

УРОК 2. Разрабатываем простую игру «Galaxy».

Примечание: перед компиляцией демонстрационного проекта, вы должны обратиться к папке References, которая находится в дереве файлов проекта, и там указать ссылку на Finis2D.dll из комплекта поставки.

В уроке 1 мы научились создавать игровые приложения, используя компоненты графического пользовательского интерфейса (GUI), входящие в «**Финист 2D**». Также мы изучили принципы организации приложения, когда в его основе лежит проект, унаследованный от класса **GameProject**, а игровая логика разносится между формами – потомками класса **Form**. Конечно, приложения можно создавать и без участия форм, но их применение делает процесс разработки более структурированным и простым.

Этот урок мы посвятим созданию небольшой, но некогда очень популярной компьютерной игры – «Galaxy». Её сюжет незамысловат: на Землю напала армада пришельцев из космоса, а нам – пилотам космического корабля – предстоит защитить родную планету от их нашествия.

Правила игры

В нижней части экрана находится космический корабль. Мы можем им управлять, двигая его влево или вправо. Корабль может стрелять в направлении верхней части экрана. Напротив него, в верхней части экрана, располагается звёздный флот пришельцев, состоящий из двух типов кораблей. Одни из них, серого цвета, не могут стрелять в космический корабль, управляемый игроком, а другие – красного цвета – время от времени выстреливают сгусток энергии в направлении нижней части экрана.

Условие победы: все летающие тарелки пришельцев должны быть уничтожены.

Условие поражения: в космический корабль попал сгусток энергии из летающей тарелки.

Не очень сложные правила, не так ли? Однако для нашей первой игры их вполне достаточно.

Теперь попробуем представить себе, как будет выглядеть игра при использовании «**Финист 2D**». Нам понадобится проект как центр нашего приложения. Конечно, игру можно реализовать и без форм, но это не так удобно. Потому мы используем следующие формы:

1. Форма главного меню. Здесь будет экранная заставка и две кнопки, дающие нам возможность начать новую игру или выйти из программы.

***Это важно!** Если ваше приложение работает в полноэкранном режиме, то игрок не сможет завершить его работу штатными средствами, если вы не предусмотрели для этой цели никаких визуальных компонентов или команд клавиатуры.*

2. Форма игры. Это своеобразная площадка, где непосредственно будет проходить игровое действие.
3. Форма меню. При помощи этой формы мы сможем поставить игру на паузу или продолжить её, а также выйти в главное меню.

Вот, пожалуй, и всё что нам нужно. Время приступать!

Начать работу над игрой нужно с создания нового игрового проекта. Из главного меню «Visual Studio» выполните команду «File -> New -> Project -> Windows Game» (последний пункт меню находится в разделе «XNA Game Studio»).

В открывшемся проекте нужно найти файл проекта, который называется Game1.cs. Переименуйте его в **GalaxyGame.cs**. Этот класс унаследован от стандартного Microsoft.XNA.Framework.Game. Нужно унаследовать его от **Finist2D.GameProject** (добавьте в раздел uses пространство имён Finist2D). Нам понадобится конструктор, принимающий на вход экземпляр проекта, а также переопределённый метод базового класса *LoadContent*. Прочий программный код, созданный автоматически Visual Studio, необходимо удалить.

Обратите внимание на метод *LoadContent*. В первую очередь настроим хранилище строковых ресурсов *StorageStrings*. Подробно о нём рассказывалось в Уроке 1; дополнительную информацию можно получить в справке.

***Это важно!** Обратите внимание, что игра запускается в полноэкранном режиме. Это не очень удобно во время её разработки и отладки. Закомментируйте строчку `Fullscreen = true` в методе `LoadContent`, чтобы предотвратить запуск приложения в полноэкранном режиме.*

Теперь необходимо создать сами формы и добавить их в проект. Начнём с формы главного меню.

Форма главного меню

В корне проекта создайте папку Forms, где мы будем хранить классы всех форм игры. Добавьте в папку два класса: **FormMainMenu.cs** и **FormMainMenu.Designer.cs**. Унаследуйте класс **FormMainMenu.cs** от **Finist2D.GUI.Form**. Подробно процесс создания

форм описан в Уроке 1, потому мы не будем останавливаться на реализации базового функционала унаследованной формы. Выделим лишь её особенности.



В классе **FormMainMenu.Designer.cs** объявим два поля типа **Button** для кнопок «Новая игра» и «Выход».

Теперь перейдём в код класса **FormMainMenu.cs**. В этом классе нужно объявить конструктор, чтобы тот принимал на вход ссылку на проект. В конструкторе настроим форму точно таким же образом, как в Уроке 1. Обратите внимание, что мы даём имя форме. Это нужно для того, чтобы мы смогли потом отыскать эту форму среди прочих, включенных в проект, и получить на неё ссылку. Мы установим свойство *StartPosition* формы в *StartPosition.FullScreen*, чтобы форма открылась в полноэкранном режиме.

Это важно! Свойство формы *StartPosition* не запускает всё приложение в полноэкранном режиме. Она открывает форму на все доступное пространство, предоставляемое приложением.

Выставим свойство формы *BorderStyle* в *BorderStyles.None*. Это предотвратит отрисовку заголовка формы.

Далее создадим экземпляры кнопок, одна из которых будет отвечать за начало новой игры, а другая - за выход из программы. Настройку свойств этих кнопок будем осуществлять в переопределённом методе базовой формы *LoadContent*.

Каждый графический компонент пользовательского интерфейса, входящий в «Финист 2D», обладает возможностью настройки визуального облика. Это сделано с целью максимально просто и удобно реализовать внешний вид интерфейса игры в зависимости от её направления и темы.

Установка внешнего вида компонента осуществляется путём вызова методов «Load...» с указанием ключа текстуры и ассета, где она находится.

На форме главного меню мы изменим внешний вид самой формы, а также кнопок. Сменить текстуру формы можно методом *LoadBackGroundTexture*. Чтобы изменить внешний вид кнопки требуется вызвать не один, а два метода: *LoadButtonFace* и *LoadButtonFacePressed*. Первый из них изменяет текстуру тела кнопки в обычном состоянии, а второй – когда на кнопку нажали.

Подпишемся на события кнопок *OnClick*. В обработчике этого события для кнопки *bExit* впишем уже знакомый нам код по Уроку 1. Теперь нажатие на кнопку *bExit* завершит работу приложения. В обработчике события для *bNewGame* мы получим ссылку на форму игры путём вызова метода *GetForm* у проекта (в качестве аргумента принимается имя формы) и открываем форму игры методом *Show*. Затем закрываем форму главного меню при помощи метода *Close*. Нужно отметить, что вызов этого метода лишь скрывает форму, но не выгружает её из памяти. Ссылка на неё по-прежнему хранится в списке форм проекта.

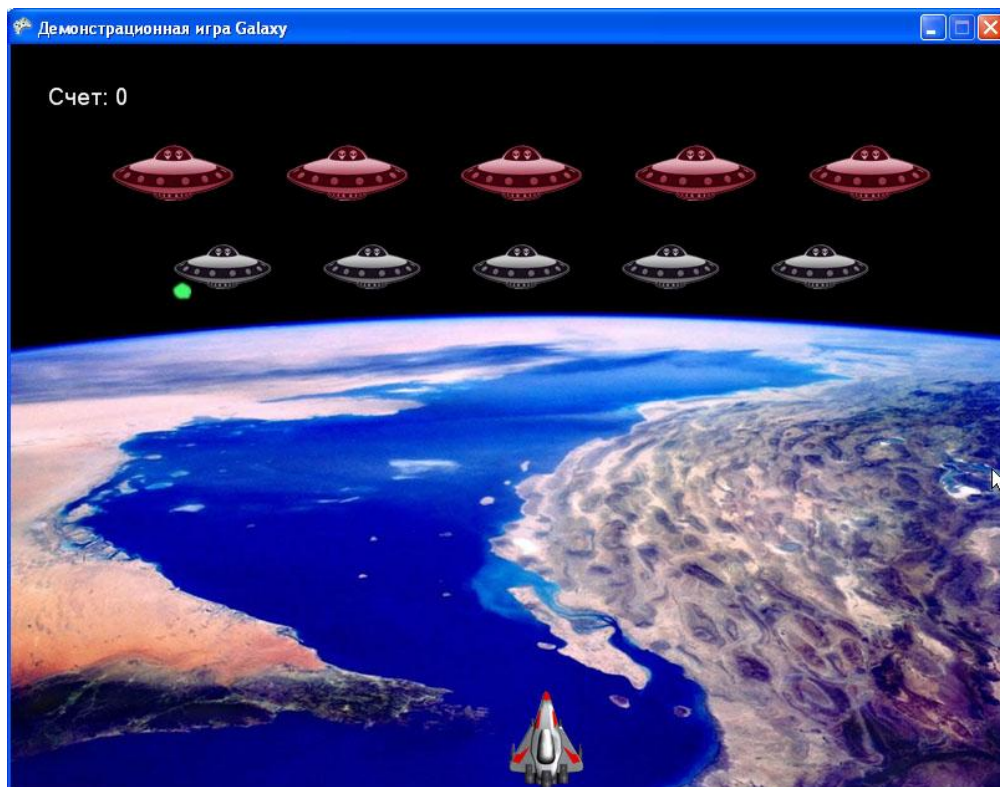
Форма игры

Теперь самое время заняться формой игры. Это площадка, на которой будет происходить основное игровое действие.

В папку *Forms* добавим два класса: **FormGame.cs** и **FormGame.Designer.cs**. Их структура очень похожа на классы **FormMainMenu.cs** и **FormMainMenu.Designer.cs** соответственно.

Объявим два компонента нашей формы в классе **FormGame.Designer.cs**. Речь идёт о текстовой метке, которую предполагается использовать для вывода количества заработанных очков, и о компоненте-рисунке для отображения картинки в случае победы или поражения.

Теперь перейдём в класс **FormGame.cs**. Код конструктора этого класса очень похож на код конструктора класса **FormMainMenu.cs**. Мы точно также определяем имя формы, её размеры, положение на экране (она тоже запускается в полноэкранном режиме), и создаём внутренние компоненты: текстовую метку для отображения очков *lScore* и компонент-рисунок *imgFinal*.



Переопределим базовый метод *LoadContent*, и в его содержимом произведём инициализацию внутренних компонент формы.

Для каждого компонента укажем его расположение на экране. Пусть *IScore* располагается в левом верхнем углу, а *imgFinal* точно посередине экрана. Обратите внимание, что *imgFinal* задаются размеры 400 на 300 пикселей по габаритам картинок, которые мы предполагаем загружать в этот компонент. Компонент класса *ImageBox* имеет свойство *DrawSize*, которое описывает область внутри компонента, куда нужно вывести изображение. Значения этого свойства не могут превышать ширину и длину компонента. Сделаем *imgFinal* невидимым, пока игра не окончится победой или поражением.

Подпишемся на событие формы *OnShow*, которое происходит каждый раз, когда форма показывается на экране методом *Show*. В этот момент мы будем запускать новую игру, вызывая специальный метод *NewGame*, проводящий инициализацию игрового мира. О нём речь пойдёт ниже.

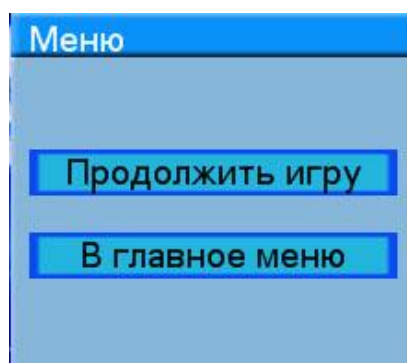
После того, как мы запустим наше приложение и перейдём с формы главного меню на форму игры, мы никаким образом не сможем вернуться обратно, поскольку форма игры не обладает никакими компонентами, обеспечивающими это. Потому мы создадим форму меню (вспомогательного меню), которое будет обеспечивать взаимодействие и переход между формами приложения.

Форма меню

Прежде всего, нужно различать форму главного меню и форму меню. Представьте себе какую-то популярную игру, например, «Fallout». При старте игры вы попадаете в главное меню, где видите полноэкранную заставку, несколько кнопок. Если начнёте новую игру или загрузите сохранённую, а затем нажмёте «Esc», то увидите прямоугольное окно с кнопками, которые позволяют выполнять операции, связанные непосредственно с игровым процессом. В рамках «Galaxy» мы будем придерживаться такого же принципа.

Форма меню вызывается из формы игры. Когда это происходит, игра встаёт на паузу. По закрытии формы меню, игра возобновляется. Внутренние компоненты формы меню включают в себя две кнопки: `bResume` («Продолжить игру») и `bToMainMenu` («В главное меню»). Из названий очевиден функционал, выполняющийся по нажатию на кнопки. При нажатии на первую форма меню закрывается. При нажатии на вторую кнопку, завершается выполнение игры, и происходит переход в главное меню.

Процесс создания формы нам уже хорошо знаком: создаём в папке `Forms` два класса (`FormMenu.cs` и `FormMenu.Designer.cs`), в классе `FormMenu.Designer.cs` определяем два внутренних компонента (кнопки `bResume` и `bToMainMenu`).



В переопределённом методе `LoadContent` произведём настройку кнопок точно таким же образом, что и на форме главного меню. Визуальный стиль кнопок на разных формах будет унифицированным. Нужно отметить, что на форме меню мы скрываем кнопку закрытия формы (системный компонент в заголовке формы).

Это важно! Обратите внимание, как вызывается форма меню из формы игры. Используется вариант метода `Show`, принимающий на вход параметр булевского типа. Если параметр задан в `true`, то форма открывается в модальном (монопольном) режиме. Это означает, что с неё нельзя переключиться на другую форму, пока она не будет закрыта. Тем не менее, с формы, находящейся в модальном режиме, можно открыть другую форму в модальном режиме. По умолчанию формы открываются в немодальном режиме.

Что же касается обработчиков события нажатия на кнопку, то обработчик для `bResume` просто закрывает форму меню, а обработчик `bToMainMenu` закрывает форму игры, прекращая её ход, открывает форму главного меню, а затем закрывает форму меню.

Это важно! Теперь, когда все три формы готовы, нужно вернуться в код класса проекта `GalaxyGame.cs`, перейти к методу `LoadContent` и создать по экземпляру класса каждой формы. Созданные экземпляры добавляем в проект, выполнив метод `AddForm` для каждой из них.

Первой должна открываться форма главного меню, потому найдём её среди перечня форм проекта, выполнив метод `GetForm`, а затем покажем её путём вызова метода `Show`.

Игровые объекты

Наконец-то мы закончили с инфраструктурой игры. У нас есть форма главного меню, которая открывается в момент старта приложения, форма игры, где происходит игровое действие, а также форма меню, которая вызывается из формы игры, и предназначена для установки игры на паузу или её завершения.

Теперь пришло время поговорить о действующих лицах нашей небольшой игры. Каждый из них представлен классом. Ниже приведён их перечень:

1. Космический корабль. Это главный герой, управляемый игроком. Он находится в нижней части экрана. Мы можем управлять им, нажимая стрелки клавиатуры «влево» и «вправо», а также стрелять, нажимая кнопку «пробел».
2. НЛО тип 1. Небольшая летающая тарелка, которая просто висит в космическом пространстве.
3. НЛО тип 2. Этот противник куда серьёзнее, чем Тип 1. Данный вид НЛО умеет обстреливать ту область пространства, где находится управляемый нами космический корабль.
4. Выстрел космического корабля. Поскольку наш корабль умеет стрелять, то нужен объект, обозначающий его выстрел.
5. Выстрел НЛО. Время от времени НЛО тип 2 посылает в пространство плазменные сгустки, пытаясь уничтожить космический корабль землян, то и для его выстрела нужен специальный объект.

Это важно! Для создания различных сущностей в игре, которые требуют графического отображения, рекомендуется использовать класс `Sprite`, находящийся в

пространстве имён *Finist2D.GraphicComponents*. Подробно о его членах можно узнать из справки, а об особенностях применения – из уроков.

Как вы уже, наверное, догадались, первопредком всех классов игровых объектов является именно **Sprite**. Поговорим подробно о каждом из них.

Космический корабль

Корабль – это первый объект, который мы создадим на основе класса *Sprite*. Мы должны определить конструктор, который принимает на вход ссылку на проект игры (точно так же как внутренние компоненты формы).

Наш корабль умеет стрелять. Потому создадим список выпущенных им зарядов. Назовём её *Bullets* (**List<BulletSpaceship>**). Также нам понадобится таймер, чтобы не давать игроку стрелять слишком часто. В конструкторе класса укажем, что время срабатывания таймера равно одной секунде. То есть игрок не сможет выпустить больше одного заряда в секунду.

У корабля есть свойство *Speed*, которое определяет скорость его перемещения при нажатии кнопок клавиатуры «влево» и «вправо».

Перегрузим метод *LoadGraphic* и загрузим текстуру, которая будет обозначать космический корабль.

Создадим метод *Shoot*, который будет вызываться по нажатию кнопки клавиатуры «пробел». В этом методе создадим заряд и расположим его над космическим кораблём. Зададим заряду скорость и укажем, что он может выходить за пределы экрана (по умолчанию, спрайт не может выходить за пределы видимого экрана). С точки зрения игрока картина выглядит следующим образом. Не чаще одного раза в секунду по нажатию кнопки «пробел» космический корабль выстреливает заряд, который летит в верхнюю часть экрана, уходит за видимую область.

Нам потребуется выявлять столкновения между спрайтами, то есть определять, пересеклись ли они. Пересечение можно определять любым способом, но «**Финист 2D**» имеет в своём составе вспомогательный статический класс **CollisionHelper**, находящийся в пространстве имён *Finist2D.Helper*. В данной игре мы будем использовать именно его.

Спрайт обладает событием *OnCollision*, которое вызывают методы **CollisionHelper** для тех спрайтов, у которых произошло столкновение с другими спрайтами.

Создадим событие *Dead*, которое будет означать гибель космического корабля при попадании в него плазменного стутка, выпущенного одной из летающих тарелок пришельцев.

Создадим метод *SetStartParameters*, который устанавливает параметры космического корабля в исходное состояние.

Перегрузим метод *Update*, в котором обработаем поведение каждого выстрела космического корабля. Если он уходит за экран, то удалим этот выстрел.

Выстрелы космического корабля и летающих тарелок пришельцев

По сути, эти классы очень просты и похожи между собой. За тем исключением, что выстрелы космического корабля движутся снизу вверх, а выстрелы вражеского флота – сверху вниз.

Летающие тарелки пришельцев

Структурно эти классы похожи на класс **SpaceShip**. Летающая тарелка типа 1 (класс **UFO1**) не умеет стрелять. Её нужно только отображать на экране. Кроме того, её класс содержит свойство *ScorePerUFO*, определяющее сколько очков будет начислено за уничтожение этой летающей тарелки.

Класс летающей тарелки типа 2 (**UFO2**) более сложный. Он обладает функционалом, реализующим стрельбу. Данный функционал схож с аналогичным у космического корабля, за тем исключением, что для каждой летающей тарелки случайным образом определяется частота стрельбы.

Обратите внимание, что класс **UFO2** является наследником класса **UFO1**. Такой подход позволяет реализовывать функционал, общий для схожих сущностей, на базовом уровне. В качестве упражнения попробуйте сделать для **Spaceship**, **UFO1** и **UFO2** общий базовый класс.

Ход игры

Вернёмся в класс **FormGame.cs**. Создадим новый метод *NewGame*. Это отправная точка начала нашей игры. Именно здесь мы создадим все игровые объекты и установим переменные в исходное состояние.

Это важно! Мы будем использовать специальный объект – экземпляр класса *Renderer* из пространства имён *Finist2D.GameWorlds.RectangleTiles*. Парадигма рендереров, к которым относится данный экземпляр, такова. Для каждого принципа визуализации игрового уровня можно подобрать унифицированный алгоритм, воплотив его в специальном классе, а затем применять этот класс для решения схожих задач.

Класс **Renderer** из пространства имён *Finist2D.GameWorlds.RectangleTiles* позволяет отображать содержимое прямоугольной локации, на которой находятся прямоугольные тайлы произвольных размеров. Также класс поддерживает отображение спрайтов при

помощи специального списка. Отличие элементов карты (тайлов) от спрайтов заключается в том, что элемент карты гораздо более простой класс, предназначенный для отображения статичных (анимированных или нет) сущностей, в то время как класс спрайта реализует динамические сущности со сложной структурой и множеством свойств.

Рендерер обрабатывает сцену. Это специальный объект класса **Scene**. Он несёт в себе информацию об игровой локации. Создадим сцену, и передадим ссылку на неё в конструктор рендерера.

Мы создадим экземпляр класса **Spaceship**, загрузим текстуру, которая будет его обозначать, настроим свойства. Затем выполним метод *CreateUFOs*, который создаст флот пришельцев. Для космического корабля и для каждого НЛО определим обработчик события столкновения с другим объектом и события гибели. В случае уничтожения НЛО начисляются очки (это реализовано в обработчике события *Dead* летающих тарелок).

Нужно обнулить количество заработанных очков, ведь игра может перезапускаться много раз, скрыть *imgFinal*, в который мы договорились выводить картинки в момент победы или поражения, а также выставим флаг *isGameOver* в *false*. Эта переменная используется для определения, когда игра продолжается или окончена вследствие победы или поражения. Затем запускаем игру, выполнив метод *PauseGame* с параметром равным *false*.

Но нам запускать и наслаждаться игрой пока рано, потому что работа ещё не закончена. Давайте дополним класс **FormGame.cs** недостающим кодом.

Реализуем обработку события *Dead* у **Spaceship**. В первую очередь остановим игру, выполнив метод *PauseGame* с параметром *true*. Выставим флаг *isGameOver* в *true*. Очистим все объекты на экране за исключением космического корабля методом *ClearObjects*. И покажем компонент-картинку, куда загрузим изображение разбомбленной планеты. Корабль погиб, Землю некому больше защитить.

Но что если бы нам удалось победить? Мы должны реализовать обработчик события *Dead* у летающих тарелок. В первую очередь начислим количество призовых очков за уничтожение НЛО. Затем уберём сбитую тарелку из перечня игровых объектов, предварительно вызвав её метод *Dispose*, чтобы уничтожить внутренние объекты (в частности, её выстрелы).

Перегрузим метод формы *Update*. В его начале мы должны определить, какие кнопки клавиатуры нажаты в данный момент. Если нажата кнопка «Esc», то нужно показать форму меню в модальном (монопольном) режиме. Когда эта форма показывается, игра ставится на паузу. Следуем дальше по коду метода *Update*. Обновляем количество заработанных очков, выводя новое значение в текстовую метку *IScore*. Обратите внимание на следующую строку. Все дальнейшие действия по обработке движений космического корабля и НЛО не будут выполняться, если игра уже окончена.

Если игра продолжается, то нам нужно обработать ввод с клавиатуры, который управляет движением и стрельбой корабля, а также обновить состояние флота пришельцев.

Также мы должны обработать столкновения между спрайтами. Для этого воспользуемся вспомогательным статическим классом **CollisionHelper**, о котором рассказывалось выше. Его метод *CheckCollision* проверяет столкновение между всеми спрайтами, помещёнными в список, и вызывает событие *OnCollision* у тех из них, где произошло столкновение с другими спрайтами. Входным аргументом в нашем случае является список спрайтов у сцены, где находится космический корабль, весь инопланетный флот и все выстрелы.

И, наконец, можем определить условия победы. Они достигаются, когда не осталось ни одного НЛО и игра по-прежнему продолжается. В таком случае мы выполняем действия, похожие на те, что выполнялись в случае уничтожения космического корабля. Вот только картинку мы выведем другую – вид космонавта на орбите сообщает нам, что Земля спасена и космос открыт для исследования.

Выйти из игры или начать её заново можно нажав «Esc» и перейдя в главное меню.

В этом уроке мы создали несложную игру «Galaxy», получили представление о её организации, принципах создания игровых сущностей, обработки их поведения, определения условий победы и поражения, а также познакомились со сложной организацией визуальных форм и их взаимодействия между собой.