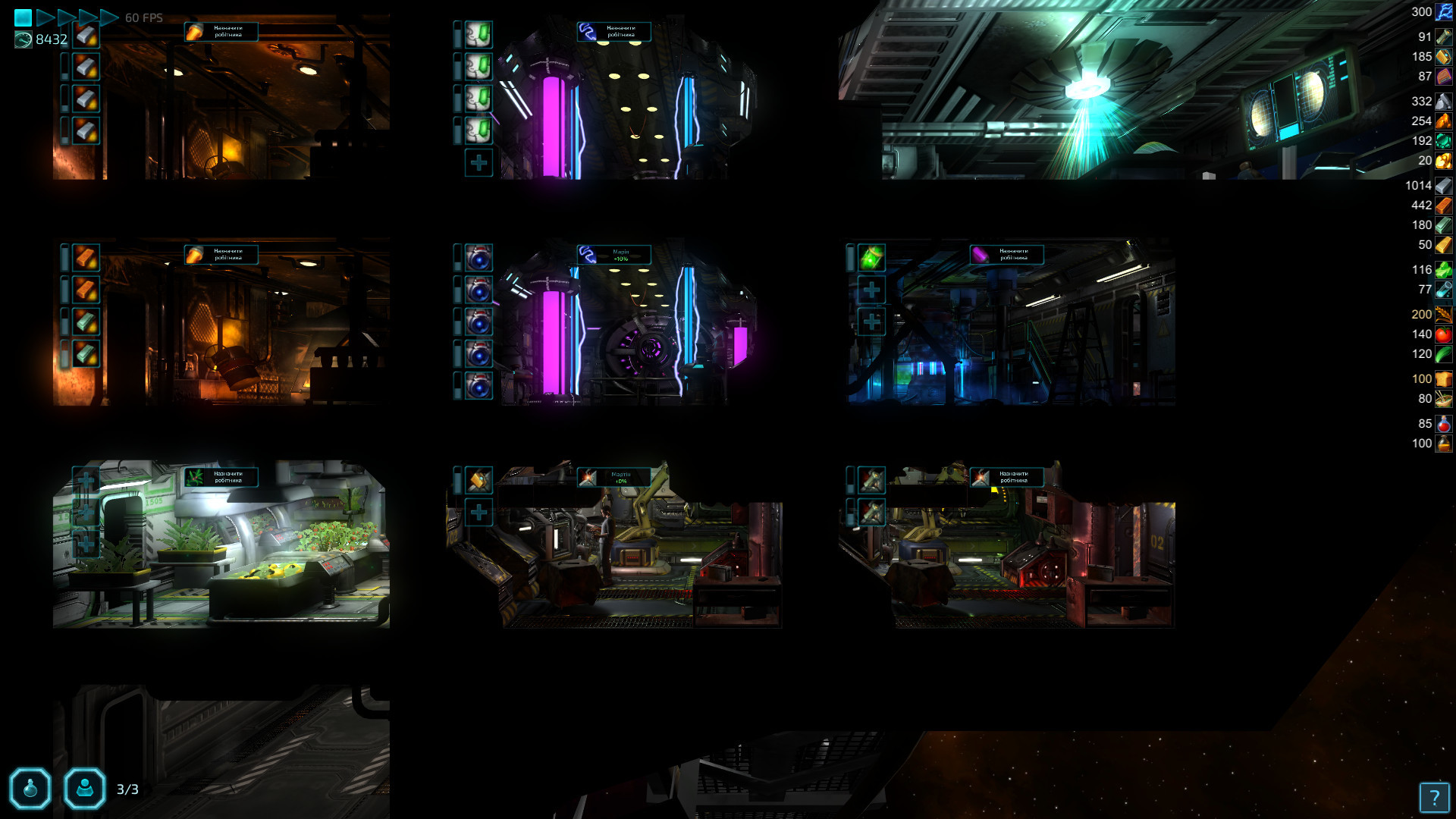
## Краткое описание GD, которого автор хотел достичь

1. Игрок управляет кораблем. В корабле можно выстраивать комнаты, в комнатах можно добавлять в слоты модули.

Адаптация под Марсианскую Колонию: На платенте Марс высаживаются колонисты и основывают колонии (аналогия с комнатами), в каждой колонии есть ландшафт, здания, ресурсы и пр. Далее буду приводить аналогии

2. Для постройки чего-либо необходимо потратить ресурсы и подождать время.

Через полгода разработки результат по мнению автора должен выглядеть как-то так) Рисунок 1.



## 1. Настраиваем проекты

На первых порах Unity Editor нам не понадобится — мы пишем ГеймЛогику. Открываем VS и создаем два проекта: GаameLogic и LogicTests (Unit Tests Project). В первом мы будем писать собственно логику игры на чистом C# не используя namespace Unity, второй будет тестить нашу логику встроенной тест-тулзой. Добавим в GameLogic первый класс Core и напишем первый тест, чтобы проверить нашу связку:

public class Core

{

public static void Main () {}

public Core () {}

}

[TestClass]

public class Init

{

[TestMethod]

public void TestMethod1 ()

{

Assert.IsInstanceOfType(new Core(), typeof(Core));

}

}

## 2. Создаем ядро — базовые сооружения

“Что ж, это указывает, что настроили мы корректно и можно переходить к программированию логики.

Итак, разберемся с нашим гейм-дизайном. У нас есть корабль (Ship), в нем комнаты (Room), в каждую комнату может быть построено строение (Building), а в каждом строении могут быть модули (Module). Конечно, Room и Building можно было бы объединить в одну сущность, но далее такое разделение нам только поможет.”

**Задача первой версии**

Играбельный прототип, в который будут включены 19 базовых зданий, будет реализована основная экономика и основной сюжет. При этом параметры игрового процесса, такие как стоимость зданий, текстовые сообщения в игре и т.п. (эта цель будет подробно описана ниже) должны меняться без участия программиста, силами тестировщика. Т.е. они должны быть вынесены в отдельные файлы конфигурации или изменяемы из самой игры.

**Цель игры**

Цель игры до 100 хода накопить определенную сумму требуемого ресурса (эти данные должна быть возможность изменить без участия программиста). По умолчанию: 1000 пищи. Если на 100 ход у игрока нет требуемого количества, игра заканчивается поражением. Если есть, игра оканчивается победой. О чем игроку приходит сообщение в игре.

**Игровое поле**

Представляет собой карту поверхности марса, на которой располагаются в самом начале игры ресурсы и объекты – те что мы уже включили в игру). Саму карту должна быть возможность изменять без участия программиста (считывать из текстового файла).

У нас есть класс Марсколония (Colony) у которой есть игровое поле (Map), на котором может быть построено строение (Building), также могут располагаются залежи ресурсов (DepositsResource) и особенности ландшафта (LandscapeFeatures ).

На карте отражены залежи ресурсов, строения, особенности ландшафт.

Предлагается сделать в виде словаря объектов.

*// Каждое здание, ресурс и т.д должен иметь свою сообтвенную позицию*

*//ToDo пока закодировано числом, потом массивом - координатой*

private readonly Dictionary<int, Building> buildings = new Dictionary<int, Building>();

Аналогия - у нас есть планета Mars (может лучше Planet?), на ней основываются колонии Colony (вместо Room), в каждой колонии есть свой ландшафт, ресурсы, здания и т.д. - это по сути дела есть карты Map (в SpaceLab это Building, тут может быть путаница!) на каждой карте размещены объекты ObjectOnMap для простоты у нас все что на карте будет все типа ObjectOnMap, ресурсы, ландшафт, марсоход, потом нужно будет разнести по классам.

В статье позиция здания-модуля закодирована одним числом position у нас же на карте нужны две координаты x,y. Пока для простоты пусть координата будет закодирована 1,10,100, а координата y 1000,10000,100000. Т.е. Мы получим поле 999,999. Пример координата (x,y) = (1,2) будет закодирована числом position=001002 (x,y) = (517,62) position=517062 (x,y) = (1,999) position=001999

Впоследствии можно будет закодировать массивом int[] из двух элементов.

Другим, ограничением является, что в SpaceLab одной комнате размещается ОДНО строение, т.е. в одной колонии одна карта, что не очень удобно. Нужно будет это переделать, это не очень сложно, но пока мы оставим как есть. Это ограничение влечет за собой, что на карте нельзя разместить ресурс и ровер. Т.е. При установке ровера ресурс будет удален. Поэтому пока будут два разных ровера каждый на свой ресурс. Когда ровер будет убран ресурса не будет. Но для начала оставим как есть...

UPD/ Попробуем убрать это ограничение,

// Добавим по несколько карт в одну колонию различного типа

private readonly Dictionary<MapType, Map> maps = new Dictionary<MapType, Map>();

public IEnumerable<Map> Maps

{

get { return maps.Values; }

}

public Map GetMap(MapType type)

{

return maps.ContainsKey(type) ? maps[type] : null;

}

public void SetMap(MapType type, Map map)

{

maps[type] = map;

}

ToDo. Теперь есть параллельно две системы наши карты и карта. Нужно покрыть тестами карты и сделать экземпляры карт. Нужно добавить типы зданий и карт (добавлены).

“Для всех этих сооружений я создам отдельный namespace Architecture и базовые классы. А также enum для индексов комнат. Многие вещи, которые мы сейчас делаем — временные и необходимы, чтобы запустить первый тест гейм-логики.”

У нас вместо индексов комнат будут индексы карт или “слои”.

public enum MapType

{

Empty,

Objects,//пока будет карта только этого типа, т.е. все объекты на карте

Buildings,

LandscapeFeatures,

DepositsResource,

Rovers,

PowerPlant //рудимент старого кода нужно убрать

}

public enum ObjectOnMapType

{

//строения Buildings

AutominerMetallOre**,**

AutominerRareEarthOre**,**

MiningComplex**,**

ConcentratingFactory**,**

SolarPanel**,**

NuclearPlant**,**

HeatGenerator**,**

WaterRig**,**

Greenhouse**,**

Conservatory**,**

HydroponicsFoodProduction**,**

HydroponicsVegetableRawProduction**,**

VacuumSmelter**,**

ElectrrolysSmelter**,**

ChemicalLaboratory**,**

PolymerSyntesis**,**

//марсоходы Rover

RoverOnMetallOre**,**

RoverOnRareEarthOre**,**

//залежи ресурсов DepositRecources

DepositIce**,**

DepositMetallOre**,**

DepositRareEarthOre**,**

//рудимент старого кода нужно убрать

Generator**,**

Furnace**,**

Digger**,**

Miner

}

public class Core

{

public static void Main () {}

public readonly Mars Mars = new Mars();

public Core ()

{

Mars.CreateEmptyColonies();

}

}

public class Mars

{

*// Временно добавим некоторое количество колоний = 2*

public readonly int ColonyLimit = 10;

private readonly List<Colony> colonies = new List<Colony>();

public IEnumerable<Colony> Colony {

get { return ; }

}

public void CreateEmptyColony()

{

for (var i = 0; i < ColonyLimit ; i++) {

colonies.Add(new Colony (i));

}

}

public Colony GetColony (int index)

{

return colonies[index];

}

}

public class Colony

{

public readonly int Index;

*// каждая комната является пристанищем для строения*

public Map Map { get; set; }

public Map (int index)

{

Index = index;

*// и по-умолчанию - это пустое строение*

Map = new Map(MapType.Empty);

}

}

public class Map

{

*// Ограничим количество зданий, которые можно разместить на карте*

public readonly int ObjectsOnMapLimit= 1000000;

public readonly MapType Type;

*// Каждое здание, ресурс и т.д должен иметь свою сообтвенную позицию*

*//ToDo пока закодировано числом, потом массивом - координатой*

private readonly Dictionary<int, ObjectOnMap> objectsOnMap = new Dictionary<int, ObjectOnMap>();

public IEnumerable<ObjectOnMap> ObjectsOnMap{

get { return objectsOnMap.Values; }

}

public Map (MapType type)

{

Type = type;

}

public Building GetObjectonMap(int position)

{

return objectsOnMap.ContainsKey(position)

? objectsOnMap[position]

: null;

}

public void SetObjectonMap(int position, ObjectOnMap objectOnMap)

{

if (position < 0 || position >= ObjectsOnMapLimit) {

throw new IndexOutOfRangeException(

"Position " + position + " is out of range [0:" + ObjectOnMapLimit+ "]"

);

}

objectsOnMap[position] = objectOnMap;

}

}

public class Building

{

public readonly BuildingType Type;

public Building (BuildingType type)

{

Type = type;

}

}

## 3. Добавляем и тестируем первые команды — создать карту и разместим объекты на карте

“Теперь мы сможем написать первую «фичу» — постройка строения и постройка модуля в нем. Все подобные действия я буду описывать отдельным классом, который будет наследоваться от класса Command:”

У нас соответственно будет соответственно создание карты и размещение объектов на ней

public abstract class Command

{

public Core Core { get; private set; }

public bool IsValid { get; private set; }

public Command Execute (Core core)

{

Core = core;

IsValid = Run();

return this;

}

protected abstract bool Run ();

}

“И хотя сейчас даже такая маленькая структура излишня — чуть позже благодаря ей мы прикрутим необходимые нам события. А существование каждого атомарного действия в отдельной команде позволит нам их комбинировать. Напишем наши первые два действия:”

В создании карты наша логика “расходится”, т.к. карта у нас пока одна и должна быть создана типа “Objects”.

Также мы позволим строить добытчики ресурсов на месте ресурса, заменяя его собой, пока у нас не реализованы слои.

public class MapConstruct : Command

{

public readonly Colony Colony;

public readonly Map Map;

public MapConstruct (Colony colony, Map map)

{

Colony = colony;

Map = map;

}

protected override bool Run ()

{

*// Нельзя строить там, где уже что-то есть*

if (Colony.Map.Type != MapType.Empty) {

return false;

}

*// Нельзя строить пустую комнату*

if (Map.Type == MapType.Empty) {

return false;

}

Colony.Map = Map;

return true;

}

}

public class BuildingConstruct : Command

{

public readonly Map Map;

public readonly Building Building;

public readonly int Position;

public ModuleConstruct (Map map, Building building, int position)

{

Map = map;

Building = building;

Position = position;

}

protected override bool Run ()

{

if (Map.Type == MapType.Empty) {

return false;

}

if (Position < 0 || Position >= Map.BuildingLimit) {

return false;

}

if (Map.GetBuilding(Position) != null) {

return false;//добавить логику добычи ресурсов???

}

Map.SetBuilding(Position, Building);

return true;

}

}

“Пришло время посмотреть, работает ли наш движок. В тестах создаем ядро, пробуем построить комнату, а в нее пытаемся построить модуль. Кроме этого стоит добавить проверку, что нельзя построить то, чего гейм-логика не должна позволять строить:”

В нашем случае создаем ядро Core, Mars колонию Colony, в ней карту строений Map и здание Building.

[TestClass]

public class Architecture

{

[TestMethod]

public void CorrectConstruction ()

{

var core = new Core();

var colony = core.Mars.GetColony(0);

Assert.AreEqual(MapType.Empty, Colony.Map.Type);

Assert.AreEqual(0, colony.Map.Buildings.Count());

Assert.IsTrue(

new MapConstruct(

colony,

См. гитхаб }

## 4. Выносим настройки строений и модулей в отдельный файл

К счастью, наши тесты прекрасно проходятся. Теперь нам необходима возможность линейно расширять количество строений и модулей — для этого необходимо сделать следующее:

В нашем случае карт (хотя наверно для карт сейчас это особо и не нужно) и строений (нужно).

1. Создать конфигурацию для строений и модулей — "class MapConfig" и "class ObjectOnMapConfig", именно они будут хранить все настройки наших сооружений.
2. Map и Building при создании должны принимать соответствующие настройки
3. Сделать фабрику для создания карт и строений
4. Добавить настройки для нескольких карт и строений
5. Адаптировать существующий код под новые входные данные

Поскольку у нас пока только одна карта Objects, AvailableObjectsOnMap пока не особо полезны, но в будущем пригодится. Например, на карте ресурсов можно размещать только ресурсы. На карте строений только строения. На карте роверов, только роверы. На карте ландшафта только элементы ландшафта. Хотя это можно сделать через типы. Сделать базовый тип MapObjects.

Было бы очень удобно, если здание знало бы на каком месте его можно строить, например добытчик ресурса можно можно было бы строить на месте необходимого ресурса и т.п. AvailablePlaceForBuildig

*// Создаем конфиги*

public class MapConfig

{

public MapType Type;

*// Теперь никаких констант*

public int ObjectsOnMapLimit;

*// На карте можно размещать только определенные здания*

public ObjectOnMapType[] AvailableObjectsOnMap;

public class ObjectOnMapConfig

{

public ObjectOnMapType Type;

*// На карте можно размещать только определенные здания*

public BuildigType[] AvailableObjectsForBuildig;

}

public class Map

{

*// ...*

public readonly MapConfig Config;

*// ...*

*// В конструкторе принимаем конфиг, а не индекс*

public Map(MapConfig config)

{

Type = config.Type;

BuildingLimit = config.BuildingsLimit;

Config = config;

}

}

public class ObjectOnMap

{

*// ...*

public readonly ObjectOnMapConfig Config;

*// В конструкторе принимаем конфиг, а не индекс*

public ObjectOnMap(ObjectOnMapConfig config)

{

*// ...*

Type = config.Type;

Config = config;

}

}

“Как можно понять, теперь наш код нерабочий. Для того, чтобы не таскать каждый раз с собой конфиги создадим фабрику, которая будет выпускать наши сооружения зная только их тип. Я знаю, что название пока слишком общее, но мы всегда с легкостью можем его переименовать благодаря IDE, так же, как и разделить на две фабрики:”

public class Factory

{

public Map CreateMap(MapType type) //наверно лучше CreateMap

{

throw new Exception("Not implemented yet");

}

public Building ProduceBuilding (BuildingType type)

{

throw new Exception("Not implemented yet");

}

}

*// А также добавим нашу фабрику в ядро:*

public class Core

{

*// ...*

public readonly Factory Factory = new Factory();

public Core ()

{

*// В аргумент метода передаем фабрику*

Mars.CreateEmptyColony(Factory);

}

}

*// Марс теперь принимает фабрику в качестве аргумента:*

public class Mars

{

*// ...*

public void CreateEmptyColony (Factory factory)

{

for (var i = 0; i < ColonyLimit; i++) {

colony.Add(new Colony(i, factory.CreateMap(MapType.Empty)));

}

}

}

*// А карта - принимает строение по-умолчанию:*

public class Colony

{

*// ...*

public Colony (int index, Map map)

{

Index = index;

Map = map;

}

}

Сейчас IDE указывает, где мы имеем ошибки — заменим там вызов конструктора на использование фабрики.

*// в тестах*

new Map(Type);

*// заменяем на*

core.Factory.CreateMap(Type);

Type - в разных строках разный MapType.PowerPlant,MapType.Empty

*// в тестах*

new ObjectOnMap(Type);

*// заменяем на*

core.Factory.ProduceBuilding(Type);

Type - в разных строках разный ,ObjectOnMapType.Generator

И хотя сейчас код корректен — при запуске наших тестов мы словим "Not implemented yet". Для этого вернемся к нашей фабрике и реализуем несколько карт и строений.

Пока оставим типы зданий как в статье для проверки, ниже сделаем свои тесты со своими типами

public class Factory

{

private readonly Dictionary<MapType, MapConfig> maps = new Dictionary<MapType, MapConfig>() {

{

MapType.Empty, new MapConfig()

{

Type = MapType.Empty

}

},

{

MapType.PowerPlant, new MapConfig()

{

Type = MapType.PowerPlant,

ObjectsOnMapLimit = 1000000,

ObjectsOnMapLimit= new[]{ ObjectOnMapType.Generator }

}

},

{

MapType.Landscape, new BuildingConfig() {

Type = BuildingType.Landscape,

ModulesLimit = 1000000,

AvailableModules = new[]{ ModuleType.Mount, ModuleType.Crater }

}},

{ BuildingType.Resourcers, new BuildingConfig() {

Type = BuildingType.Resourcers,

ModulesLimit = 1000000,

AvailableModules = new[]{

ModuleType.Ice,

ModuleType.Metall

}

}}

свои типы карт:

};

}

private readonly Dictionary<BuildingType, BuildingConfig> buildings = new Dictionary<BuildingType, BuildingConfig>() {

{ BuildingType.RoverMetall, new ModuleConfig() {

Type = ModuleType.RoverMetall

}},

{ ModuleType.WaterRig, new ModuleConfig() {

Type = ModuleType.WaterRig

}},

{ ModuleType.Digger, new ModuleConfig() {

Type = ModuleType.Digger

}},

{ ModuleType.Miner, new ModuleConfig() {

Type = ModuleType.Miner

}}

};

Поскольку задний 19 а объектов на карте еще больше в дальнейшем мы будем их загружать их из файлов, но уже сейчас фабрика занимает более 150 строк, поэтому разобьем это класс на более мелкие. Создадим папку Factory, куда поместим наши фабрики. Сделаем отдельные фабрики для конфигов карт FactoryConfigMaps, объектов различных карт FactoryConfigObjectsOnMap.

## 5. Добавляем течение времени

**Игровой процесс**

Игра делиться на ходы, в течении каждого хода игрок может совершить несколько действий. Этими действиями являются: строительство здания, покупка/продажа ресурсов. Здания, построенные игроком могут использовать ресурсы и производить их. Пересчет произведенных/используемых ресурсов происходит при нажатии кнопки «*конец хода*». В начале хода, или в результате определенного события игра может вывести для игрока текстовое сообщение, которое сообщает игроку некоторую информацию.

***Цель игры***

Цель игры до 100 хода накопить определенную сумму требуемого ресурса (эти данные должна быть возможность изменить без участия программиста). По умолчанию: 1000 пищи. Если на 100 ход у игрока нет требуемого количества, игра заканчивается поражением. Если есть, игра оканчивается победой. О чем игроку приходит сообщение в игре.

**Строительство**

Игрок может выбрать в меню строительство какого-либо здания, при этом в меню строительства должна отображаться цена строительства в ресурсах и время строительства в ходах. Это можно реализовать всплывающей подсказкой либо информационной плашкой в самом интерфейсе. Данные в это окно или подсказку игра должна брать из того же файла, откуда и параметры постройки.

Эти параметры должны быть изменяемы без участия программиста.

Компьютеры дискретны. И все игры дискретны. Если говорить просто, то представим, что все игры — пошаговые. У большинства игр шаги пропускаются автоматически и 60 раз в секунду. Такие игры называются риалтайм. Я понимаю, что это очень грубо, но для реализации гейм-логики довольно удобно представлять, что ваша игра — пошаговая и мыслить такими категориями. А потом уже на клиенте можно запустить tween между двумя состояниями и юзеру будет красиво и игра будет работать быстро. Для начала введем понятие хода:

Для начала введем понятие хода:

public class Turns

{

public int CurrentTurn { get; private set; }

internal void NextTurn ()

{

CurrentTurn++;

}

}

public class Core

{

public readonly Turns Turns = new Turns();

}

А также введем команду, которая позволяет переключать хода. Я сразу добавил команду, которая позволяет переключить несколько ходов — будет довольно удобно во время тестирования. В тестах одним выстрелом покроем сразу двух зайцев.

public class NextTurn : Command

{

protected override bool Run ()

{

*// Именно тут будет вся логика хода*

Core.Turns.NextTurn();

return true;

}

}

public class NextTurnCount : Command

{

public const int Max = 32;

public readonly int Count;

public NextTurnCount (int count)

{

Count = count;

}

protected override bool Run ()

{

if (Count < 0 || Count > Max) {

return false;

}

for (var i = 0; i < Count; i++) {

var nextTurn = new NextTurn().Execute(Core);

if (!nextTurn.IsValid) return false;

}

return true;

}

}

[TestClass]

public class Turns

{

[TestMethod]

public void NextTurnsCommand ()

{

var core = new Core();

Assert.AreEqual(0, core.Turns.CurrentTurn);

Assert.IsTrue(

new NextTurnCount(4)

.Execute(core)

.IsValid

);

Assert.AreEqual(4, core.Turns.CurrentTurn);

}

}

## 6. Добавляем Constructible

Давайте теперь что-то привяжем к течению времени. Пусть постройки и модули строятся не сразу, а несколько ходов (зависимо от конфигурации). Для начала во все настройки добавим пункт ConstructionTime. Если ConstructionTime равно нулю — структуру построить невозможно.

public class BuildingConfig

{

*// ...*

public int ConstructionTime;

}

public class Factory

{

*// ...*

Type = BuildingType.PowerPlant,

ConstructionTime = 8,

*// ...*

Type = BuildingType.Smeltery,

ConstructionTime = 10,

*// ...*

Type = BuildingType.Roboport,

ConstructionTime = 12,

}

Теперь создадим класс Progression, которым мы будем реализовывать любые прогрессии, которые текут во времени, например, строительство.

public class Progression

{

public readonly int Time;

public int Progress { get; private set; }

public bool IsFake {

get { return Time == 0; }

}

public bool IsReady {

get { return IsFake || Progress >= Time; }

}

public bool IsRunning

{

get { return !IsReady && Progress > 0; }

}

public Progression (int time)

{

Time = time;

Progress = 0;

}

public void AddProgress ()

{

if (!IsReady) Progress++;

}

public void Complete ()

{

if (!IsReady) Progress = Time;

}

public void Reset ()

{

Progress = 0;

}

}

Теперь добавим в наши комнаты и модули возможность постройки.

public class Building

{

*// ...*

public readonly Progression Constructible;

*// ...*

public Building (BuildingConfig config)

{

*// ...*

Constructible = new Progression(config.ConstructionTime);

}

## 7. Добавляем ресурсы

**Ресурсы**

В данной версии игры будет присутствовать 12 видов ресурсов. Их все лучше разместить в верхней части экрана, где и сейчас. Список ресурсов в порядке отображения: Марсоходы, Энергия, Минералы, Редкозем, Вода, Пища, Сплавы, Растения, Химикаты, Полимеры, Электроника, Колонисты\*.

Давайте реализуем ресурсы, чтобы игроку пришлось оплачивать свои постройки. Ресурсов будет 12:Марсоходы, Энергия, Минералы, Редкозем, Вода, Пища, Сплавы, Растения, Химикаты, Полимеры, Электроника, Колонисты\*.

public enum ResourceType

{

Energy,

Ore,

Metal

}

Также создадим Банк, где игрок будет хранить и откуда забирать ресурсы.

public class Bank

{

private readonly Dictionary<ResourceType, int> resources = new Dictionary<ResourceType, int>();

public int Get (ResourceType type)

{

return resources.ContainsKey(type) ? resources[type] : 0;

}

public void Change (ResourceType type, int value)

{

var current = Get(type);

if (current + value < 0) {

throw new ArgumentOutOfRangeException("Not enought " + type + " in bank");

}

resources[type] = current + value;

}

}

public class Core

{

*// ...*

public readonly Bank Bank = new Bank();

}

Теперь добавляем цену производства в настройки модулей и строений:

public class BuildingConfig

{

*// ...*

public Dictionary<ResourceType, int> ConstructionCost;

}

public class Factory

{

*// ...*

Type = BuildingType.PowerPlant,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 20 }},

*// ...*

Type = BuildingType.Smeltery,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 20 }},

*// ...*

Type = BuildingType.Roboport,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 20 }},

*// ...*

*// ...*

Type = ModuleType.Generator,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 10 }},

*// ...*

Type = ModuleType.Furnace,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 10 }},

*// ...*

Type = ModuleType.Digger,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 10 }},

*// ...*

Type = ModuleType.Miner,

ConstructionCost = new Dictionary<ResourceType, int>() {{ ResourceType.Metal, 40 }},

*// ...*

}

Теперь добавим команду, которая позволяет платить ресурсы и сразу же попробуем ее в деле (в тестах):

public class Pay : Command

{

public readonly Dictionary<ResourceType, int> Cost;

public Pay (Dictionary<ResourceType, int> cost)

{

Cost = cost;

}

protected override bool Run ()

{

*// Если хотя бы одного ресурса не хватаем - отменяем всю оплату и возвращаем ошибку*

if (Cost.Any(item => Core.Bank.Get(item.Key) < item.Value)) {

return false;

}

*// Если всех хватает - забираем из банка*

foreach (var item in Cost) {

Core.Bank.Change(item.Key, -item.Value);

}

return true;

}

}

## 8. Добавляем цикл производства

Забирать ресурсы, конечно, приятно, но давать значительно приятнее. Давайде запрограммируем возможность запускать производственные цепочки. Каждый модуль сможет скушать определенное количество сырья и потом выдать готовый материал. Снова начинаем с конфигурации:

public class ModuleConfig

{

*// ...*

public int CycleTime; *// сколько времени модуль будет перетравливать сырье*

public Dictionary<ResourceType, int> CycleInput; *// сколько сырья*

public Dictionary<ResourceType, int> CycleOutput; *// какой выход готовой продукции*

}

public class Module

{

*// ...*

public readonly Progression Cycle;

public Module (ModuleConfig config)

{

*// ...*

Cycle = new Progression(config.CycleTime);

}

}

public class Factory

{

*// ...*

{ ModuleType.Generator, new ModuleConfig() {

*// ...*

CycleTime = 12,

CycleInput = null, *// электростанция ничего не требует, только дает*

CycleOutput = new Dictionary<ResourceType, int>() {

{ ResourceType.Energy, 10 }

},

}},

{ ModuleType.Furnace , new ModuleConfig() {

*// ...*

CycleTime = 16,

CycleInput = new Dictionary<ResourceType, int>() {

{ ResourceType.Energy, 6 },

{ ResourceType.Ore, 4 },

},

CycleOutput = new Dictionary<ResourceType, int>() {

{ ResourceType.Metal, 5 }

}

}},

{ ModuleType.Digger , new ModuleConfig() {

*// ...*

CycleTime = 18,

CycleInput = new Dictionary<ResourceType, int>() {

{ ResourceType.Energy, 2 }

},

CycleOutput = new Dictionary<ResourceType, int>() {

{ ResourceType.Ore, 7 }

}

}},

{ ModuleType.Miner , new ModuleConfig() {

*// ...*

CycleTime = 32,

CycleInput = new Dictionary<ResourceType, int>() {

{ ResourceType.Energy, 8 }

},

CycleOutput = new Dictionary<ResourceType, int>() {

{ ResourceType.Ore, 40 }

}

}}

Теперь добавим в каждый ход прогресс по производству:

public class NextTurn : Command

{

protected override bool Run ()

{

new CycleProgress().Execute(Core); *// Добавьте его в начало, это будет важно в тестах*

*// ...*

}

}

public class CycleProgress : Command

{

protected override bool Run ()

{

foreach (var room in Core.Ship.Rooms) {

BuildingProgress(room.Building);

}

return true;

}

private void BuildingProgress (Building building)

{

if (!building.Constructible.IsReady) return;

foreach (var module in building.Modules) {

ModuleProgress(module);

}

}

private void ModuleProgress (Module module)

{

if (!module.Constructible.IsReady || module.Cycle.IsFake) {

return;

}

*// Добавляем прогресс только если модуль уже запущен (ресурсы были заплачены)*

*// Или если мы можем запустить его сейчас (заплатить ресурсы)*

if (module.Cycle.IsRunning || TryStartCycle(module)) {

AddStep(module);

}

}

private void AddStep (Module module)

{

module.Cycle.AddProgress();

*// Если после добавления прогресса работа модуля завершена...*

if (module.Cycle.IsReady) {

*// ... отдаем игроку его ресурсы*

CycleOutput(module);

*// ... и обнуляем прогресс, следующий раз ему придется запускаться сначала*

module.Cycle.Reset();

}

}

private bool TryStartCycle (Module module)

{

if (module.Config.CycleInput == null) {

return true;

}

*// Пытаемся заплатить ресурсы и если удается - модуль запущен*

return new Pay(module.Config.CycleInput).Execute(Core).IsValid;

}

private void CycleOutput (Module module)

{

foreach (var item in module.Config.CycleOutput)

{

*// Отдаем игроку каждый ресурс, который ему был нужен*

Core.Bank.Change(item.Key, item.Value);

}

}

}

Класс получился крупноват, но мы всегда можем его отрефакторить, если сложность будет завысокая. Теперь пишем тест. Он будет довольно длинный, проверять и корректность производства, и незапуск в случае недостачи ресурсов. Также я специально для теста создал отдельные настройки для модуля и строения (вдруг ГД их поменяет и у меня тесты упадут). В идеале все тесты можно было бы поменять на специальные тестовые настройки:

public class Cycle

{

[TestMethod]

public void CheckCycle ()

{

var buildingConfig = new BuildingConfig() {

Type = BuildingType.Smeltery,

ModulesLimit = 1,

AvailableModules = new [] { ModuleType.Furnace }

};

var moduleConfig = new ModuleConfig() {

Type = ModuleType.Furnace,

ConstructionTime = 2,

ConstructionCost = new Dictionary<ResourceType, int>() {

{ ResourceType.Metal, 10 }

},

CycleTime = 4,

CycleInput = new Dictionary<ResourceType, int>() {

{ ResourceType.Ore, 10 },

{ ResourceType.Energy, 5 }

},

CycleOutput = new Dictionary<ResourceType, int>() {

{ ResourceType.Metal, 1 }

}

};

var core = new Core();

core.Bank.Change(ResourceType.Metal, 10);

core.Bank.Change(ResourceType.Ore, 80);

core.Bank.Change(ResourceType.Energy, 10);

var building = new Building(buildingConfig);

core.Ship.GetRoom(0).Building = building;

var module = new Module(moduleConfig);

Assert.IsTrue(

new ModuleConstruct(building, module, 0)

.Execute(core)

.IsValid

);

new NextTurn().Execute(core);

Assert.IsFalse(module.Cycle.IsRunning);

new NextTurn().Execute(core);

Assert.IsTrue(module.Constructible.IsReady);

Assert.IsFalse(module.Cycle.IsRunning);

new NextTurn().Execute(core);

Assert.IsTrue(module.Cycle.IsRunning);

Assert.AreEqual(1, module.Cycle.Progress);

Assert.AreEqual(70, core.Bank.Get(ResourceType.Ore));

Assert.AreEqual(5, core.Bank.Get(ResourceType.Energy));

Assert.AreEqual(0, core.Bank.Get(ResourceType.Metal));

new NextTurnCount(3).Execute(core);

Assert.IsFalse(module.Cycle.IsRunning);

Assert.AreEqual(70, core.Bank.Get(ResourceType.Ore));

Assert.AreEqual(5, core.Bank.Get(ResourceType.Energy));

Assert.AreEqual(1, core.Bank.Get(ResourceType.Metal));

new NextTurn().Execute(core);

Assert.IsTrue(module.Cycle.IsRunning);

Assert.AreEqual(60, core.Bank.Get(ResourceType.Ore));

Assert.AreEqual(0, core.Bank.Get(ResourceType.Energy));

Assert.AreEqual(1, core.Bank.Get(ResourceType.Metal));

new NextTurnCount(3).Execute(core);

Assert.IsFalse(module.Cycle.IsRunning);

Assert.AreEqual(2, core.Bank.Get(ResourceType.Metal));

new NextTurn().Execute(core); *// Cant launch because of Energy leak*

Assert.IsFalse(module.Cycle.IsRunning);

Assert.AreEqual(60, core.Bank.Get(ResourceType.Ore));

Assert.AreEqual(0, core.Bank.Get(ResourceType.Energy));

}

}

## 9. Настройки в JSON

Класс Factory получился раздутым, но если вынести настройки в JSON, то и он будет вполне ничего. Используя Json.NET нам необходимо написать что-то вроде этого:

Настройки в JSON

var files = Directory.GetFiles(path + "/Items/Modules", "\*.json", SearchOption.AllDirectories);

var modules = new List<ModuleConfig>();

foreach (var file in modules) {

var content = File.ReadAllText(file);

modules.Add( JsonConvert.DeserializeObject<ModuleConfig>(content) );

}

{

"Type": "Generator",

"ConstructionTime": 5,

"ConstructionCost": {

"Metal": 10

},

"CycleTime": 12,

"CycleInput": {

"Energy" 6,

"Ore": 4,

},

"CycleOutput": {

"Energy": 10

}

}

Для тех, кто просто любит код — есть [отдельный репозиторий на ГитХаб](https://github.com/theshock/GameLogicArticle)

Кроме этого, если вас интересуют вопросы по разработке SpaceLab — задавайте, отвечу на них в комментариях или в отдельной статье

Скачать для Windows, Linux, Mac бесплатно и без СМС, а так же поддержать нас можно на [странице SpaceLab на GreenLight](http://steamcommunity.com/sharedfiles/filedetails/?id=868755125)

# View

**Игровые действия**

Как это и происходит сейчас, игрок взаимодействует с игровым полем посредством мыши и интерфейса (нажимая на иконки в интерфейсе, либо щелкая левой клавишей мыши).

## 10. Контроллер камеры