Pac Man Man

Kernm. GDV 1

Ruben Bergshoeff

Explanation UML

LevelLoader *<<Singleton>>*

This class is responsible for the loading of a level. A scene will start completely empty with just an instance of this class, prepared with information about the level. How the level should be constructed is based on the pixel colors of a low resolution image, like the image to the left. This way the creation of levels can easily be drawn instead of dragging elements in Unity.

The level is set up by populating the Grid Singleton. More about this can be read below. When all occupants have been assigned, EnemyBase and Player instances are created, at which point the Player will issue an update to the Grid instance, revealing the visible objects in the scene.

Grid *<<Singleton>>*

The Grid class is the container of the current state of the playing field. It contains a Dictionary containing all GridVectors of the game, in which information is stored for each position in the Grid.

The Grid is also responsible for managing visible objects. When update is called with the current position of the Player, objects that have moved out of vision are deactivated and returned to their pool, while new visible positions are made visible by pulling them out of the pool and setting them up according to the assigned types.

GridVector

GridVector instances contain relevant information for a specific position in the Grid. By using an Enum with possible occupants, no objects or references are needed when the specific position is not visible, reducing memory cost. This Enum, OccupantType, is also used to call upon needed objects, allowing for most scripts not needing to know exactly what the contents are.

EnemyManager

The EnemyManager class controls and connects all EnemyBase instances. When EnemyBase instances become active, they register at the EnemyManager, allowing it to update their states. Whenever an EnemyBase is attacked or dies, the EnemyManager will be aware and send out messages to other EnemyBase classes if relevant.

PelletType *<<Enumeration>>*

Because this Enum has such an important role I decided to point it out in this documentation. It has three vallues, around which a lot of the decision making is based: WeaponBase instances have one assigned, determining their damage type, EnemyBase instances have one assigned, determining their weakness, and Pellet instances have one assigned, determining the type of pellet you collect when picking one up.

Design choices

Singleton

I've used the Singleton design pattern at a few points in this project: for the Grid, the EnemyManager, EnemyStates and the AudioManager. First and foremost because each of these objects may never appear more than once at the same time. Secondly because all do not need to exist as long as no-one is calling for them (although they will exist at most points in the game), so lazy initialization is desirable.

Object Pool

While this game is in no way a demanding game, I decided to use the Object Pool design pattern for the sake of practice, efficiency and optimization. All objects that need to be used on a regular basis have been pooled, from purely graphical objects like walls to active objects like enemies. When changing levels these object pools and their contained objects are not destroyed but remain existing in an inactive state.

Factory Method

For the creation of OccupantType related objects I've made use of the Factory Method design pattern. This way the Grid needs not to be aware of the actual objects residing within, but can simply call for the necessary objects. These factories are also the managers of the object pools.

Observer

Quite a few classes in this project use the Observer design pattern to know when to perform actions without having to check the data every frame. For example, the EnemyBase instances register with the EnemyManager and receive a message when another EnemyBase dies. The WeaponBase instance being used by the Player subscribes as an Observer to listen when it has to fire, and it subscribes to the Inventory Dequip message to know when it is being unequipped.

State

For the varying behavior of the EnemyBase instances I've decided to use a Finite State Machine using the State design pattern. As they do not need to contain any data in my application I've decided to make them Singletons as well.

Strategy

The States are responsible for choosing what the target is of EnemyBase instances, but there are multiple ways to get to this target. This is where the pathfinding comes into place. I've emulated the decision making of the original Pac Man, but.I wanted the option open to use other pathfinding methods like A*, so.I decided to use the Strategy design pattern to make this algorithm easily replaced.

Changes along the way

From multidimensional array to dictionary

At first the Grid used a multidimensional array of GridVector instances to mimic the visualization of the play area. While this seemed to make sense at first, it often meant I had to convert separated x and y values to Vector2 values and vice versa to be able to determine movement and location.

The solution was to use a dictionary with Vector2 keys and GridVector instances as the values. This also makes sense from the local perspective, as a GridVector corresponds to a position, the Vector2, and each position should only exist once. An added bonus was that I could easily use the built-in functionality of directions within Unity's Vector2 to acces GridVectors in various directions from a key.

From Grid instances to a Singleton

Initially my Grid class was a regular object which I created each level. This meant a lot of reference management and object creation at each level. Then came the realization that I actually always need but one instance of the Grid, and it is always called upon by others so it could be created by lazy initialization.

Making it a Singleton and adding static methods also made it a lot easier for other classes to acces important information like the occupants of a GridVector.

From collision detection to temporary occupants

I started out building the game and tracking collisions using the method I've used often for 3D projects: Unity colliders. But it soon became pretty bulky and it did not seem like a very efficient thing to do, checking all these colliders all the time.

I decided to first build the LevelLoader and Grid system and look at this problem later, but while creating the GridVectors I realized I could use them to track the position of various objects and avoid collision detection altogether.

Walls do not need any collision detection as I just forbid walking into a GridVector if the occupant is a wall. "Collision detection" for enemies and the player is done by using temporary occupants. Every time an instance enters the area of a GridVector, it adds itself to the list of temporary occupants, and when it leaves it removes itself from said list. Whenever two objects are within a unit distance of each other, they will be subscribed to the same list and thus a collision will take place.

LevelLoader reverted to non-singleton

For most of development LevelLoader was a Singleton. I thought it made sense as a few factories needed information from the LevelLoader, and the Grid needed to be initialized by it. But when I tried to create more levels and load from one to another I ran into quite a few problems.

I read a bit into the Singleton design pattern and realized that it was actually not an ideal candidate for the Singleton design pattern: I wanted various setups of it and it could not be created via lazy initialization.

Another time….

Missed chances for base classes or interfaces

While the whole game is built around the principle of a Grid populated by GridVectors, all instances that are on the Grid are in no way related to each other. I think that it would have been better to define a base class for these objects.

This case continues into moving objects on the Grid. The Player and EnemyBase classes share a lot of properties and do even contain comparable methods, but are again in no way related to each other. Being aware of their similar nature would have spared me quite some work in setting things up.

Builder and Prototype design patterns

One of the things that currently bothers me in the project is the fact that I need object references for the use of prefabs. While this functionality of Unity is at times very useful, it doesn't seem to be the right way to create objects that are often quite similar to each other. Combined with better base classes a Builder and Prototype combination could be used to create the right configuration of objects without the need of prefabs.

Better organization of methods

While I am already pretty proud of the level of organization I have achieved, certainly compared to earlier projects, I feel like some decisions were not the right ones. For example, currently the methods EnableObject and DisableObject are part of the Grid class, but their only purpose is to change data within a GridVector, so it would have made more sense to make these public methods of the GridVector class which could be called upon from the Grid.

Even less public data

Baby steps, baby steps… Like I said before, compared to earlier projects this is already pretty solid, but it could be better. For example; currently objects that enter a GridVector add themselves manually to public lists. This should rather be done with subscription methods to enforce more control over when the list is edited.

What I would have liked to add

Response to dying enemies

While I prepared the system to be able to deal with this, I sadly didn't have time to actually create a response to events like the death of an EnemyBase instance. I wanted each type of enemy to respond differently to the death of a fellow enemy. For some to flee, for others to maybe portray a more aggressive behavior.

Interesting graphics

I hoped to have some time to create interesting visuals on deaths and level loading by using shaders, but sadly I didn't get to actually learning how to work with shaders. Maybe I'll try to add it later…