Project Assignment

Advanced Programming

2023-2024

1 Introduction

This year's project consists of designing and implementing an interactive game inspired by Pac-Man[1], in C++ and using the SFML[2] graphics library. The main goal of this project is to demonstrate that you are able to create a well-designed architecture, fully utilize advanced C++ features and provide high-quality code that implements the requirements. Of course, it's great if you add creative extra features or fancy graphics and animations, but make sure the basics work well first, and you have a good, extendable codebase to work with. You can play the game here and use this as a reference for your project. Also make sure to read the Gameplay section on Wikipedia.

2 Gameplay

The gameplay requirements are divided into three sections: core functionalities, visuals and aesthetics, and optional extensions for bonus points.

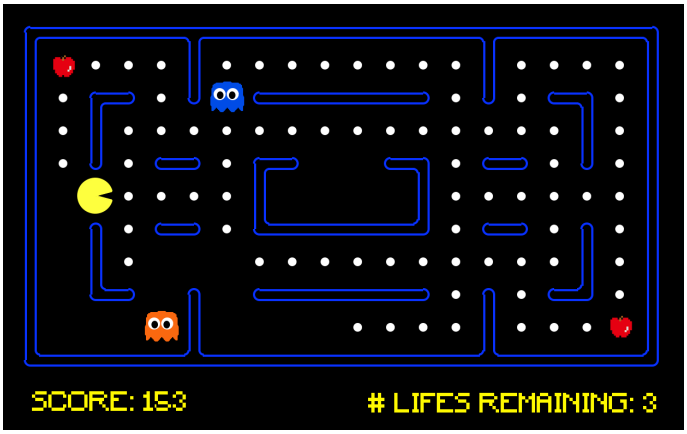

Figure 1: The maze layout you should replicate.

2.1 Core Functionalities

Game startup & Initalization: When launching the game, the user is presented with an interface displaying the scoreboard, containing the top five scores. Below, a "Play" button is displayed. Clicking this button clears the screen and starts the game.

Choose a representation for the maze. How will you represent walls, Pac-Man, the ghosts and other elements in the maze? For the maze, you can replicate the layout shown in figure 1. You may use the existing SFML shapes to replicate the walls. It is not important if the aesthetics do not exactly match those from the figure.

Player Controls & Interactions with Objects: Implement controls for moving Pac-Man through the maze. Ensure Pac-Man responds to player input (e.g., arrow keys). Movement should be

[1] https://en.wikipedia.org/wiki/Pac-Man
[2] https://www.sfml-dev.org/

continuous and not discrete. Implement collision detection to prevent Pac-Man from moving through walls. Pac-Man will continue following in the direction last provided through player input, until it receives new input or until it collides with a wall.

Ghost Movement and AI: Create four ghost objects, spawning them in the center of the maze. Implement their AI as follows: the first two ghosts leave the center immediately and enter "chasing mode," while the last two ghosts do the same after five and ten seconds. In chasing mode, the ghosts should always be locked to moving in a fixed direction (either always up, down, right or left). If a ghost reaches a corner or intersection it will reconsider which direction it will be locked to. In particular, with probability $p = 0.5$, the ghost will lock to a random direction (that is viable). Otherwise, the Ghost chooses the direction that minimizes the Manhattan distance to Pac-Man. For this, look at the viable actions from the set {up, down, left, right} and compute for each viable action what the Manhattan distance to Pac-Man would have been if the ghost had taken one step in that particular direction. Ties between the best actions are broken at random.

Coin Collection and Scoreboard: Display a game score based on the number of coins Pac-Man has eaten (see figure 1). The score decreases over time, and eating coins increases it by a set value. The increased score that a particular point gives should be dependent on the time since the last coin was eaten. E.g.: when the time window between eating two coins is small, then the score should be increased more than if the window is large. Once all the coins have been collected, the level is cleared and the game goes on to the next level. You should also give bonus points for clearing a level and eating fruits and ghosts (see the paragraph on fruits and ghost transformations). The score should be preserved to the next level. Maintain a scoreboard with the top five scores and ensure it's visible at game launch.

Fruits and Ghost Transformation: Place fruits in the upper-left corner and bottom right corners of the maze. When Pac-Man eats a fruit, all ghosts turns into a vulnerable "fear mode" for a limited time window. During this duration ghosts become slower and reverses their direction. In addition, when passing by a corner or intersection the ghosts behave similarly as in chasing mode, but instead of choosing the action that minimizes the Manhattan distance, the ghosts will now try to maximize the distance. While in chasing mode, Pac-Man can eat the ghosts for bonus points. If a ghost has been eaten, it respawns back in the center and immediately starts chasing Pac-Man again.

Clearing Levels and Scaling Difficulty: Once all coins and fruits are eaten, move to the next level, by respawning all the coins and fruits, and moving all ghosts back to the center. In addition, the difficulty of the game should also increase; Ghosts get faster, and the duration of fear mode shortens with each level. Determine specific values for movement speed and fear duration for increasing difficulty.

Multiple Lives and Game Over: Pac-Man starts with three lives. The remaining lives are displayed on the screen, as shown in Figure 1. If a ghost touches Pac-Man, he loses one life, and both Pac-Man and the ghosts go back to where they started. The coins and fruits you've collected stay collected, and your remaining lives don't reset when you move to the next level. When you run out of lives, the game is over, and you go back to the home screen.

2.2 Visuals and Aesthetics

For Pac-Man, fruits, and ghosts (whether in fear or not), use the sprites provided in figure 2. This image is also provided as an attachment to the assignment on Blackboard. Implement animations for Pac-Man and the ghosts: Pac-Man's mouth should open and close as it moves, and the ghosts should have a walking animation. In addition, choose the sprite based on the direction Pac-Man or a Ghost is moving in. The eyes of a Ghost should indicate the current locked-in direction. During fear mode, make the ghosts appear blue. Note that the figure provides more sprites than necessary for the project.

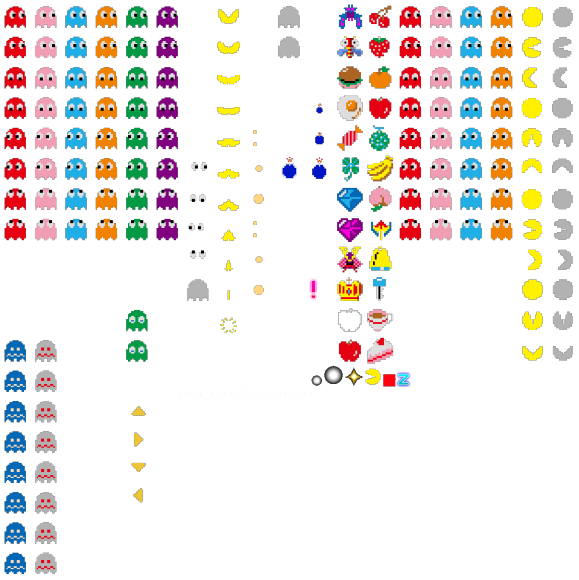

Figure 2: Sprites for the game. Not all figures should be used.

3 Technical Requirements

3.1 Code Design

An important part of this project is creating a flexible design of the game entities and their interactions, as well as the correct use of design patterns. You'll need to design a class structure for the different game entities that facilitates this and keep in mind that your game should be easily extendable. An essential aspect of this design is that there needs to be a clear separation between game logic and representation. Classes that contain game logic should not contain any code related to representation or the other way around. By having this clear separation, you could easily make an alternative representation using a different graphics library, without needing to change any code related to the game logic. To further facilitate this, you'll have to encapsulate the game logic into a standalone (static or dynamic) library using CMake[3] and link your representation code with this library in order to generate the final binary. In theory, you should be able to compile this logic library without having SFML installed. A possible (incomplete) hierarchy could for example look like the following:

You are free in how you design this class structure, but the following key classes need to be present, as well as the use of the design patterns discussed in section 3.1.1:

Game: As part of the game representation, this class is responsible for setting up anything that is not related to the core logic of the game. Examples of this include creating the SFML window, running the main game loop and setting-up the StateManager (see section 3.1.1). Other responsibilities may be delegated to the StateManager or concrete States, such as instantiating a concrete factory and processing user interaction. Note that if the user is currently playing a level, the game representation can translate specific key presses to actions in the World, such as *move left*, *move right*, *move up* or *move down*. But it should not be responsible for how these actions influence the actual game logic.

Stopwatch: This class keeps the difference in time between the current update step (tick) and the previous one (*deltaTime*). Any time your entities move during an update step, the distance should be multiplied with this *deltaTime* value. This is used to ensure that the game logic runs at the same speed, regardless of the speed of the device it is running on. You will also need this value in your representation code to make sure that animations last for a fixed

[3] `https://cmake.org/`

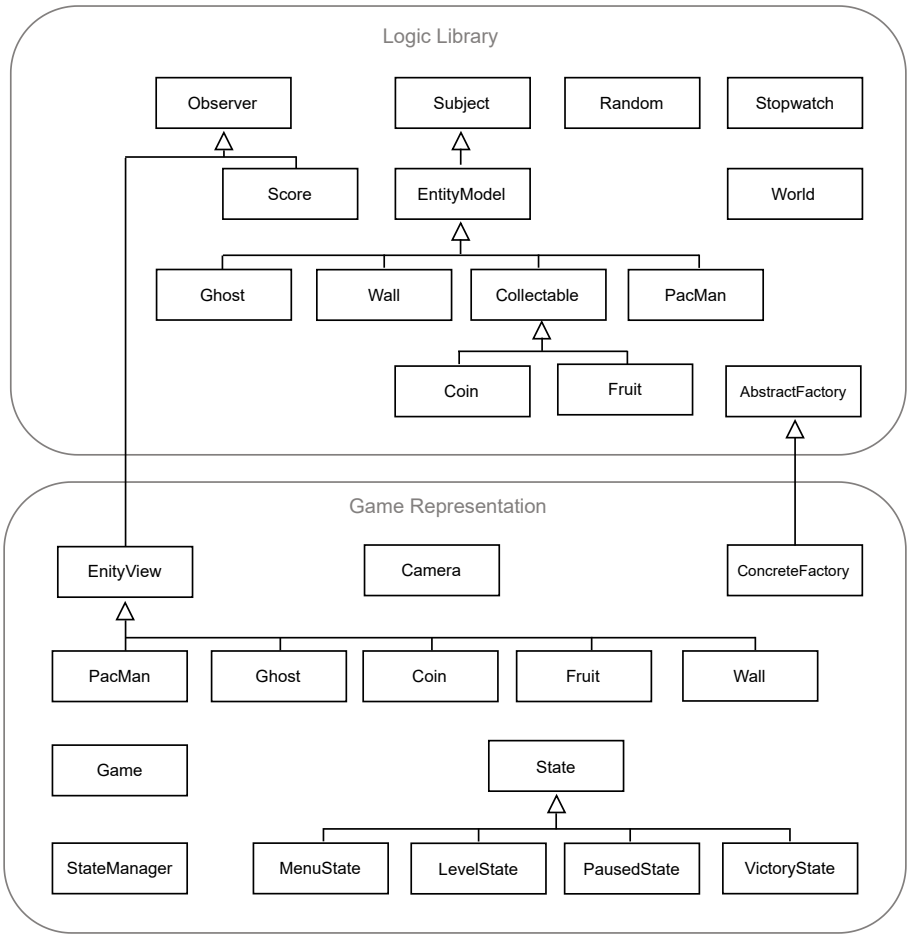

Figure 3: Example class hierarchy with clear separation between logic and representation

amount of time before switching to the next sprite. You're not allowed to use busy waiting to slow down devices that are running too quickly; the frame rate needs to be dynamic. The only exception to this is that you can cap the frame rate at a certain maximum value (for example 60 FPS), to match the maximum refresh rate of your display. To implement this class, you must use C++ functionality, not the SFML Clock class.

World: All entities are stored in the World, which is responsible for orchestrating the overall game logic and the interactions between the entities it contains, such as the creation and destruction of entities and collision detection between them. You can detect these collisions using basic intersecting rectangles, but don't use SFML utilities here, since this class is part of the logic library.

Camera: The Camera class models how the positions of your entities are projected to pixel values on your window. The positions of entities in a World should be modelled using a normalized coordinate system, with the World width and height bounded by [-1, 1]. This ensures that the logic of your game is not dependent on the resolution of the display it's running on. Theoretically, it could even be played through a completely text-based interface. You can also use this projection to compute the proper width and height of your sprites in the representation part of your codebase. This functionality again needs to be implemented manually, without relying on SFML utilities.

Score: This class is responsible for computing the current score, by interacting with the Stopwatch class and through events it receives through the Observer pattern (discussed in the next section). Each event should result in a different modifier applied to the current score. It also

manages the high scores, so they can be displayed on the starting screen, saving them to a file to make sure it is maintained through different runs of your game.

Random: This class is used to generate all the random numbers you'll need throughout the game, mainly in the AI for your Ghosts. Make sure to use proper C++ functionality for this, through a pseudo-random number generator such as a Mersenne Twister[4], instead of the legacy rand/srand from C. This generator should be stored as a data member to ensure you receive a new value from the same generator, instead of creating a new generator for each value, increasing randomness significantly.

3.1.1 Design Patterns

You will need to incorporate the following design patterns in your design:

Model-View-Controller (MVC): This pattern is used in order to clearly model the separation between game-state, graphical representation and the logic of the game. In this pattern, the World can be seen as an *Entity Controller* that manages the interactions between entities, or you could have multiple controllers in your design to further delegate responsibilities. Your *Model* is then responsible for holding any data related to a certain entity and to provide some methods that manipulate this data, while the corresponding *View* is responsible for what this entity looks like and for drawing this to the screen.

Observer: You will need to use the Observer pattern for two purposes. Firstly, for updating the *View* when the *Model* state changes. By attaching the *View* observers to the *Model* subjects directly when they are created in your concrete factory, you can separate the logic from the SFML representation completely transparently. A *View* can then receive an event for every update step (tick) to make sure the *Model* is drawn on the window. To make this easier, you could hold a (smart) pointer to your window object in your *View*. It can also receive events when Pac-Man moves in a different direction or when it dies, such that the correct animation can be displayed. These same generic events are at the same time used by the Score class (also an Observer) for updating the current score when a coin or fruit is collected, when Pac-Man dies, or it eats a transformed Ghost.

Abstract Factory: This pattern is used to provide an easy interface which the World can use to create new Entities, without it needing to be aware of how to create instances of SFML-specific *View* classes. The logic library defines a simple abstract factory interface, which is adhered to by a concrete implementation in the representation code. Finally, the Game class provides a pointer to this concrete factory to the World, which can then use it to produce Entities that already have the correct *View* attached.

Singleton: The Stopwatch and Random helper classes needs to be implemented using the Singleton pattern. This ensures that only one instance will be present during the execution of your program, and they are easily accessible to all classes that will need to use them. You could additionally opt to encapsulate your *sf::RenderWindow* in a singleton instead of storing a reference or pointer to it as member in the *View* classes.

State: This pattern is used to facilitate the transition between the starting menu, the level with gameplay, pausing the game, seeing a victory screen when you've won or lost, and continuing to the next level. These can all be modelled as different *States* in a finite state machine (FSM). A *StateManager* (sometimes called the *Context*) is used to orchestrate the registration and execution of different *States*.

In particular, the *StateManager* will start with an initial state of *MenuState*, which will register a transition a *LevelState* when a key is pressed. This *LevelState* will transition to the *PausedState* when the escape key is pressed, or to a *VictoryState* when either all coins are collected or all lives are lost. In the *PausedState* you can either go back to the *MenuState* or continue your *LevelState* where you left off. You can also split up the *VictoryState* by creating

[4] `https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution`

an additional *GameOverState*. From there, a transition is made to a different *LevelState* when the next level is loaded, or back to the *MenuState* if the game was lost.

In order to support returning to your existing *LevelState* from a *PausedState*, your *StateManager* should be implemented using a State Stack. States can then either add new states on the stack, be popped from the stack or both. When in the *PausedState*, you should have a stack containing the *MenuState*, *LevelState*, and *PausedState*. By simply popping the *PausedState* from the stack, your *LevelState* will continue.

It should be the responsibilities of the individual states to decide when to switch to what new state, not the *StateManager*, which should theoretically not be aware which concrete state is currently running. When a new state is pushed onto the stack, it should always be a new object. Don't keep a list of already instantiated *State* objects to transition to. This also means that a new World instance should be created as part of *LevelState* when transitioning to the next level.

3.2 Code Quality

Below you can find a list of things that need to be present in your code to improve its quality:

- Use **namespaces** to clearly divide modular sections of your code.
- Include **exception handling** to catch and deal with possible errors, such as the absence of required files.
- Proper use of the **static**, **const** and **override** keywords where they can be applied.
- Make sure to avoid memory leaks by explicitly creating **virtual destructors** where necessary.
- Always explicitly initialize **primitive types**. (Hint: check these last two with *valgrind*.)
- Avoid unnecessarily copying objects where they can be passed as a reference or pointer.
- Refrain from relying on **dynamic casts**! This usually means your design is lacking proper polymorphism.
- Avoid duplicate code, solve this by using better **polymorphism** or **templates**.
- Use **clang-format** with *this configuration* to format your code.
- Write proper **code comments** and **API documentation**.
- Use of **smart pointers** throughout the whole project is obligated. This is used to test your insight on where to use unique, shared or weak pointers, depending on the type of ownership. No raw pointers are allowed, except in certain design patterns where the use of smart pointers is prohibitive. But if you need them, please discuss this with us and provide your reasoning first. Passing objects by reference is also perfectly fine, this does not necessarily always need to happen through the use of pointers.
- Keep the design principles in mind. Aim for low coupling and high cohesion in your code.

4 Project Extensions

As discussed later in section 5.2¸ it will be possible to earn up to 10% bonus points by adding extra features to extend your game. Here are some ideas you could consider, or you can try to come up some creative ideas of your own!

- Sounds and Music: Add sounds for events like death and background music to make the game more enjoyable.
- Smarter Ghosts: Make the ghosts smarter using better AI, by using a different search algorithm, such as a breath first search (BFS).

- Random Maps: Explore procedural map generation using online resources for new and unique challenges.

In additional to functional extensions, you can also receive bonus points for technical extensions. Some suggestions include the use of:

- Multi-threading: perform more complex computations on multiple threads to make your game run faster. Potentials for this could be your collision detections, more complex Ghost AI with pathfinding, or splitting your logic updates and rendering steps into different threads. But be careful that creating and destroying threads also causes overhead, so it's not always worth it to start a new thread for simple computations.

- Generic programming: there are quite a few places where you could apply templates to make the overall design of your project even more generic. For example, there's a fancy implementation of the Observer pattern where, through the use of templates instead of inheritance, you can retain the type of the concrete subject you're observing. Or you could simplify the interface of your Abstract Factory by using a single create method that is templatized on the type of Entity you want to create.

- Design patterns: include the correct use of additional design patterns to elegantly solve problems you encounter. The Command pattern could for example come in handy to resolve what needs to happen when colliding with different types of Entities. Or you could use a Visitor when looping over a list of generic Entities in your World to resolve them to their concrete type.

5 Practical Information

5.1 Additional Resources

Below you can find some useful extra resources to help you get started on your project:

- The *SFML Game Development* book: highly recommended for getting started on working with SFML and some general game concepts, such as how a main game loop works. There's also *SFML Game Development By Example*, which is a similar book that explains the same concepts by guiding you through the process of creating snake clone. The pdfs for both can easily be found online, but let me know if you have trouble finding it.

- *Head First Design Patterns* book: contains an explanation for all the design patterns mentioned in this assignment. Again, the pdf can easily be found online, but there are also more than enough other free resources available that explain these patterns well.

- *The C++ Programming Language* book: contains everything you may need to know about how C++ works to complete this assignment and much, much more.

5.2 Grading

The grading of your project will consist of five separate criteria:

- **40%:** Core game requirements: you have a basic (working) implementation that implements all the gameplay elements.

- **40%:** Good design and code quality: you have properly implemented the design patterns and you make good use of polymorphism in your well-designed class hierarchy.

- **10%:** Project defence that will be organized during the examination period: 3 minutes of gameplay demonstration and 7 minutes of discussion on design choices and implementation details.

- **10%:** Documentation: report and comments. In your report of around two (A4) pages, please include an overview of your design choices and use this as an opportunity to convince me to give you good grades for your project. If you are able to explain your choices well, this can also result in higher grades for the other criteria. Avoid simply repeating what is already explained in this assignment.
- **10%:** Bonus points: you can earn these by implementing creative extra gameplay mechanics, making the game look and feel extra fancy or making correct use of additional design patterns (see section 4 for some potential ideas). These extra points are simply added to the grade of your project, so your total becomes $\min(40 + 40 + 10 + 10 + 10, 100)\%$ in case all parts of your project are perfect. If you added any of these extra features to your project, make sure to also document them thoroughly in your report.

5.3 Submitting

Take the following things into account before submitting your code:

- Your code should **compile and run successfully on the reference platform at the university computer labs**:

  Ubuntu: 22.04, **SFML:** 2.5.1, **CMake:** 3.22.1, **G++**: 11.3.0, **Clang**: 14.0.0

- This project has to be completed **individually**, plagiarism will not be tolerated. You can of course freely discuss design decisions or implementation problems with other students.

- Accept the GitHub Classroom assignment and push your project to the associated repository. Make sure to include your name and student number in the *README.md* file. Frequently commit code increments and set up CI to automatically test whether the project still compiles successfully. The final commit of your project must show a **successful build on a CI platform that is linked to your GitHub repository**. We recommend you use CircleCI[5] for this, since their free plan allows for considerably more than enough weekly builds for this project and it's easy to set up. You are also allowed to use an alternative CI platform, but make sure that we can see the build results for each commit on GitHub and the build configuration files (e.g. *.circleci/config.yml*).

- If you have any questions, don't hesitate to contact us (Thomas.Ave@uantwerpen.be or Fabian.Denoodt@uantwerpen.be).

- The project deadline will be in **January 2024**. The exact date will be announced later on Blackboard.

- The final project needs to be submitted on **Blackboard** and on **GitHub**.

Good Luck!

[5] `https://circleci.com/`