Pac-Man

Eric Bjuhr
Game Engine Architecture, TDA572
A.A. 2016/17

2017-09-10

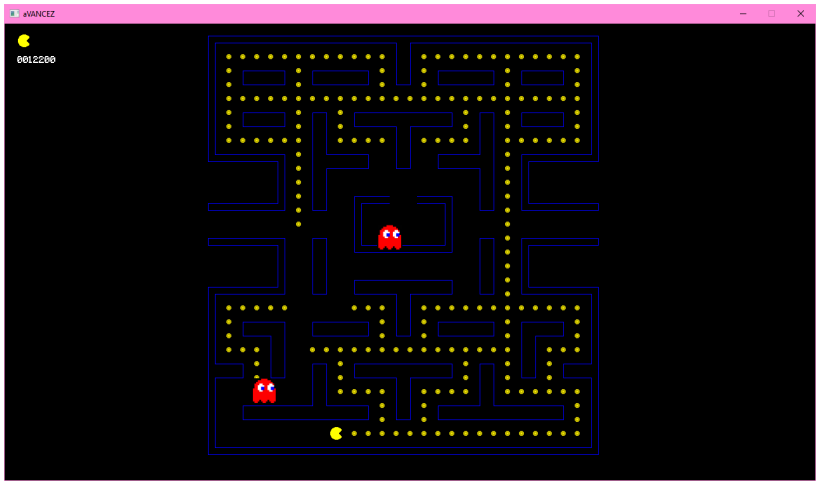

Figure 1: A picture of the game. The map is structured in a grid, where each cell is either a wall, a coin or empty. There are also outdoor cells that help define the outline of the outer walls. Both players and ghosts travel in the grid from cell to cell, with animated movement from one to the next. The ghosts are trying to reach the player, and the player is supposed to pick up coins as Pac-Man walks along the grid.

1 Introduction

Pac-Man is an arcade game in which a circular character is eating dots while trying to avoid being eaten alive by ghosts. By eating said dots, the player will earn points adding to his/her score. Pac-Man game was an international success, and has become one of the most famous titles since it's release in 1980.

This report will detail how this game has been implemented in C++ using modern game engine architecture as taught in the course `TDA572`. All relevant code has been included in an archive, including the libraries it depends on.

2 Specification

The original game consists of Pac-Man, the Maze, the ghosts, the dots and the pickups. However, pickups are not included in the implementation of Pac-Man that is presented here. The goal is to eat as many dots as possible, reaching a higher and higher score every time you do.

- **Pac-Man** is the main character of the game, who is navigating through the maze, eating dots as he passes them by. If a ghost reaches Pac-Man, he dies, losing one life from 3 in total.
- **The maze** defines where Pac-Man and the ghosts can wander around, limiting their navigation through the placement of walls. Throughout the maze, the floor is lined with dots, which Pac-Man can eat. The maze itself consists of tiles which be of various types, such as dots, walls and empty.
- **The ghosts** are enemies, trying to kill Pac-Man. If they reach him, they kill him, consuming one of his lives. There are 4 ghosts who are originally confined to the ghost house in the

middle of the maze, but one of the ghosts has been able to escape into the maze, and is now helping the others do the same by opening the prison door. As a result, two of the ghosts will be chasing Pac-Man from the start of the game, and the other two will join in eventually after Pac-Man has eaten 30 dots and 1/3 of the dots respectively.

- **The dots** cover the floor of the maze, bringing points to Pac-Man if he picks them up and eats them by walking over them. These points are added up to the final score of the player.
- **The end of the game** occurs as soon as Pac-Man has died three times.

3 Overview

This implementation of Pac-Man was made using object oriented C++. As such, the game consists of many different classes, each acting as a blueprint for each instance of that type of object. Additionally, in order to easily grant all classes access to constant values, the values were defined using define directives in a header file. Finally, some types of values are known to only take a low and constant number of states, such as a direction in a grid, and were represented as enumerations in separate header files.

3.1 Programming patterns

Among the classes, some special cases include abstract classes meaning that they have functions without a definition, used to achieve programming patterns. In this game, the most important such patterns are the state pattern, the strategy pattern, the visitor pattern and the game update loop pattern.

- **The state pattern** is used to define variances in behavior depending on the value of enumerations. For instance, depending on which direction you are going in the maze, you will want to add or subtract from your x and y coordinates. Additionally, based on which direction you are currently going, you are only able to change direction to an orthogonal direction.
- **The strategy pattern** is used to define variances in behavior depending on type. For instance, depending on what type of component is being updated, the update function will handle different aspects of the objects. For this reason, that element in the `Component` class follows by inheritance the strategy pattern.
- **The visitor pattern** is used for objects that rely on the context they are being used in. It is sometimes referred to as a reversed listener pattern, in which the context notifies all listeners every time there is an event. In visitor patterns, the roles are reversed, and it is the visitor that requests data from the context. For instance, a player may want to know if a tile on the map is accessible, and can in this game obtain that information by asking a `VisitableMap` object.
- **The game update loop pattern** is used to make sure that the game is continuously updating all of its components. By having a loop based architecture, the underlying models of the game can be updated with continues steps in time, moving the entire simulation forward with the same speed regardless of performance. This also includes verifying the winning and losing

conditions within the game. Additionally, by updating the models and their components, their respective visual representations are also updated and drawn to the screen. In addition to updating the visual representation of the game objects, the game loop also updates the visual representation of the game state as defined by the underlying model.

3.2 Objects

The objects that the game consists of are the player, the ghosts and tiles, which are combined to create a map. These objects are generally only containers of data, including a collection of components which are responsible for utilizing this data to implement the functionality of the object through an `update` function. The only exception is that the objects themselves may have functions that are called from the outside to modify them and to obtain information from them. Additionally, the objects can communicate through messages using a listener pattern. The components come in three different kinds:

- **Behavior components** are responsible for updating the underlying model based on time. For instance, the player behavior component is updating the position of the player, and the enemy behavior component is also deciding how the object will move.
- **Render components** are responsible for drawing the objects. This is done in this implementation of Pac-Man using one or more sprites, or by drawing lines.
- **Collision components** are responsible for checking collision between two objects. This is used in Pac-Man to notify the game that the player has been touched by a ghost.

The game itself is represented by a `Game` class which is created in the `main` function of the program. The Game class contains the update function for the game, which is continuously updated from the main function. The main function is also responsible for updating the game engine, by contacting the update function in the `AvancezLib` class. The game engine will then check for user input and reset the graphics engine so that the game can be re-rendered.

3.3 Path-finding algorithm and ghost behavior

The game uses a path-finding algorithm in order to deterministically define the ghosts' movement behavior. The behaviors are unique to each ghost and are implemented through a strategy pattern. Generally, the ghosts are trying to target specific tiles based on the position of the player, but there are variations in how they do so.

4 Implementation

4.1 Game Update Loop

The game and its game engine is created in the `main` function of `pacman.cpp`, from which they are then continuously called to update the game, creating a pattern known as a game update loop. The game engine listens to keyboard input and window event, toggling a boolean variable when `Q` or `Esc` are hit or the game window is closed. This variable is then read within the `main` function, which

terminates the game loop and closes the game engine, freeing allocated memory. The duration between the previous and the current update loop iteration starts is measured in seconds, and passed to the `update` functions of the game and the game engine. As mentioned in section 3.1, by updating the game with regards to time, it ensures that the entire simulation progresses with the same speed each frame, regardless of performance.

4.2 Object

In this implementation, all entities in the game share a common ancestral class, called `GameObject`. The purpose of this class is to provide the shared functionality that is to be found in any entity in the game among all of the entities in the game, without having to define it multiple times. The class was based on the results from Lab 4, but was extended to suit the need of the game.

Game objects inherit from the class `MovablePosition`, which inherits from the structure `Coordinates`, giving it a two dimensional position and direction. It also provides the public fields `pMovement`, which determines how far the object has moved towards its next tile in percent; `enabled`, which determines whether or not the object is being used and its components should be called; and `active`, which is used to determine whether or not the behavior should progress over time. The latter is important, since sometimes you want to visualize the object without having the behavior active.

4.3 Object Pool

When the game is initialized, it creates objects known as an object pools. This is a type of data structure that lets the application store and pre-allocate required for all present and future game objects (Nystrom, 2014, ch. 19). The objects are initiated in a disabled state, which means that their components with not be updated within the game loop. The object pool provides a method `getFirstAvailable` which provides the first stored object that is still disabled. By using this method, the game can simply pull the object out of the pool when necessary, initialize it and enable it.

The benefit of pre-allocating and creating the objects beforehand is that the game entirely avoids having to allocate large amounts of memory while the game is running. The drawback is that the game is constantly using more memory than it requires at that current moment, because of the disabled objects. When the game is exited, the object pool is destroyed, freeing all of the allocated memory.

The object pools used in Pac-Man are of the same type as the ones used for Lab 4 in the course `TDA572`. This means that they have a templated class, from which generic object pools can be created. The two types of object pools that are created are `tile_pool` containing tiles for the map, and `enemy_pool` containing the ghosts. Both of these object pools are created in the `Game` class.

4.4 Component

Components are a type of class that provides decoupling to game objects, by "outsourcing" certain areas of responsibility from the object itself to its components (Nystrom, 2014, ch. 14). As such, it forms a contract between the object and the component, where the object entrusts the component in finishing a certain task or activity, which is considered functionality of the object. Thus, the object

becomes more dependent on that the component fulfills its end of the contract, but less dependent on how. Additionally, it is very likely that one type of object has functionality that is shared among multiple types of objects. In this case it is pragmatic to have a component which can provide that functionality for all of those objects.

Render Component

Based on the results from Lab 4 in the course, this game also includes render components for various types of objects. For the ghosts and the dots, the visual representation is defined by a single sprite. For the dots, the sprites are static and do not move over time. Since player and the ghosts move, their movement is stored as a percentage of a tile in their respective game object instance. This distance is then added to their position, based on the direction they are facing. Finally, the coordinates of the game objects are in grid space, which means that they need to be scaled and translated into screen space.

For players, the render component also changes the sprite used based on which direction the player is facing. This is to make sure that Pac-Man's visual representation is facing the same direction as his logical representation. In contrast, the ghosts and the dots are always facing out from the screen, and do thus not need to change their visual representation according to the direction they are facing.

```
float renderX = go->x * TILE_SIZE + xOffs,
      renderY = go->y * TILE_SIZE + yOffs;
switch (go->dir) {
        case left:
                sprite2 = sprite_left;
                renderX -= go->pMovement * TILE_SIZE;
                break;
        ... //Other directions
}
```

The walls are visually represented by lines, that go through the center of their tile. One can imagine that the wall tiles consist of posts in the middle of the tile, to which a fence is attached. The visual representation of that fence is then consisting of lines going from one post halfway to another post, which has a line going halfway back.

Collision Component

Similarly to the render component, the collision component is based on the results from Lab 4. The purpose of the collision component is to test for collisions between two objects, one of which it belongs to. One key difference in this implementation of Pac-Man from Lab 4 is that the collision detection is grid based. In addition, the size of every object is a single tile. For this reason, it only checks that the `x` and `y` coordinates are equal between the two components.

The use of the collision component in Pac-Man is limited to checking for collisions between the player and the ghosts. If the player comes in contact with a ghost, he loses his life and a message is sent to the `Game` class which start the game over. After three deaths, the game finally reaches its game over state. The score is transferred over from one game to the next up until game over.

Behavior Component

Precisely like with the collision and the render components, the behavior components had their origins in Lab 4. However, this is the type of component which has seen the most drastic changes. Unlike in the lab, the grid does not update its position over time, so there is no need for a behavior component for that purpose. However, there is a component for future extensions of this implementation, to allow for pick ups to be placed on the map after a certain amount of time or in time intervals.

The player and the ghosts' both have a type of component that is called movement behavior component. This type of component has functions for walking towards the next cell, and walking based on time. In it's update function, it checks if the game object is active, and if so calls the move function, and sets the new direction by calling the `setGameObjectDirection` function. The new direction is obtained through an abstract function called `getNextDirection`. This component is inherited by both the player and the ghosts' behavioral components that implement the `getNextDirection` function.

The player behavior component manages the movement of Pac-Man by implementing the `getNextDirection` function. The movement is as described in section 4.4 modeled with a percentage that Pac-Man has moved in his current direction, which is reset to 0 upon reaching the next cell.

The ghosts have a strategy pattern based behavior component, in which their `getNextDirection` function calls the virtual `hunterStrategy` and `huntedStrategy` functions, which are implemented in inheriting classes. In this implementation, the hunted state never occurs. Nevertheless, the `huntedStrategy` function is still included for future extensions. These two methods are called in the `update` function according to a state pattern.

The four enemy behavior components are named after the ghosts that use them, namely Blinky, Pinky, Inky and Clyde. They all use Dijkstra's path-finding algorithm to find the shortest path to their destination, but chose their destination differently. Blinky, the red ghost, uses the algorithm to find the shortest path to the player's location, typically resulting in him following the player. Pinky, the pink ghost, also the algorithm to find the shortest path to the player's position, but ignores all edges from the player's position except for the one the player is walking, resulting in her appearing in front of the player. Inky, the blue ghost, is looking at Blinky and the player, targeting a tile that is of equal distance that between Blinky and the player, but in the opposite direction of the player. Thus, the result is in Blinky and Inky trying to "box in" the player. Clyde, the orange ghost, is curious but scared of the player, and is following the player like Blinky does, until he reaches a certain proximity of the player, upon which he starts moving to his "home corner".

As a result of Inky not targeting a tile where the player is, it is not guaranteed that the tile is accessible, or even in the grid. Because of this, the path-finding algorithm starts by finding the nearest accessible tile, and targets that one instead.

4.5 State Pattern

In this implementation, the state pattern has been frequently used. The state pattern is characterized as having the behavior depend on a state, an enumeration, and letting the process diverge based on the switch/case of the state. This pattern is used in this implementation in order to for instance define different behaviors depending on the type of tile.

4.6 Map

The map was defined using a two dimensional constant array, in which every cell represents a tile, and the value in each cell represent the type of tile.

```
const char map[30][28] =
{
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1},
{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1},
{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1},
{1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1},
{1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1},
{1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 3, 1, 1, 3, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1},
{2, 2, 2, 2, 2, 1, 0, 1, 1, 1, 1, 1, 3, 1, 1, 3, 1, 1, 1, 1, 1, 0, 1, 2, 2, 2, 2, 2},
{2, 2, 2, 2, 2, 1, 0, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 0, 1, 2, 2, 2, 2, 2},
{2, 2, 2, 2, 2, 1, 0, 1, 1, 3, 4, 4, 4, 5, 5, 4, 4, 4, 3, 1, 1, 0, 1, 2, 2, 2, 2, 2},
{1, 1, 1, 1, 1, 1, 0, 1, 1, 3, 4, 2, 2, 2, 2, 2, 2, 4, 3, 1, 1, 0, 1, 1, 1, 1, 1, 1},
{3, 3, 3, 3, 3, 3, 0, 3, 3, 3, 4, 2, 2, 2, 2, 2, 2, 4, 3, 3, 3, 0, 3, 3, 3, 3, 3, 3},
{1, 1, 1, 1, 1, 1, 0, 1, 1, 3, 4, 2, 2, 2, 2, 2, 2, 4, 3, 1, 1, 0, 1, 1, 1, 1, 1, 1},
{2, 2, 2, 2, 2, 1, 0, 1, 1, 3, 4, 4, 4, 4, 4, 4, 4, 4, 3, 1, 1, 0, 1, 2, 2, 2, 2, 2},
{2, 2, 2, 2, 2, 1, 0, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 0, 1, 2, 2, 2, 2, 2},
{2, 2, 2, 2, 2, 1, 0, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 0, 1, 2, 2, 2, 2, 2},
{1, 1, 1, 1, 1, 1, 0, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1, 0, 1, 1, 1, 1, 1, 1},
{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1},
{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1},
{1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1},
{1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1},
{1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1},
{1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1},
{1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1},
{1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1},
{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
};


Legend:
0: Coin
1: Rounded Wall
2: Outside
3: Empty
4: Straight wall
5: Door
```

By doing so, creating tiles became a trivial task, only requiring looping through the array and creating a tile for each cell with its corresponding type. The pointers to the tiles were then stored in a separate array, and render components were provided to the walls and the dots.

4.7 Visitor Pattern

Visitor pattern is an inheritance relation between an object and a `Visitor`, and the context object and a `Visitable` object. The purpose of the pattern is to allow the visitor object to fetch data and affect the visitable object, which it sometimes is also part of. For this implementation of Pac-Man, the maps are visitable, and the objects frequently visit the map in order to find out if a tile is accessible.

This is used in order to minimize circular dependencies, where one object A is dependent on another object B. By using a visitor pattern, A does not have to know what B is, it only needs to know that it is using a `Visitable` (See figure 2).

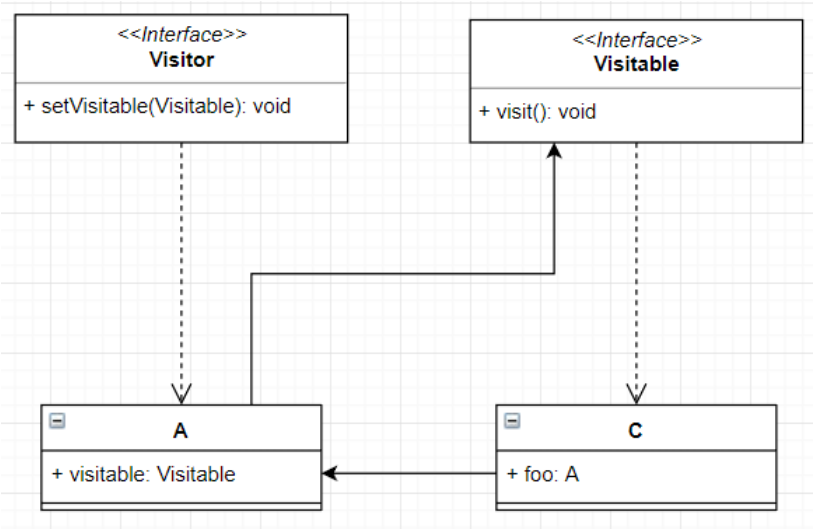

Figure 2: A class diagram showing the relations in a Visitor pattern.

5 Reflection

5.1 Criticism

As described in section 4, this implementation of the game is missing some core features of the original game:

- Primarily, it is lacking the pick up to toggle between states (hunter/hunted), and the associated behavior.
- The visual representation of the ghosts is supposed to have their eyes turn towards the direction they are walking.
- There are no pick ups other than dots in this implementation, whereas in the original game there were several such as the cherries.

5.2 Future Improvement

For the toggle between the two states, only a new type of tile needs to be added to complement the existing ones including dots. In the original game, the toggle was a larger dot found in each respective corner of the grid. A regular dot can be replaced with a larger dot, which can be visited by the player upon arrival using the visitor pattern, resulting in the toggling of the hunter/hunted state. Once there is a toggle ready for the two states, the ghosts need to have their `huntedStrategy` function defined, providing them with the behavior of the new state.

6 Conclusion

The game was made using architecture and programming design patterns taught in class. The architecture includes the use of components, a game update loop, objects, visitor patterns, state patterns and strategy patterns. Although the game is supposed to be a modern implementation of the original game, it is incomplete, and some criticism is brought forth towards parts of the implementation in section 5.1. Nevertheless, the game engine and its implementation provides a good foundation for future work as described in section 5.2.

Bibliography

Nystrom, B. (2014). *Game Programming Patterns*. URL: http://gameprogrammingpatterns.com/index.html (cited on page 4).