OOP – Environment System

The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

Problem 1. Overview

You are given a small **OOP project** that models an **environment** and simulates **different object behavior**.

Your first task is to **study the given project** in depth – go through the **classes/interfaces** and try to **get a good idea** of how the environment system works.

Here's a quick briefing on what we've got:

- **Core**
  - **Engine** – the central unit of the system
  - **CollisionHandler** – called to process collisions between objects in the system
  - **ConsoleRenderer** – responsible for rendering graphics on the console
  - **ObjectGenerator** – generates objects for the engine, you will insert your test objects here whenever you want them to appear in the environment
- **Interfaces –** the following interfaces define the behavior (methods) and properties of:
  - **IObjectGenerator** - produces a set of objecets that are to be added to the environment system
  - **ICollidable** – objects that can collide
  - **ICollisionHandler** - defines behavior for managing collisions of **ICollidable** objects
  - **IObjectProducer** – objects that may produce other objects
  - **IRenderable** – objects that may be renderered to the screen
  - **IRenderer** – describes what a renderer should do
  - **IController** – defines a controller (e.g. mouse, keyboard, touch screen) with a set of **events**, will discuss later
- **Models**
  - **Objects**
    - **EnvironmentObject** – base class for all objects in the game, implements ICollidable, IObjectsProducer, IRenderable
    - **MovingObject** – inherits Environment object and also holds Direction of movement
    - **Ground** – a ground object
    - **Snowflake** – a snowflake object
  - **Data.Structures**
    - **QuadTree** – a tree-like data structure used for collision detection optimization
    - **Rectangle** – represents a rectangular entity, used to describe the 2D position and size of objects

Analyze the described classes more thoroughly. Understand how the **EnvironmentObject** and **his descendants** work and how the **engine uses them**.

…and don't forget to run your project!

OOP – Environment System

The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

Problem 2. Disappearing Snowflakes

Now that you've had a look of how the environment system works, it's time we tweak it a little bit. If you haven't already seen (press Ctrl + F5), there is **snow falling** and a **ground**. However, the snow passes through the ground and it shouldn't.

Step 1 – Disappearing Snowflakes

Make so that whenever any snowflake hits the ground, it disappears (it is removed from the environment).

Hint: Taka a close look at the base **EnvironmentObject** class – see what must be changed in the **Snowflake** to achieve this.

Note: You are only allowed to edit the contents of **Models.Objects** namespace.

The **visual result** should be something similar:

OOP – Environment System

The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

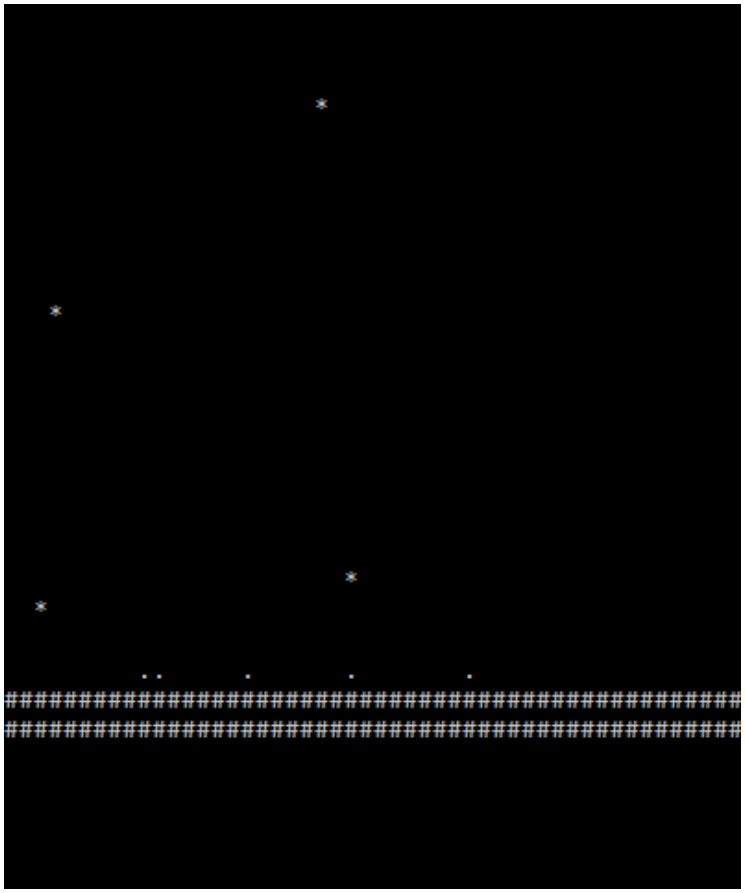
Problem 3. Melting Snowflakes

Good, the snow now hits the ground properly, but shouldn't it also…melt? …and produce snow in the process?

Step 1 – Melting Snowflakes

Whenever a snowflake hits the ground it should not only cease to disappear, but produce **snow**. What is actually snow? – it is something that the **snowflake produces with quantity of 1**. It may look however you wish (e.g. '**.**', to distinguish it from snowflakes).

Note: You are only allowed to edit the contents of **Models.Objects** namespace.

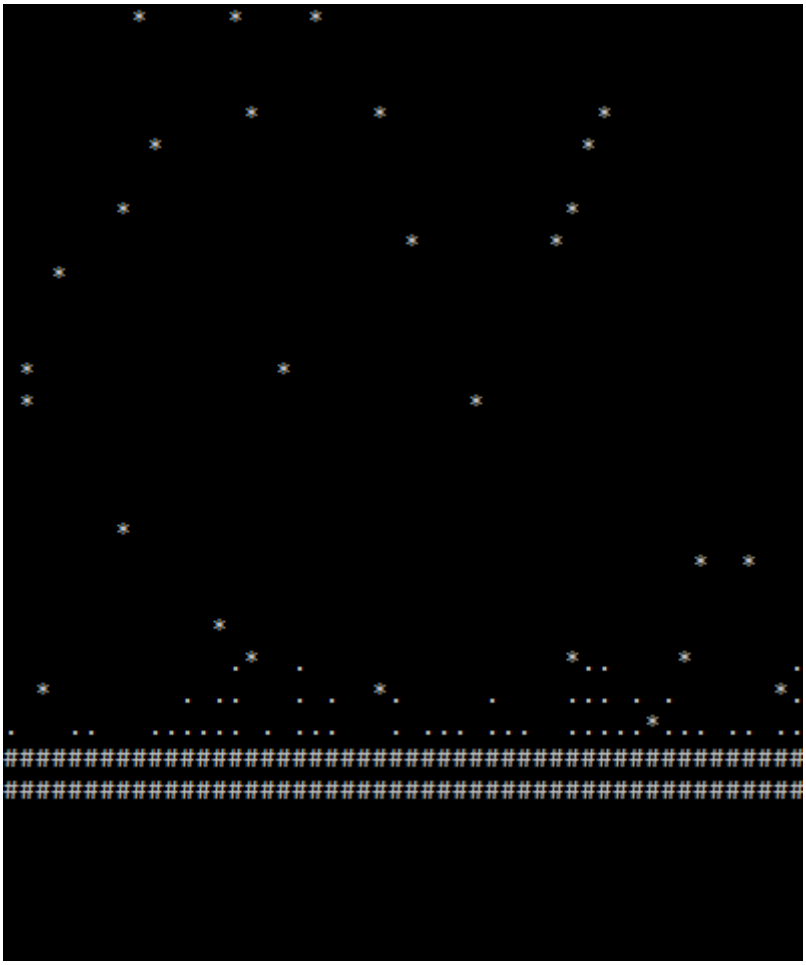


If you look close enough, the little dots on the ground are '.', whereas the snowflakes are '*'. The dots were created when the snowflakes hit the ground.

Step 2 – Stacking Snow

Make so that snow stacks – i.e. if a snowflake falls on snow – it produces more snow.

Notice how at certain places the snow has stacked:



OOP – Environment System

The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

Problem 4. Night Sky

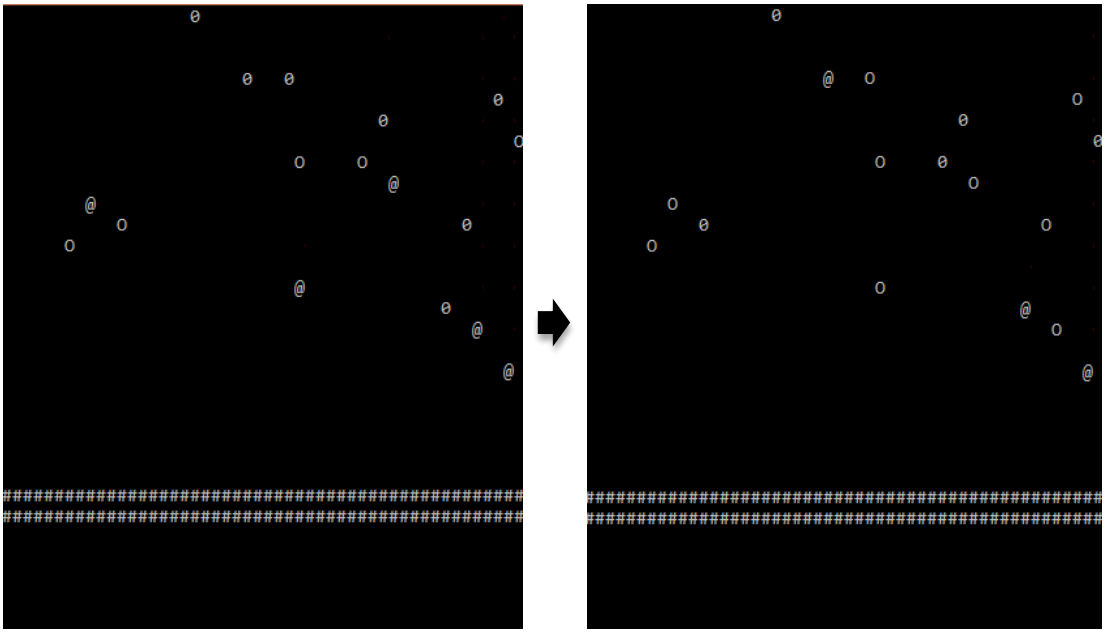
So far so good – snowflakes fall down, melt and stack. Let's **turn off** the snow for now (can be done in the **ObjectGenerator** class) and implement a **night sky**.

Step 1 – Stars

Let's create a **Star** class that will simply model a star in the night sky. The star should stay static and only **change its image** every **10th frame** (i.e. every **10th** engine loop iteration). The image should be one of the following `'o'`, `'@'`, `'0'`, and should be chosen **at random**. The effect should resemble stars flickering.

Test your new star by adding several of it to the **ObjectGenerator**.

The visual result should be similar (stars changing their visual representation):



OOP – Environment System

The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

Problem 5. Falling Stars

Who doesn't love stargazing? What's even more beautiful are **falling stars**. Let's make some!

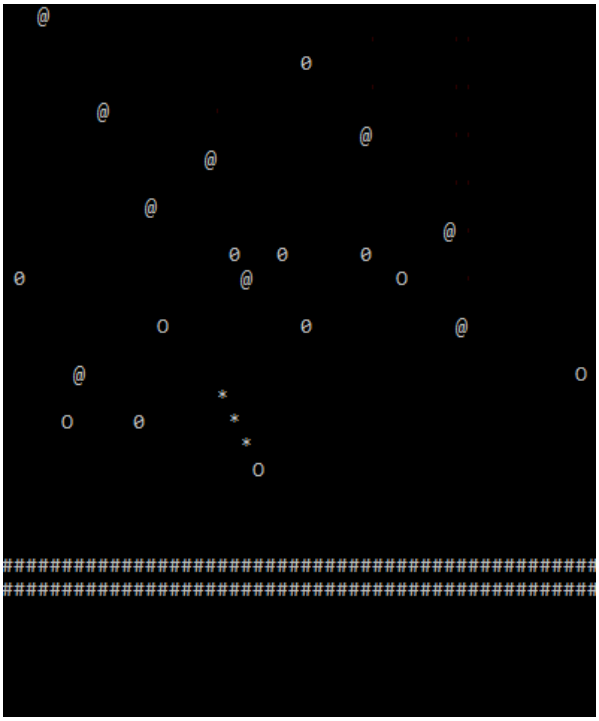
Step 1 – Falling Star

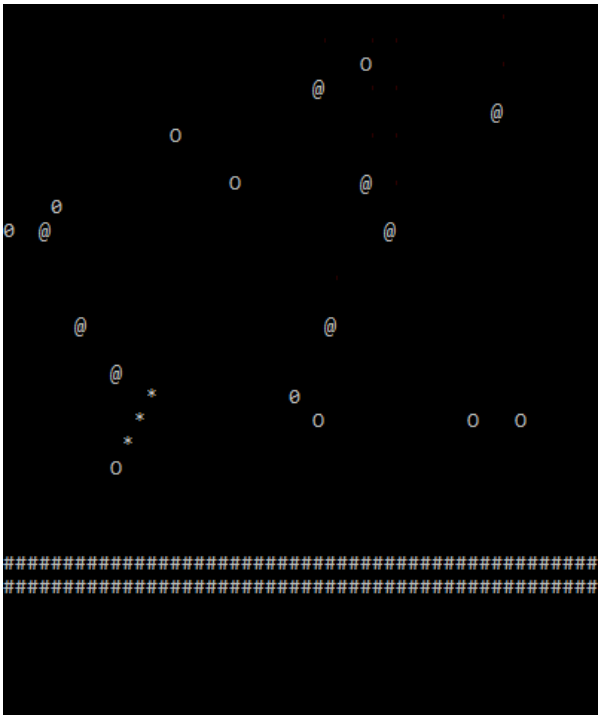
The **falling star** should pretty much fall in a downward direction. Create a class that models such behavior with an image by choice (e.g. **'O'**). The falling star should **disappear** on contact with the **ground** (assume it's the horizon).

Step 2 – Star Trails

We're forgetting something though – falling stars have leave **trails**. Modify your falling star so that it leaves a trail of 3 **'*'** while falling.

Note: The trail left by the star should be relative to its direction.





Step 3 – Make a Wish ☺

OOP – Environment System

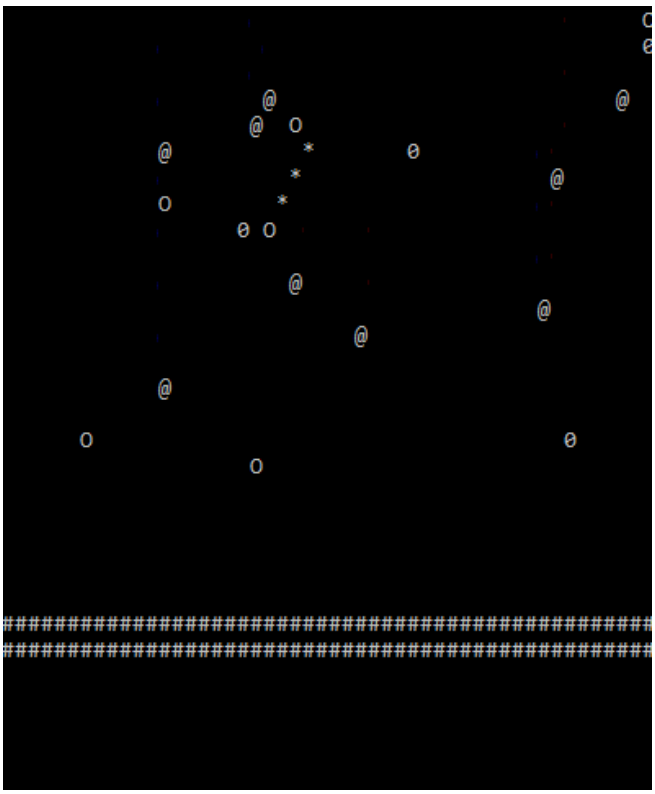
The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

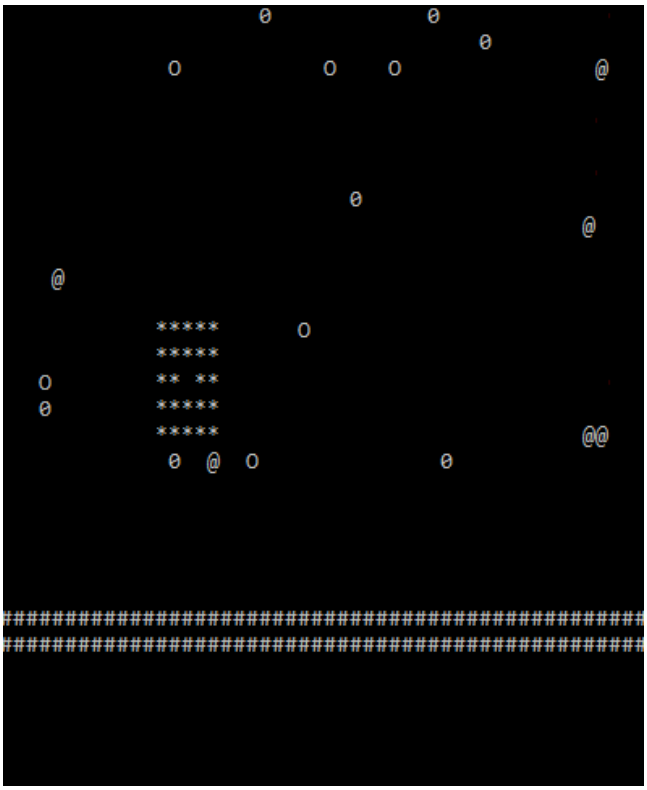
Problem 6. Unstable Stars

Some stars don't have much luck and explode before they fall – and that's why they're most beautiful.

Step 1 – Instability

Such stars have a **lifetime** – i.e. time before they explode. Create an unstable star that has a lifetime of **8** (8 frames of life). Just like the falling star, it should **fall in its direction** and **explode** after its **lifetime expires**. When it explodes, the falling star should produce an explosion with **radius 2** in every direction, except the center. The explosion should persist for **2 frames** (have a lifetime of 2).





Step 2 – Explosion Damage

Whenever any **explosion** from an unstable star occurs, **all stars** (static stars, falling, other unstable stars) caught in the explosion radius should be **destroyed**.

Test this by adding several stars of different types and many unstable stars.

OOP – Environment System

The goal of this lab is to study the **Environment System** and extend its functionality. You are **NOT allowed to directly edit** any of the contents of the **Core** and **Interface** namespaces (only the **Generator** namespace).

Problem 7. Engine Modifications

We've seen that the engine is quite versatile and adding new objects with different behavior is fun. What's missing though?

Step 1 – Validation

Maybe you haven't noticed, but the **Insert()** method in the Engine does not validate whether the object is **null**. However, we cannot simply edit the engine – we must obey the **open/closed principle** (open for extension, closed for modification) and avoid directly editing someone else's code. What we must do is **inherit the Engine class** and **extend** it using the **best OOP practices**.

Extend the engine and perform validation in the **Insert()** method.

Note: Do not repeat any of the base code! Find a way to reuse it.

Step 2 – Pausing

Wouldn't it be fun if you could **pause** and **analyze the environment** at any given time with a **single press of a button**?

This is where the **IController** interface comes in. It defines **event Pause** and **void ProcessInput()**.

What is an event? An **event** is a special type that **keeps methods** and executes them when it (the event) 'happens'.

For example, we can **attach** a **Print()** method to a **MouseClick** event. When the **MouseClick** happens (when the mouse is clicked), it will execute our **Print()** method.

In our case, the **controller** will fire the **Pause** event whenever a certain button is pressed. Then, everything attached to the **Pause** event will be executed. What will we attach to the **Pause** event? – think about it.

- The new engine (the one that extends the old one) should accept an **IController** in its constructor. Create a **KeyboardController** that fires the **Pause** event whenever the **spacebar** is pressed, and pass it to the engine constructor.
- Modify your engine so that it pauses the game whenever the **Pause** event of the passed controller is fired (Hint: attach an event handler that pauses the engine when the event is fired).
- On each environment loop iteration, call the **ProcessInput()** method of the controller to check for any events.
- Again, **avoid repeating code** from the base engine implementation **at all cost**!