Report Project Advanced Programming

Name: Tibo Verreycken

Student Number: S0222730

Content

Class Hierarchy 1

Multiple keys 2

Continuous Collision Detection 2

Collision Handling 3

Intersections 3

World class structure 3

Movement direction 3

UI Objects 4

RenderWindowSingleton 5

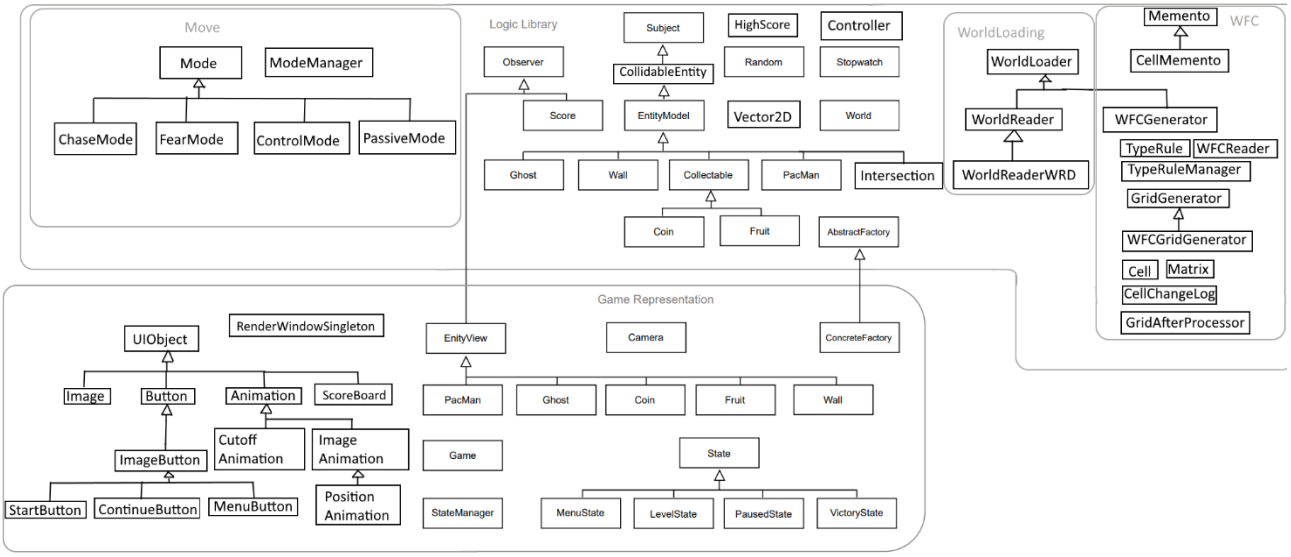
WorldLoader 5

Procedural Generation (Wave Function Collapse) 5

Bonus 6

Class Hierarchy

This Report will be about the project for Advanced Programming (Pacman). Here I will mainly discuss things that are not clearly specified in the assignment. The assignment used a visualization of the class Hierarchy (Figure 3). Below is an augmented version of this visualization which also contains the extra classes added to the project with their respective hierarchy.



Now we will discuss the project in more detail.

Multiple keys

My implementation of Pacman Supports pressing Multiple Keys at the same time.

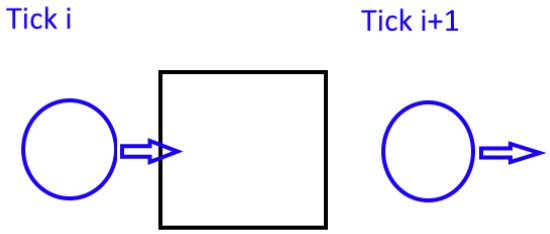
The reason I wanted to this this are the following:

1. Using continuous movement there is a possibility that we pass an intersection where we wanted to go to another direction. We can make Pacman smaller/reduce the speed to avoid this problem, but I wanted to leave those things unchanged. Assume a User is pressing 2 buttons at once Up and Left. On the Left side is a wall, so it will just go up, till there is an intersection, in that case it will go left and probably hit a new wall at the top.
2. I also think supporting multiple buttons improves the play experience.

Continuous Collision Detection

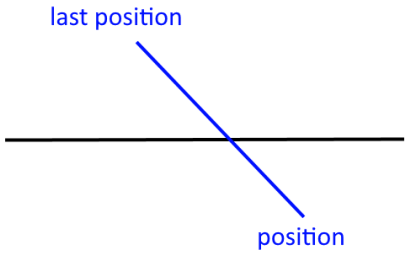
For the project we are supposed to use continuous movement.

To prevent the tunneling problem for low FPS (entity is before a wall and after a tick it moved passed the entire wall)



I used a collision detection method that is called continuous collision detection.

It stores both its position from just before and after the move, later when we check collision we just need to check if the line between position and last position intersects a border line of another object.

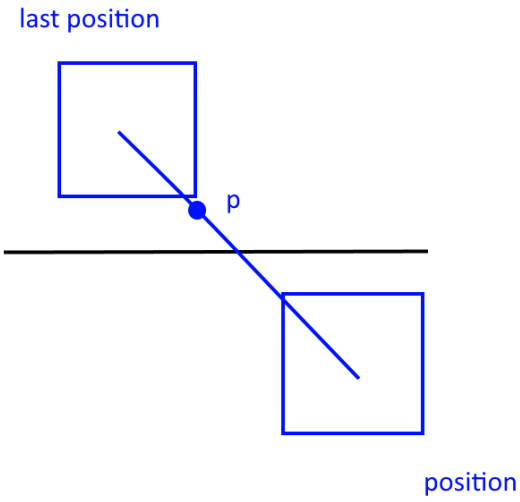


Even when we would pass the entire other Entity in 1 tick, we would still have an intersection (even multiple).

To determine which line we intersected first, we will calculate a value ‘best’. If a collision is present ‘best’ will be between [0, 1] and the point where we crossed the other line will be a linear interpolation between the last position and the current position (with respectively ‘best’ and ‘(1-best)’ as scalars).

So, intersection point p = last_position*best+position*(1-best). Determining the point of intersecting can be used later to handle the collision.

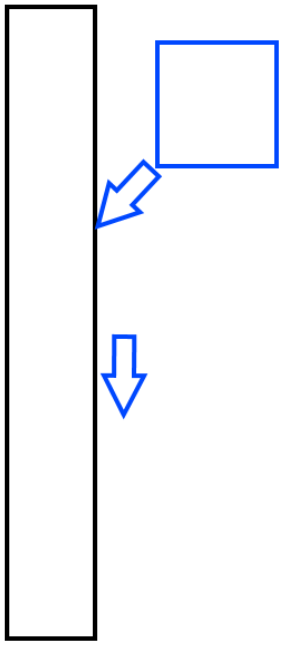
One problem we did not yet discuss is the fact that we work with squares (square collision) and not just points.



We will do the same thing, but we use the center of the square for the position, and we will change the calculations a bit so the collision will already occur from the moment that the closest line of the square crossed the black line. In the example of the image on the right we will just need to check the downside of the blue square, and if we would change ‘position’ with ‘last position’, we would need to only take the upside into account. The reason we use the center for position is so we can make the code more regular, because the upside and downside edge have both blue square height/2. This means we just need to decide whether to add or remove the height/2 from the center. When check for collision we also return the point p (point at which collision occurred) and the Axis at which collision occurred (X-Axis/Y-Axis). The reason we do this is for our collision handling.
Collision detection between 2 Entities Occurs in the class CollidableEntity. The reason this is an extra class in the EntityModel hierarchy is to separate it from the EntityModel class (to respect the Single Responsiblity Principle), but it is kept inside the same hierarchy to take advantage of the high cohesion between the datamembers and the collision function.

Collision Handling

Using multiple keys, we made our collision handling a bit more complex.



Assume We have an entity going both Left and Down (see image on the right). We want to go in both directions till we hit a wall on the left side. In that case we do not want our square to freeze in place, but to continue down. On the other hand when our square hits 2 Walls (1 left and 1 down) we don't want it to get through one of these walls. That is why our collision handling is in 2 phases.

Phase 1:

If collision occurs it will still go to the remaining directions.

If this is the only direction the remaining direction will be an empty (0, 0) vector. It will travel the remaining distance for the length that it wasn't able to do because of the collision.

Phase 2:

Check collision again but if collision occurs, we will not go into the remaining direction anymore, but go to the position just before collision. This ensures that we cannot glitch through corners by using the collision handling of one of the 2 walls.

To set the Entity to the position just before collision and to calculate the remaining direction, we just use the values calculated in the collision detection.

Intersections

We also have an extra entity: 'Intersection'. This Entity will make it easy to detect an intersection and to make sure Ghosts check if they want to change their direction, and to make sure that Pacman can move Corners in case of Low Frame Rate (preventing the collision handling its remaining distance to skip the Intersection).

World class structure

The World class is used to manage interactions between entities.

The main interaction between entities is collision. Depending on what we hit during collision we need to do something else. To make this regular we have a function 'checkCollision'. We give it a function pointer to the function we want to call when collision occurs, this function pointer has a void function with as parameters 2 Entity Model pointers of the 2 Entities that collided.

We will first check collision with not passable entities in my project: (Walls)

After we will check if we hit other things and handle it different depending on whether the entity is Pacman. At the end we check collision with Intersections and make sure the change in direction/ movement occurs.

Movement direction

To determine the direction the Entities, go to I used the Strategy Design Pattern.

We will refer to a strategy as a Mode. Each Entity Model has its own Mode managed by a Mode Manager. The Mode will be used to calculate the direction. Based on the Mode the calculations can be different. In the current Project we have 4 Modes: Chase Mode, Fear Mode, Control Mode, Passive Mode.

Chase Mode:

- Will go 50% in a random direction.
- Will go 50% minimize the Manhattan distance with a given entity.

Fear Mode:

- Will go 50% in a random direction.
- Will go 50% maximize the Manhattan distance with a given entity.

Passive Mode:

- Will never move, its direction will stay (0, 0), useful for not moving entities like walls, coins, …

Control Mode:

- Will make his direction depend on the direction of the Controller used to receive which keys are pressed.

I chose to use this Design Pattern to determine the direction for the following reasons:

1. The Direction is independent of the Object Entity itself. This makes it possible to easily change some of the movement mechanics of an Entity. For example, we would be able to make moving Fruits or Control Ghosts instead of Pacman. This makes the code easy to change, will not need this for Pacman itself, but if we would ever want to change the game, the movement can be changed easy.

2. Another reason is expandability, it is easy to add new derived classed of the Mode to create entire new Modes to the determine the direction.

UI Objects

When making small animations, buttons, text I concluded that some of the UI components could be better organized. That's why I made a class 'UIObject' with as derived classed UI objects we want to render. UI objects are things that has nothing to do with the game itself, like buttons, (image) animations (EntityView animations are not among these), images. UI Objects have a '.render() const' function to render the entity.

Image:

- Contains a sprite it wants to render.

Button:

- Represents a button we can check if it was clicked.

ImageButton:

- Button that has an Image.

Animation:

- Abstract class for animations

CutOffAnimation:

- Animation where only a part of the image is show, the part increases and decreases as part of the animation.
- Example: The text 'Victory' shown in the VictoryState is seemingly eaten by Pacman but is in reality just a part of the text is shown.

ImageAnimation:

- Changes every animation delay the image it shows.
- Example: The text 'Game Over' has an animated ghost in the 'O'.

PositionAnimation:

- Moves the image to a given position during the animation.
- Example: The Pacman shown in the VictoryState moves from left to right and back, while eating the text 'Victory'.

Scoreboard:

- Renders the top 5 high scores, using data from the HighScore Singleton

RenderWindowSingleton

RenderWindowSingleton is a class in the View part, that is used as a wrapper for the SFML Window.

This can also buffer its Drawables it needs to draw. In case an Entity Passes to its EntityView that it moved multiple times, we just want to draw the last position (Collision handling can cause the Entity being moved). So, this Object will have a buffer map with as keys a ptr to the Entity. If a new drawing for a given entity comes, it will override the old drawing.

WorldLoader

To create a World, we will use a WorldLoader. This is the Base class of the Abstract Factory for creating Worlds. We will later see how we can create worlds using Procedural Generation. If we want to read world data from a file and use it, that is possible. Reading a World File will be done by a Derived class WorldReader, which is also an Abstract class. WorldReaderWRD will read the self-invented .wrd file format. The Reason WorldReader is an Abstract Class is to respect the Open-Closed Principle, making it easy to add support for other file formats.

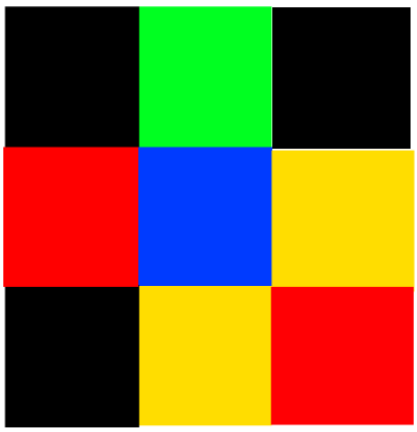
Procedural Generation (Wave Function Collapse)

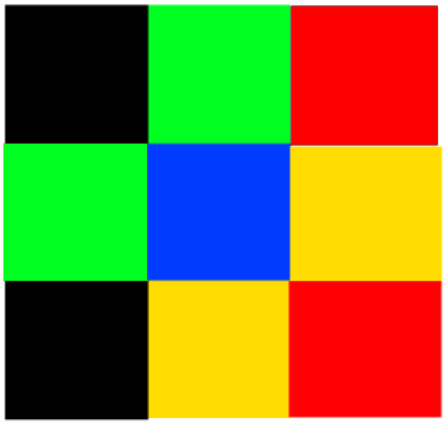
We can also generate maps using Wave Function Collapse (WFC). WFC will work on a Grid of a certain size. With WFC we will be able to easily enforce 'local' constraints. This means that we can enforce certain rules like a path is always surrounded by 2 walls (at least if we ignore intersections). We can enforce that we will never have a 2x2 wall, ….. Global changes are more difficult to resolve using WFC, for example enforcing min/max lengths on walls or guaranteeing every position is reachable.

We will first see how WFC works, in general a lot of this algorithm is independent of the Pacman Game. First, we will try to generate our map on a grid, we will try to generate a grid containing Walls, Intersections, Passages (walkable place that is not an intersection). Of course, these entries (we will refer to as 'types') have some constraints. Intersections will be surrounded by at least 3 passages for example. Passages will always have 2 walls around them.

We can hardcode these rules, but that is not ideal. Luckly, we don't have to determine these rules ourselves. A technique used for WFC is first reading a file containing a grid filled with types. We can read this grid and check the neighboring entries. We can so dynamically construct these rules. These rules will be based on the input file, and by creating a new file we can create new types of Grids. In my project Class WFCReader will be used to read a grid with types from the self-invented '.WFC' format. This will contain all the data to contruct the rules. To store and access these rules we will use the TypeRuleManager which has a map entry for each type to an TypeRule object. A TypeRule Object will contain all the data which neighbor type is allowed. In our project Neighbors are entries that are 1 entry in distance on the grid (horizontal, vertical, diagonal). Assume the image below is data from a '.WFC' file (each color represents a different type). The image are all the spots where the blue type is present.

For making the TypeRule (for type blue) we will check all the neighbors of this type. For generation we will ensure that every blue square will have a neighbor that is acceptable for type based on what is acceptable in the '.WFC' file.





In this case the
Left-Up position of blue must always be black, and the Up position always Green, but for the Right-Up position, both black and red would be fine. To map for each type which other types are acceptable on which positions we use the TypeRule Object.

We can now store which tile combinations are acceptable together, now we will generate a new grid filled with types, creating a new acceptable grid. To guarantee we have an acceptable grid, we will make sure not every possibility is possible on a certain position. Each entry on the grid we will work with exists of a Cell. Each Cell has a set of types that are still a possible option to place on that entry. At the start every entry will have every option. Each cell has also an Entropy, this value represents in our case the number of options left. None of these cells do yet have a type. We will choice the Cell with the Lowest Entropy (or random amongst lowest if same Entropy) and assign it a type among its options. The reason we use the lowest Entropy is to make sure we will have a viable grid at the end. For example, if Entropy is 1, it will just have 1 option of a type, so will take this one. We also need to reflect our changes over the rest of the grid. Here is where the term 'Wave' of WFC come. Let's continue with TypeRule of blue mentioned above.

Assume we have 5 colors and a simple 4x4 grid (representing their Entropy):

5	5	5	5
5	5	5	5
5	5	5	5
5	5	5	5

Let we now place the type blue on position (1, 1) (counting from 0):

1	1	2	5
2	1	2	5
1	1	1	5
5	5	5	5

Because the presence of the blue type, the neighboring entries have a reduction in possible options.

The not neighboring Entries are still all on 5 options, in practice this not guaranteed and will often not be so. Let's say that type red will always have a yellow type in its Right-Down neighbor. Looking at the latest grid we see that entry (2,2) has one option that will always be a read one, so we can already infer that (3, 3) will always be a yellow square. Making the following grid

1	1	2	5
2	1	2	5
1	1	1	5
5	5	5	1

Entry (3,3) has now 1 option: yellow type.

In general, the other values will also chance, and even the original 8 neighbors can change because of the change of entry (3, 3). These Changes will occur in a recursive function called (propagate) and will have a bigger chance to impact close neighbors. This algorithm will change the options a little bit like a 'wave'.

We now continue taking the lowest Entropy and giving it a type, till the entire grid is filled and we end up with a grid filled with types. One problem that might occur is the following. We have a couple options for an entry and we choice an option. We propagate the changes, and we end up with a cell with Entropy 0 (=0 options). In this case the option we just took was a bad option. So, we need to rollback this option and remove this option as a possible option for this entry. To do this I used the Memento Design Pattern. Just before we place the option and propagate the grid we store a copy of this grid into the Memento Design Pattern. If we end up with and Entropy 0, then we do an undo and remove the option from possible options. The Memento Design Pattern uses an Object as CareTaker, in my project CellChangeLog will act as the caretaker. This all will occur inside the WFCGridGenerator.

After we created a grid, we use GridAfterProcessor to do some after processing and checks to make the maps a bit nicer to play (some wall cleanup) and to guarantee that every coin is reachable. We will have a simplified grid, with value where it's clear to see which spots are walls, intersections, or passages. To create our world Object we have a WFCGenerator(Derived class of WorldLoader) to create a world (based on the data of the grid).

Bonus

For the bonus part of the assignment, I think I have multiple thinks that can be considered bonus:

- Procedural Generation (WFC).
- Generic programming is applied on different places: Vector2D, Matrix, EntityView.
- Additional Design Patterns are used: Strategy for Movement, Memento for WFC.
- Multiple keys pressed: expansion.
- UI Animations (Pacman eats Victory, …).