User Documentation

About

JADIC: Just Another Day In Cosmos is a single-player, 2D, space shooting game. It is written in C# using Windows Forms and the .NET `Graphics` library. Gameplay consists of shooting enemy spaceships while avoiding enemy attacks and trying to get as many points as possible. The games difficulty is being raised incrementally.

Installation

To build the game from source, open the solution in Visual Studio and from there build and run it using the `F5` key.

A compiled executable is available in `JADIC/JADIC/release/`.

Controls

Controls are very simple. To move your ship around use the `W`, `A`, `S` and `D` keys. To fire a missile press `Spacebar`. That's it!

Known bugs

- Sometimes movement of the player spaceship behaves like the movement keys are stuck. This is most likely caused by how Windows Forms handles input keys.
- In a very small window between players death and the GAME OVER scree, It is possible to fire a projectile have it stuck there.

Software Documentation

Program structure

The project contains a minimalistic game engine based mainly on the .NET `Graphics` library. The main logic of the game is handled by one or more objects called `Scene`s. `Scene` is responsible and has control over everything that is currently going on in the game.

`Scene`s are grouped in a class called `Game`. The `Game` class handles the correct succession of `Scene`s, for example, when one scene ends, the `Game` class instantiates and runs the next `Scene`.

Game data is grouped in the `World` class. An instance of the `World` class is shared between `Scene`s. The `World` stores, for example, the main `Player` object as well as `Projectile`s and `Enemy` objects. It also contains *drawable* objects like background and overlay.

In the subsequent sections we will describe these elements in a greater detail.

Classes

All source files are listed relatively to the `JADIC/src/` directory in the base of the project.

Game class

Source file: `Game.cs`

Ideologically, the `Game` is a collection of scenes. The `Game` class oversees the initialization of scenes. When one scene ends the class decides on which scene to run next.

The required methods:

- `public void NextFrame()`

  `NextFrame` is called every tick and runs the appropriate scene or loads the next scene.

- `public void HandleKeys(Keys keycode, bool release)`

  Mainly forwards keys to the scene, but can implement game-wide meta behavior.

Scene class

Source file: `Scenes/Scene.cs`

`Scene` is the main means of control and logic in the game. Most behavior of the game should be implemented through subclasses of `Scene`. Input is forwarded to the `Scene`s through the containing `Game` class. The `Scene` is given and acts upon a `World` object.

When a Scene ends or wishes to hand the control over to the following scene, it sets its `isRunning` variable to `false`.

To specify expected behavior, `Scene` will have to define and or modify a set of methods. The listing and expected usage follows.

- `public abstract void Initialize();`

  Initialization of `GameObjects` for the scene.

- `public abstract void Update();`

  Called every tick. Updates the state of the `World` and `GameObjects`.

- `public virtual void HandleKeys(Keys keycode, bool release)`

  Handles the input keys pressed by the player.

The base `Scene` class also provides a number of helper methods and variables.

- `protected List<GameObject> ExtractCollisions()`

  Detects collisions of `GameObject`s in the world and returns the list of them, removing them from the underlying world object. Also sets the `protected bool isPlayerCollided` variable to true.

  > Note that the state of `isPlayerCollided` is not defined before the `ExtraCollisions()` method is called. If `CleanUp()` is called after the `ExtraCollisions()` method, `isPlayerCollided` will be `false`.

- `protected void PerformActions()`

  Runs the `Action()` method of all of `World`s' `GameObject`s.

- `protected void HandleCollisions(List<GameObject> collisions)`

  This method defines the expected behavior for all collided objects. After executing the following actions are performed:

  - One life is subtracted from all collided objects.
  - Score is added to the total from all destroyed objects.
  - All objects that reached 0 lives are removed, the rest is added back to the world.

- `protected void HandlePlayerKeys(Keys keycode, bool release)`

  Handles player movement and shoot keys.

- `protected void CleanUp()`

  Final clean up after every tick. Restores scene to expected state and removes out of bounds objects.

Because `Scene` implements the `IDrawable` interface it defines a `Render(...)` method. The default implementation renders background, players, game objects, particles and overlay. If the user wishes for different behavior the method is open for redefinition and following helper methods are provided:

- `RenderBackground(Size resolution, Graphics graphics_container)`
- `RenderPlayer(Size resolution, Graphics graphics_container)`
- `RenderOverlay(Size resolution, Graphics graphics_container)`
- `RenderGameObjects(Size resolution, Graphics graphics_container)`
- `RenderParticles(Size resolution, Graphics graphics_container)`

The expected `Update()` method should look like the following:

```
public override void Update()
{
    UpdatePlayer();
    UpdateGameObjects();
```

```
    var collisions = ExtractCollisions();
    HandleCollisions(collisions);

    PerformActions();

    // Scene specific logic

    ...

    // End of scene specific logic

    CleanUp();
}
```

JADIC defines its own set of scenes that can be seen in the `Scenes/` directory.

World

Source file: `World.cs`

`World` is the main structure for storing data about the game world. It is acted upon mainly by `Scene` objects. Other objects should generally avoid modifying its contents.

GameObject

Source file: `GameObject.cs`

`GameObject` is the abstract base class for all objects capable of action in the game. Every `GameObject` is given a `Control` object which decides its next position.

- `public virtual void Update()`

  Decides the next state of the object. By default only moves the objects' location as given by the `Control` element.

- `public virtual List<GameObject> Action(World world)`

  The object is able to act upon the world by adding additional game objects based on the worlds' current state.

  > Note that it is not recommended to change the worlds' state within this method and any objects should be returned as output and passed back to the scene.

- `public virtual bool DetectCollision(GameObject other)`

  Decides on how should collision with other objects be handled. By default, a basic collision detection using rectangular hitbox intersection is implemented.

- `public virtual List<Particle> Destroy()`

  Generates a list of visual particles upon objects' destruction.

Examples of the `GameObject` subclasses are given in `Player.cs`, `Enemies/Enemy.cs`, `Projectile.cs` and others.

Control

Source file: `Controls/Control.cs`

The `Control` class is a collection of `ControlElement`s (similarly how `Game` is a collection of `Scene`s). It also oversees the correct succession of a set of `ControlElement`s.

When the `Control` object is finished with all of its `ControlElement`s it sets its `Ended` flag to `true`.

The core method `Point NextPosition(Point currentPosition)` returns the next position for the given current position. It is decided by the currently active `ControlElement`.

ControlElement

Source file: `Controls/Control.cs`

The `ControlElement` class, or rather its subclasses, handle the movement of game objects. The core method of `ControlElement` is `Point NextPosition(Point currentPosition)`.

Examples of `ControlElement`s are `LinearTransition`, `ConstantDisplacement` or `Follow` given in `Controls/Control.cs`.

Final remarks

Making a game was a very good exercise in object oriented design. Because I was not familiar with neither the process nor the C# libraries, I chose a trial and error approach. This proved to be very cumbersome, which was unfortunate, but ultimately unavoidable. However, I think through continuous refactoring of the code I was able to end up with a quite understandable code base.