Wumpus: Network
Object Oriented Programming Assessment Task

Hamzah Ahmed

Contents

1. Overview

This document provides documentation for the Preliminary Object Oriented Programming Project. The appendixes provide external documentation and context useful for those studying the code.

1.1. Hunt the Wumpus: Background

Hunt the Wumpus is a classic computer game originally created in 1972 by Gregory Yob. The game is set in a series of interconnected caves, where the player must hunt a creature known as the Wumpus. The player navigates the cave system, avoiding hazards such as bottomless pits and super bats, while attempting to deduce the Wumpus's location and shoot it to win the game.

1.2. Wumpus: Network

Wumpus: Network is the result of this project. It is a game inspired by Hunt the Wumpus, adding original features. It is a graphical game with 3D and 4D levels, expanding the scope of the original game with more challenging and interesting levels.

2. Journal

2.1. 05/05/2025

git commit hash `19c52af565583322ebfb73220f7fd8ead5119bbe`

2.1.1. Progress

- Competed text based implementation of the game
- This implementation loads the game map from a JSON file
  - This is used to create the `Level` class, which holds the position of `Cave`s and `Hazard`s
  - `Hazard` has various subclasses for `Wumpus`, `BottomlessPit` and `Superbats`, which manipulate the `PlayerController` and `Level` on events such as `on_player_enter` (implemented as methods)
  - `PlayerController` has the actual player controlling functionality implemented in subclasses such as `TextPlayerController`
    - This will ease migration to a graphical implementation

2.1.2. Future improvements

- Add unit tests and documentation
- Change `Level` to an event-based system, where entities yield or return events that control the level, rather than having a tight coupling to the `Level` and `PlayerController`

2.2. 22/05/2025

git commit hash `854cebf7a5eb777694642c36225013d7ddd866ef`

2.2.1. Progress

- Added event based architecture
- This helped to decouple the Hazards from the Level
- Handling of player interaction with Hazards was also moved to Level.

2.3. 13/06/2025

git commit hash `9003447fea0d29cc30092adac94c328893a5294f`

2.3.1. Progress

- Complete migration to event based architecture
- Separate text-based game completely from core logic
- Add automated black-box testing
- Fix various edge cases (eg: shooting to room not adjacent to current room)
- Install pygame to nix shell

2.3.2. Future tasks

- Create graphical package

2.4. 20/05/2025

git commit hash `6a933ba5beb2a14ae5db1195709b73e29b06c576`

2.4.1. Progress

- Created graphical package
  - Has a stack of scenes (Main Menu, Level Select, How to Play, Playing, Paused)
  - Modal scenes can be pushed onto stack and pop'ed to preserve game state
- Used force directed graph drawing to draw level onto scene
  - However, force directed graph drawing prefers to have all edges same length
  - So level map converges onto a perspective-like map with crossings, rather than a flattened map

2.4.2. Future tasks

- I've decided against continuing with a flat 2D approach, rather using perspective projection of the actual polyhedral graph

- Hunt the Wumpus uses a dodechedral graph, so similar levels modelled after Platonic solids can be made

2.5. 20/07/2025

git commit hash `106c2d65c672cbe5ac7d59964eb6f37db517d4f2`

2.5.1. Progress

- Playable graphical version
  - Uses Renderer class to draw level, Caves and Player are Drawable instances that get sorted by depth
  - Arbitrary dimension perspective projection allows for 3D and 4D levels (or potentially higher in the future)
  - Created 4 levels (3 3D levels and 1 4D)
  - Animations when moving player or shooting to rotate view – so the player doesn't constantly have to reposition the camera
  - Spinning level graphics on main menu
  - Death counter and screen tint to indicate player death
    - I do this since this game will take a few deaths to complete due to randomness
    - Deaths are a level statistic that players can try to optimise as a *high score*

2.5.2. Future tasks

- Scene transitions
- How to play
- Credits menu
- Win screen (with high score)
- Level 5
- Level unlocking
- Sound effects

2.6. 31/07/2025

git commit hash `0d256ab3988070b43b3bfdcdfaa36fc65c97d851`

2.6.1. Progress

- How to play
- Win screen
- Level 5 completed
- Level unlocking

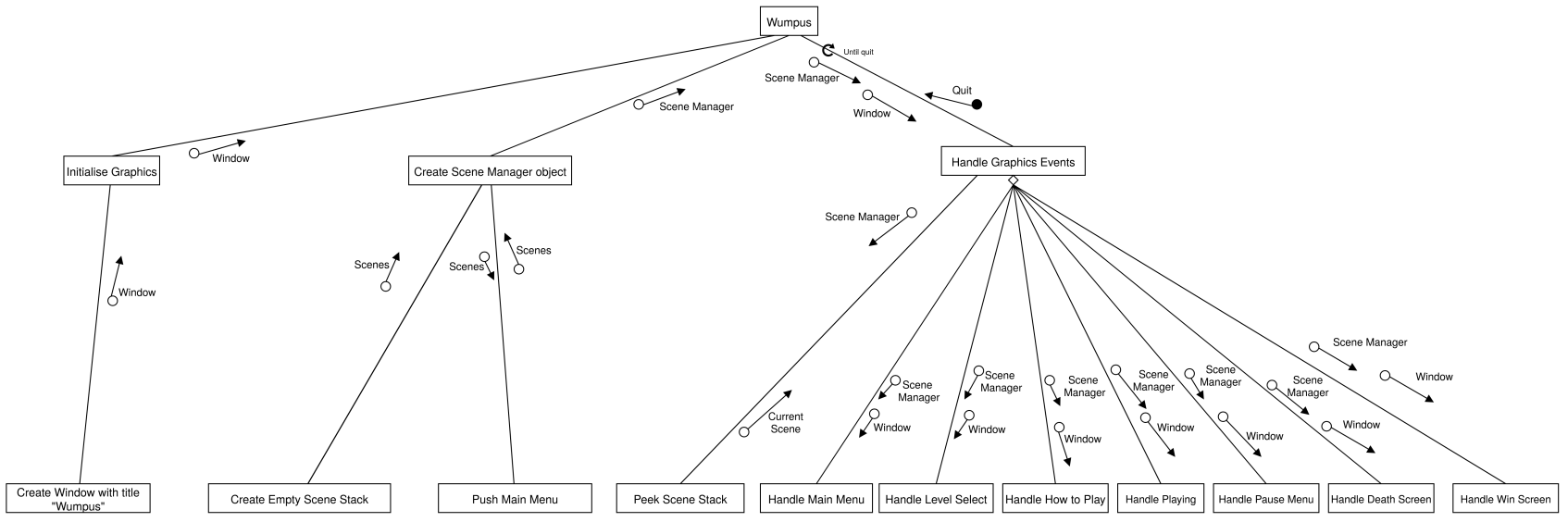
Due to time limitations I did not get to:

- Sound effects
- Credits menu (but I will include these in the folio)

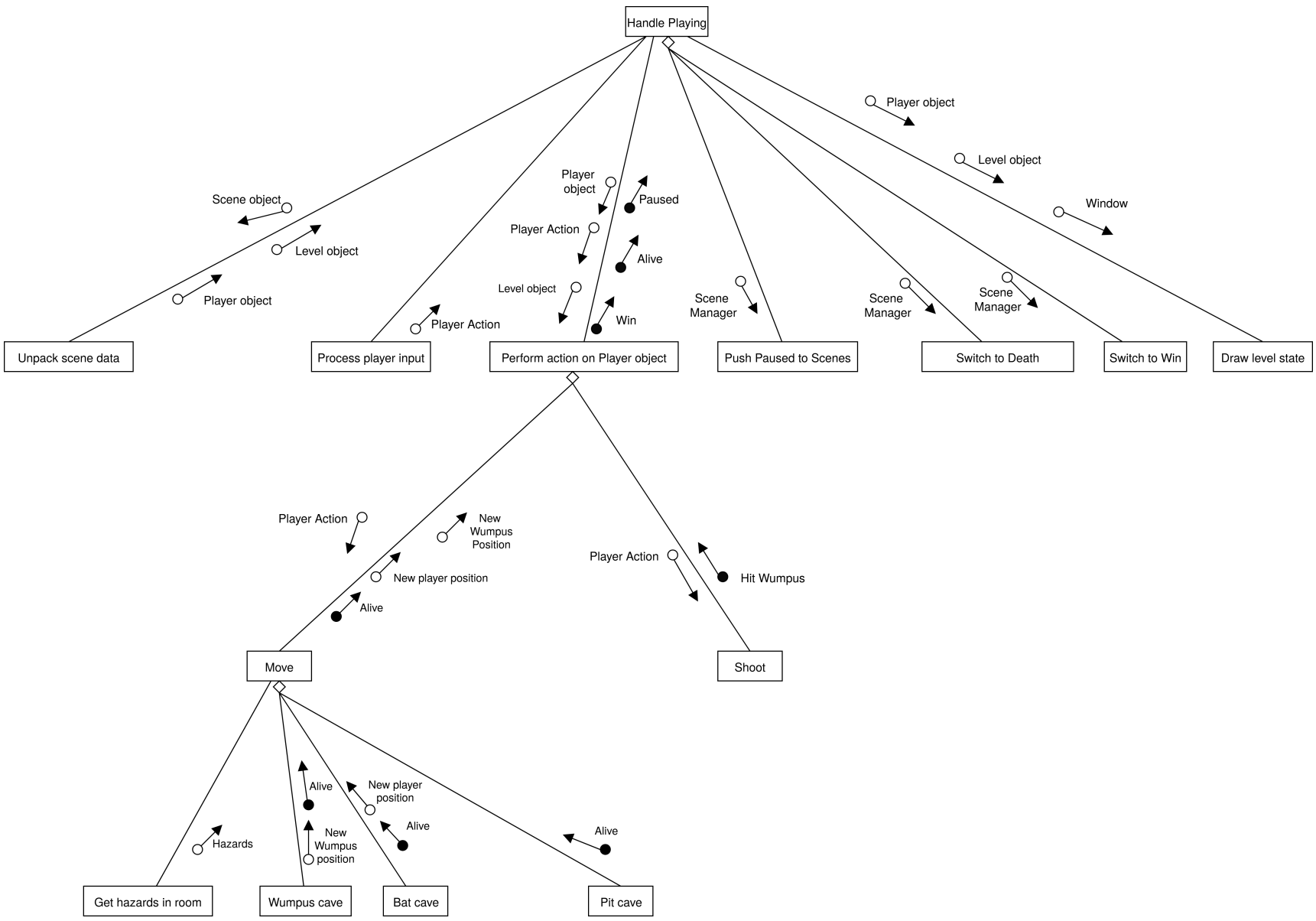
3. System Modelling

3.1. Structure Charts

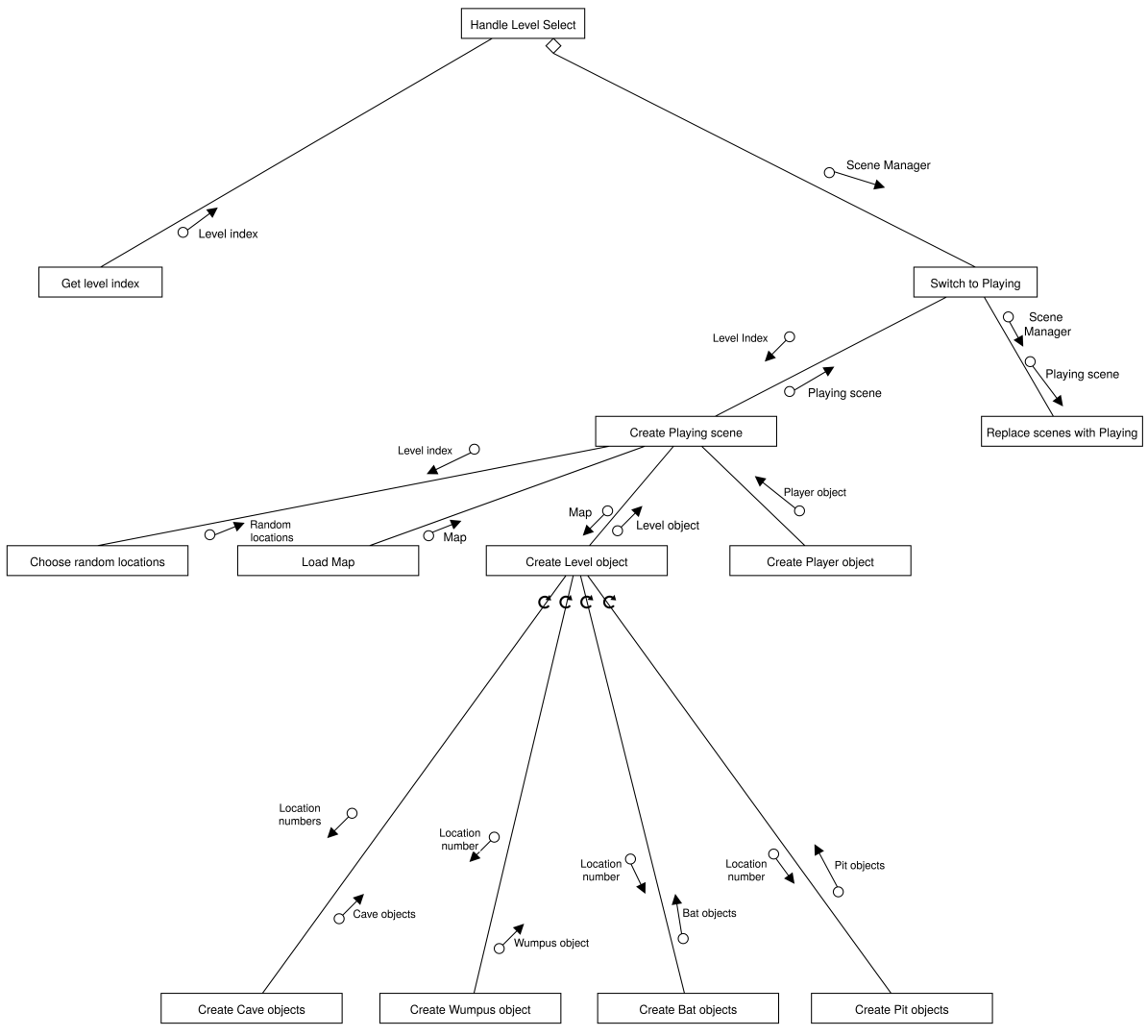
3.1.1. Wumpus



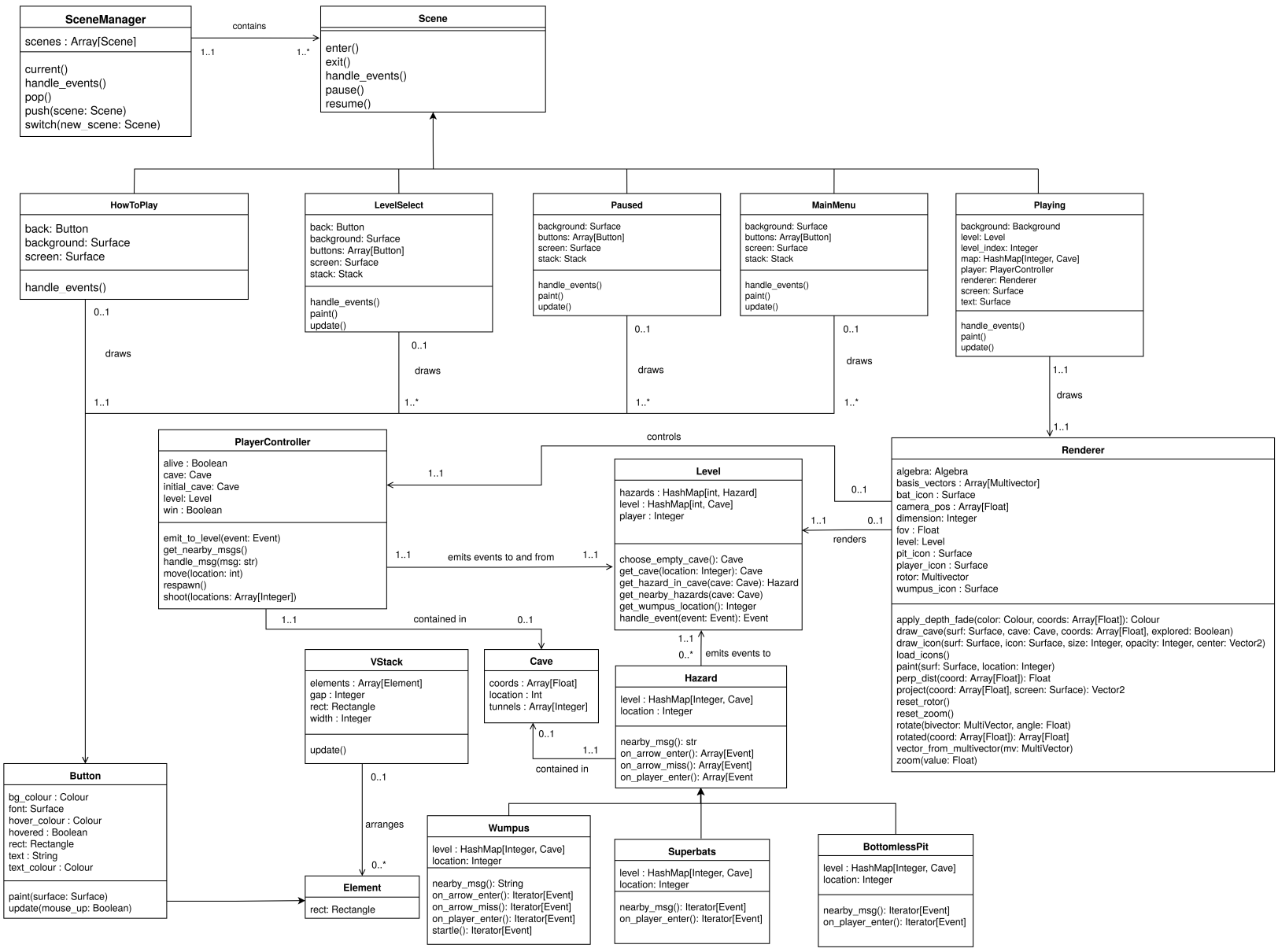
3.1.2. Handle Playing



3.1.3. Handle Level Select



3.2. Class Diagram



4. Data Dictionary

4.1. Internal representation

Data Structure (type)	Attributes	Data type	Max length	Description
PlayerController (object)	alive	Boolean	1	Whether the player is alive
	cave	Cave object	EOF	Current cave the player is in
	initial_cave	Cave object	EOF	Cave where the player first spawned in (used for respawn)
	level	Level object	EOF	Level object containing caves and hazards
	win	Boolean	1	Whether the player has won

Data Structure (type)	Attributes	Data type	Max length	Description
Level (object)	hazards	Array of hazard objects	EOF	The hazard objects in the level
	level	Hashmap of Integer to the Cave objects	EOF	Maps cave location number to cave objects
	player	Integer	3	Cave location number of the player

Data Structure (type)	Attributes	Data type	Max length	Description
Cave (object)	location	Integer	3	This cave's location number
	tunnels	Array of integers	EOF	Array of cave location numbers this cave connects to
	coords	Array of reals	EOF	3D coordinates of cave

Data Structure (type)	Attributes	Data type	Max length	Description
Hazard (object)	level	Hashmap of Integer to Cave object	EOF	Maps cave location number to cave objects
	location	Integer	3	Hazard's location number

4.2. Graphical

Data Structure (type)	Attributes	Data type	Max length	Description
SceneManager (object)	scenes	Array of Scene objects	EOF	A stack where the top scene is the one that is shown

Data Structure (type)	Attributes	Data type	Max length	Description
MainMenu (Scene)	buttons	Array of Button objects	EOF	Buttons that are shown on screen

Data Structure (type)	Attributes	Data type	Max length	Description
HowToPlay (Scene)	back	Button object	EOF	Button to close the menu

Data Structure (type)	Attributes	Data type	Max length	Description
LevelSelect (Scene)	buttons	Array of Button objects	EOF	Buttons to enter each menu
	back	Button object	EOF	Button to return to main menu

Data Structure (type)	Attributes	Data type	Max length	Description
Playing (Scene)	level	Level object	EOF	Level object containing caves and hazards (see above)
	player	PlayerController object	EOF	PlayerController to control actions of player (see above)

Data Structure (type)	Attributes	Data type	Max length	Description
DeathMenu (Scene)	buttons	Array of Button objects	EOF	Buttons that are shown on screen

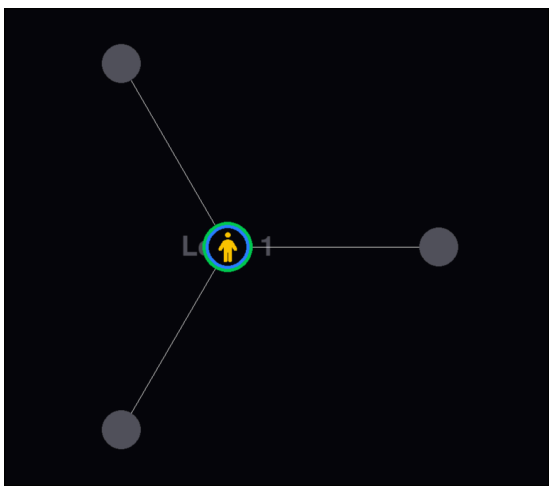
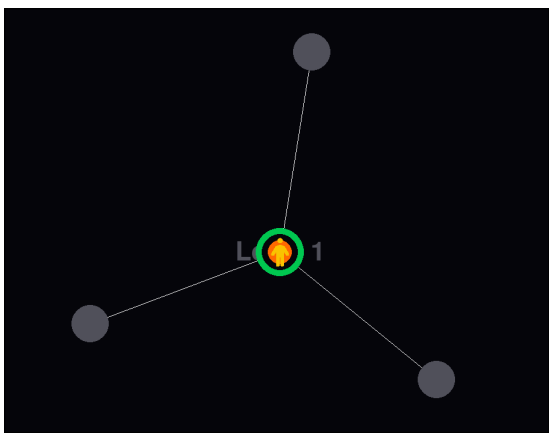
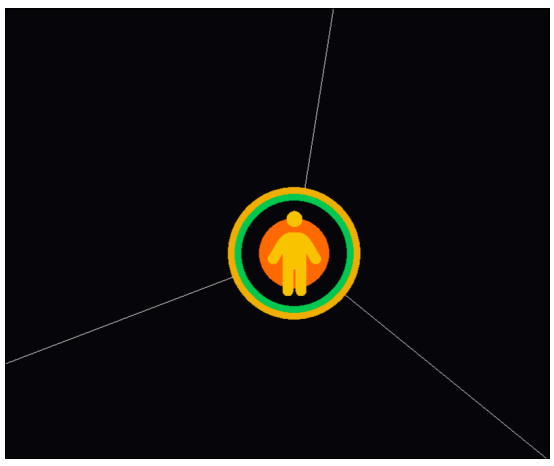
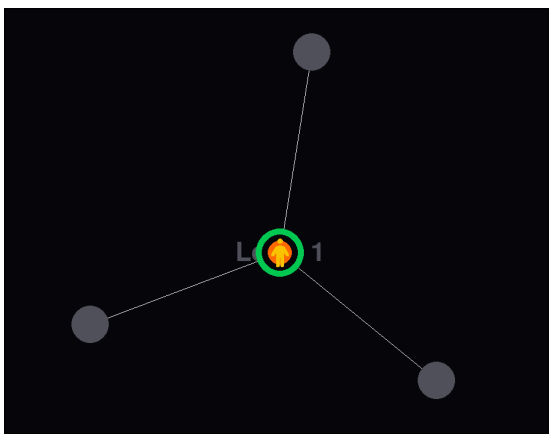
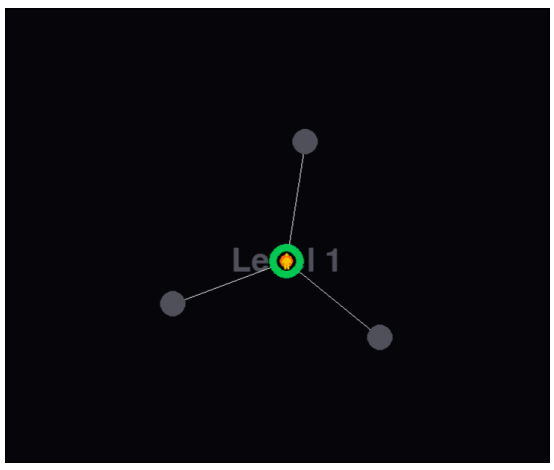
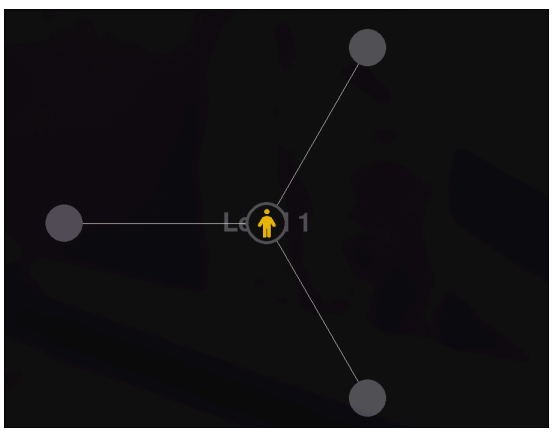
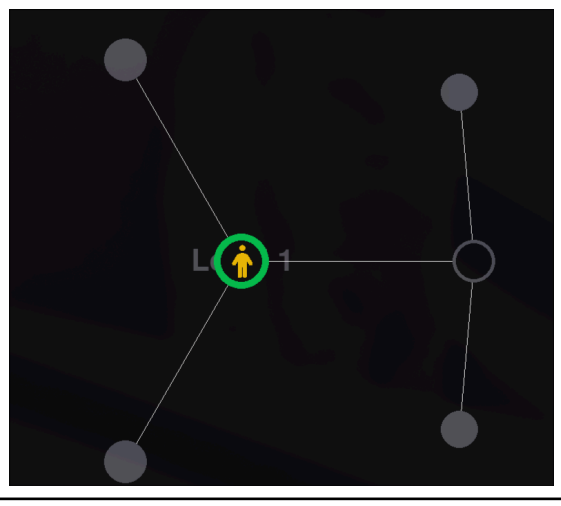
Data Structure (type)	Attributes	Data type	Max length	Description
WinMenu (Scene)	buttons	Array of Button objects	EOF	Buttons that are shown on screen

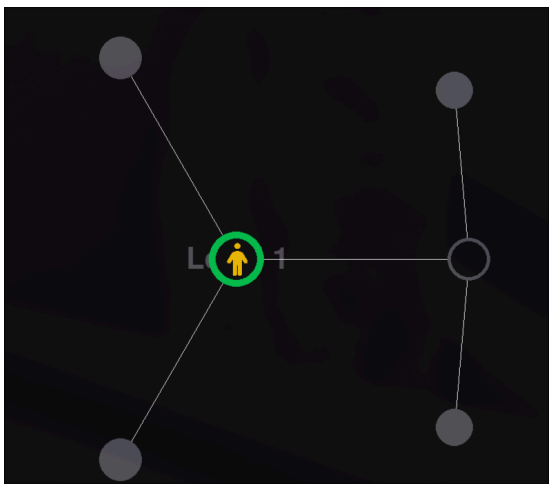
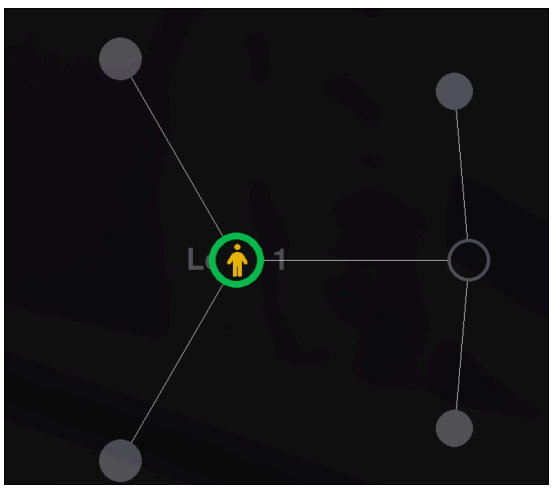
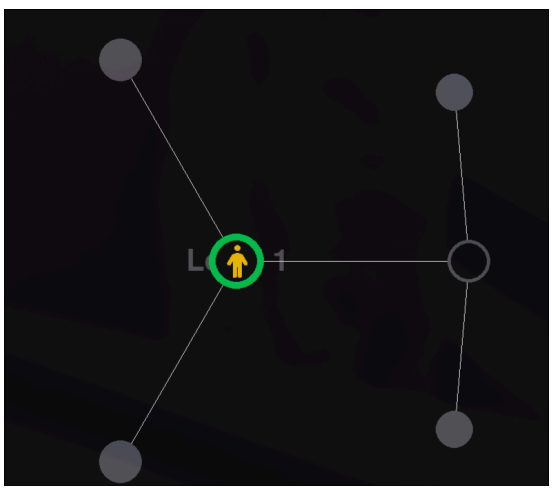
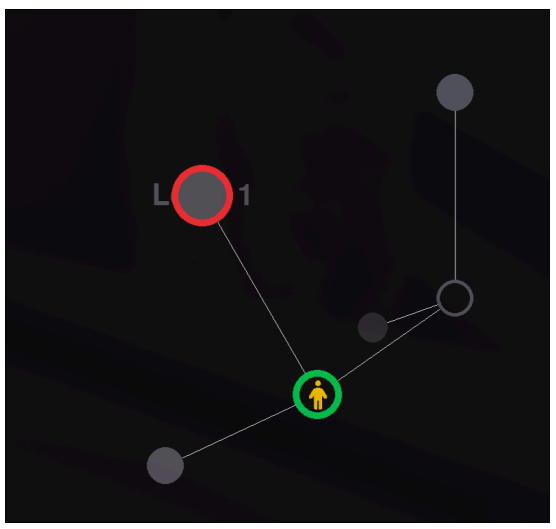
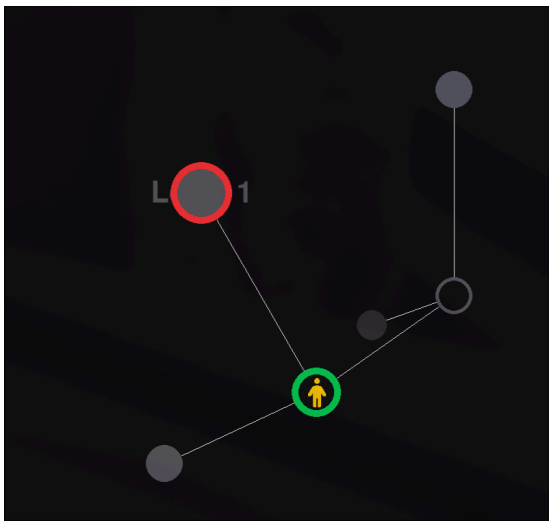
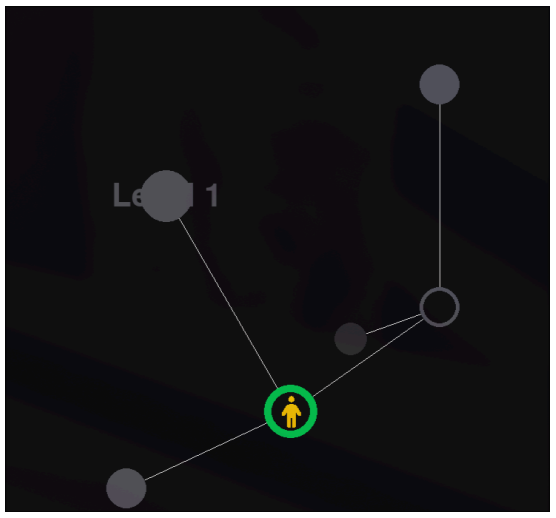
Data Structure (type)	Attributes	Data type	Max length	Description
Button (Scene)	bg_colour	RGB colour	6	Background colour of button
	font	Font file	EOF	Font file to text
	hover_colour	RGB colour	6	Background colour of button when hovered
	hovered	Boolean	1	Whether the button is currently hovered

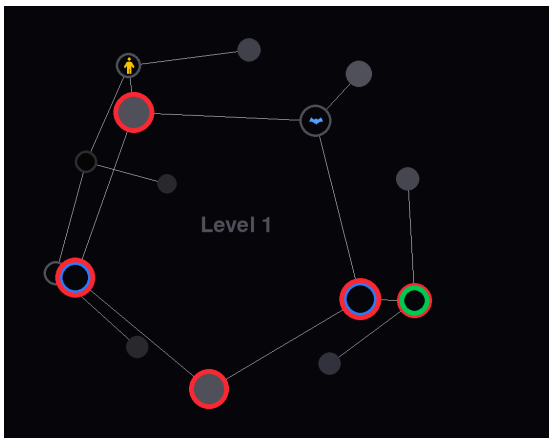
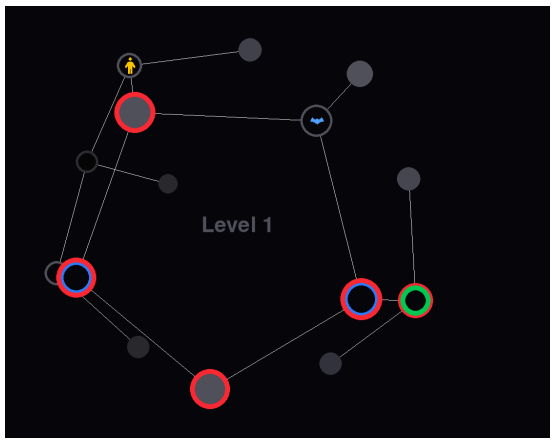
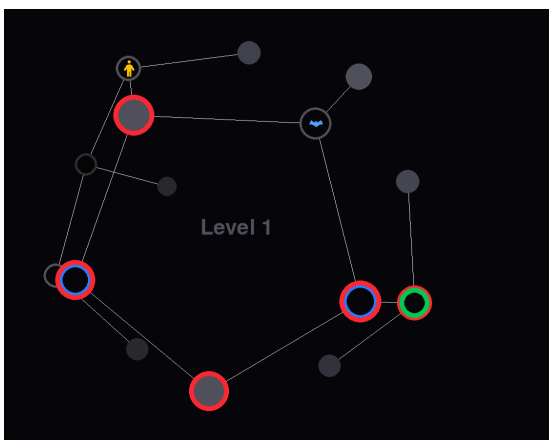
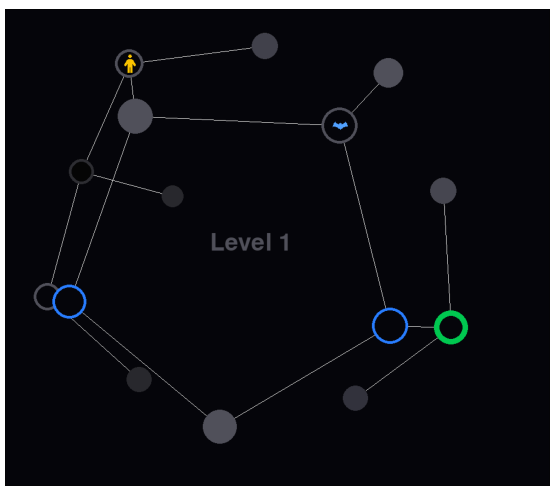
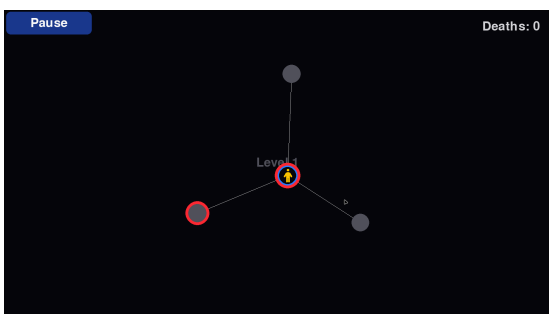
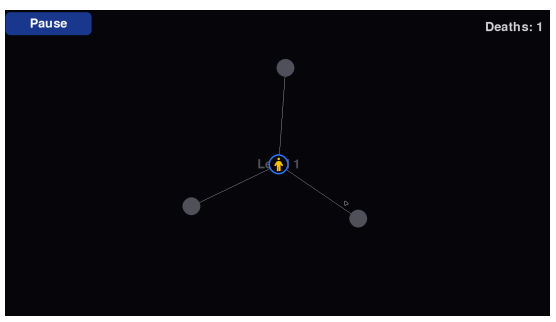
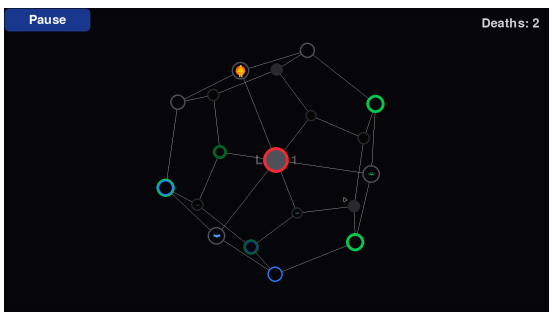
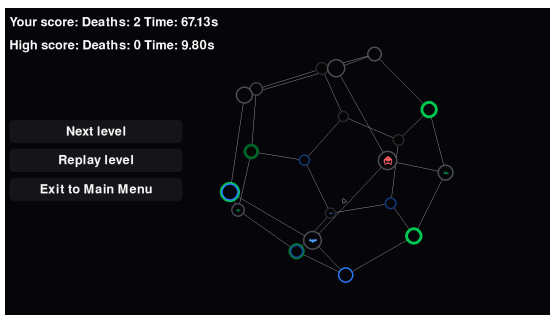
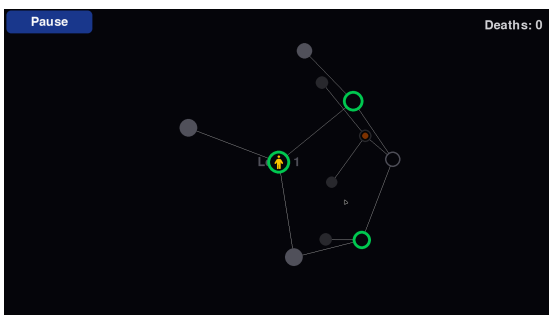
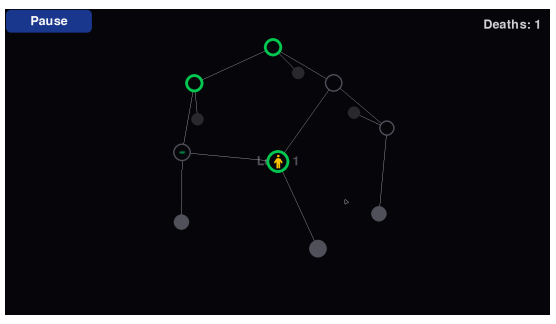
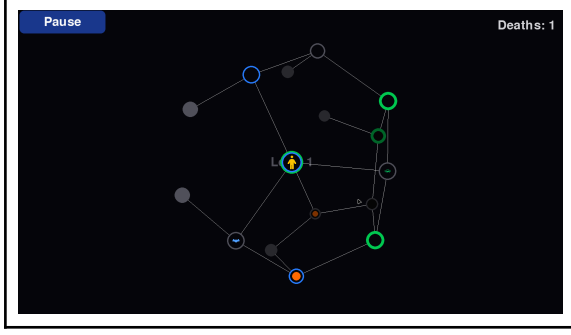
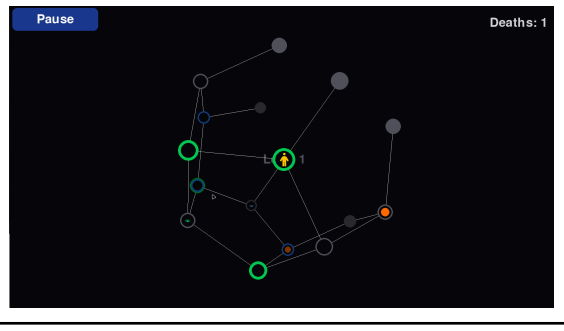
	rect	Array of integers	4	Top, left, width, height coordinates of button
	text	String	100	Label to show on the button
	text_colour	RGB colour	6	Colour to draw the text

5. Testing Strategies

5.1. Main Game

Initial state	User Action	Expected Output	Reason
	Click and drag		Test rotating view
	Scroll down		Test zooming in
	Scroll up		Test zooming out
	Left-click on cave adjacent to player		Clicking on a cave should cause the player to move to it

Initial state	User Action	Expected Output	Reason
	Left-click on cave not adjacent to player	*no change*	Player can only move to an adjacent cave
	Right-click on cave not adjacent to player	*no change*	Arrows cannot skip caves
	Right-click on a cave adjacent to player		Right-clicking specifies a shooting path (highlighted in red)
	Right-click on a cave not adjacent to end of shooting path	 *Shooting path has been reset*	Arrows cannot skip caves

Initial state	User Action	Expected Output	Reason
 *Notice the shooting path contains exactly 5 caves*	Right click on any cave	 *no change*	Arrows can travel a maximum of 5 caves
	Press the ‘c’ key		Pressing the ‘c’ key after selecting a shooting path will reset it.
 *Notice that the player’s cave is highlighted as part of the shooting path*	Shoot cave player is in (by pressing enter)		Shooting oneself leads to death and immediate respawning
	Shoot cave Wumpus is in		Shooting the Wumpus completes the level
	Moving into cave with pit		Pit leads to death and immediae respawning
	Moving into cave with bat		Bats teleport player to random cave

6. Evaluation

6.1. Implementation of Object Oriented Programming concepts

6.1.1. Encapsulation

Encapsulation is the bundling of attributes with methods to hide internal details about an object that other modules are not concerned with. An example of this is the **Facade pattern**.

The core Wumpus game logic is separated from the graphical package, as the graphical code is not concerned with it. Thus, I use the Facade pattern in `PlayerController` to provide a simplified external interface to the game logic.

```
class PlayerController:
    """
    Responsibility:
        - Emit events to level like PlayerMoved
        - Handle events such as PlayerKilled
    """

    def __init__(self, level: Level):
        self.level = level

        self.cave = level.choose_empty_cave()
        self.initial_cave = self.cave
        self.alive = True
        self.win = False

        list(self.emit_to_level(PlayerMoved(self.cave.location)))

    def move(self, location: int):
        list(self.emit_to_level(PlayerMoved(location)))

    def shoot(self, locations: list[int]):
        for location in locations:
            if any(
                [
                    isinstance(event, ArrowHit)
                    for event in self.emit_to_level(ArrowShot(location))
                ]
            ):
                return

        self.emit_to_level(ArrowMissed())

    def emit_to_level(self, event: Event) -> Iterator[ArrowHit]:
        ... # omitted for brevity

    def handle_msg(self, msg: str):
        pass

    def get_nearby_msgs(self):
        ... # omitted for brevity

    def respawn(self):
        ... # omitted for brevity
```

This class is an example of the Facade pattern, as the internal details of the event system is hidden from the other modules which use this class. Instead, other modules call the `PlayerController`'s methods to perform actions on the player. For example:

```
player.move(7)
if not player.alive:
    player.respawn()
```

6.1.2. Inheritance

Inheritance allows for a child class to reuse the methods and attributes of its parent class, while only overriding behaviour that differs. Inheritance is used as a mechanism for code reuse in the inheritance between `TextPlayerController` and `PlayerController`.

```
class TextPlayerController(PlayerController):
    def play(self) -> bool:
        """Play the game, returns whether the game was won or lost."""
        while self.alive and not self.win:
            self.get_action()
        return self.win

    def get_action(self):
        ... # text handling omitted for brevity
```

```
    def handle_msg(self, msg: str):
        print(msg)
```

This class demonstrates reuse of the core game logic defined in `PlayerController`, while adding additional functionality allowing it to be controlled using text.

6.1.2.1. Polymorphism

Inheritance is also used to achieve dynamic polymorphism, allowing multiple classes to share a common interface to implement different behaviours. This can be seen in the `Hazard` class and its subclasses.

```
class Hazard:
    """
    Hazards are located in a cave, they can affect the player's location or
    cause the player to lose.
    """

    def __init__(self, level: dict[int, Cave]):
        self.location: int | None = None
        self.level = level

    def on_arrow_miss(self) -> Iterator[Event]:
        """Called when an arrow does not hit any hazards that are not immune."""
        yield from []

    def on_arrow_enter(self) -> Iterator[Event]:
        """
        Called when an arrow hits the cave this hazard is in.

        Returns: Whether the arrow has considered to 'hit' the hazard.
        Most hazards are immune by arrows, so they should return False.
        """
        yield from []

    def on_player_enter(self) -> Iterator[Event]:
        """Called when the player enters the cave this hazard is in."""
        yield from []

    def nearby_msg(self) -> str:
        """Returns a message to be printed when a hazard is nearby."""
        return "Unknown hazard nearby."

class BottomlessPit(Hazard):
    """Kills the player when it enters the cave."""

    def nearby_msg(self):
        return "I feel a draft."

    def on_player_enter(self):
        yield "You fell into a bottomless pit."
        yield PlayerKilled()
```

Here, the `Hazard` parent class defines a common interface for hazards to define their interactions. It contains blank implementations of hazard reactions to different events. Thus, as seen in `BottomlessPit`, child classes only need to override functionality that they actually implement, while still allowing all hazards to be controlled in a unified way. This is considered polymorphism (meaning “many forms”), as objects of different classes can be used in the same way, without being concerned with what specific subclass is in use.

6.1.3. Evaluation of implementation of OOP concepts

This project demonstrates a strong implementation of OOP concepts, effectively using classes and methods to reuse functionality.

6.2. Functionality

6.2.1. Features

This project provides a complete, playable game with 5 different levels. Each level provides a unique map with randomly spawned hazards which can be interacted with. The game features menus including:

- Main Menu
- Level Select
- How to Play
- Pause Menu
- Win Menu

This game has a progress mechanism where levels are “locked”, until the level before it has been completed. This allows for a gradual increase in difficulty as the player completes the game. Futhermore, this game has a high score mechanism, which keeps track of the number of times the player has died in the level and how long the player takes to complete.

Within a level, my game implements features close to the original Hunt the Wumpus game. It includes the three original hazards: Wumpus, Bottomless Pit, Superbats and similar movement and shooting mechanics to the original game. Added functionality includes the ability to rotate the 3D view and zoom in and out (further explained in Appendix B.).

6.2.2. Polish

Polish refers to features which improve the player experience with subtle tweaks and refinements. Some examples of polish in this game:

- Camera rotation animation when clicking on a cave
- Brief red tint on screen which fades out to give an indication of player death
- Animations follow a more realistic easing curve – animations don't abruptly start and stop rather they speed up and slow down similar to real world objects
- When the mouse cursor is "hovered" over a clickable button, it changes colour subtly to indicate that it is clickable
- Borders on buttons are slightly rounded
- Important buttons like the pause button, back button or the next unplayed level in level select are a different colour for visual contrast to draw attention to them
- The Main Menu features a rotating Wumpus cave in the background for visual interest
- Upon completing a level, the entire shape of the level is revealed in a slowly rotating view

6.2.3. Possible improvements

There are some small features that could further improve player experience, but have not been implemented due to time:

- Sound effects
- Fade transitions between screens
- A settings menu to reset progress
- Text on screen to supplement visual indication of hazards
- Further levels could introduce novel hazards
- Credits in an in-game menu (see Appendix C.)

There is also a visual inconsistency in the way tunnels are rendered. The lines that represent tunnels will always render behind all caves, however sometimes this is incorrect. This is rarely noticable but fixing this could improve player experience.

6.2.4. Evaluation of functionality

This is a complete and functional game, with a high degree of polish and additional features. While it does lack some features that would be recommended for a public release of a game, it is certainly in a **playable state** with multiple levels and menus.

6.3. Originality

This project provides an original interpretation of the Hunt the Wumpus game while staying true to the original intentions of the game. The original game was text-based, containing a single level. Hunt the Wumpus was created due to the **creator's frustrations with the multitude of "hide and seek computer game[s]" based on square grids**. Yob's innovation was creating a hide and seek game where the caves were a **non-grid pattern**. However, **graphical versions of the game present their levels on a square grid**, usually by **"flattening"** the caves network by forcing them to fit on a square grid. **This ultimately limits the complexity of levels in these games to those that fit on a grid.**

This game provides a different interpretation, **preserving the non-grid pattern of the original game**. It does this through arbitrary dimension perspective projection and rotation (described in Appendix B.). What this means is the game rendering logic can draw levels of any dimension greater than or equal to 3. To effectively explore these levels, this game provides **zoom and rotate functionality**. **There are 3 3D levels and 2 2D levels included**. The mechanisms and intuition for understanding higher dimensions than 3 are provided in Appendix B., but they are not necessary to be able to play the levels. The purpose of these levels is to present a **further challenge to the player** that would not be common in other video games.

6.3.1. Levels

All levels have cave structures based on different mathematically interesting shapes.

Level 1 Dodecahedron (3D) *This level was also used in the original textual **Hunt the Wumpus**, but here it is playable in 3D.*
Level 2 Icoasahedron (3D)
Level 3 Mobius strip (3D)
Level 4 Tesseract (4D)
Level 5 24-cell (4D)

6.3.2. Evaluation of originality

Wumpus: Network is a highly original interpretation of **Hunt the Wumpus**, providing unique features that are rare even for video games in general. It does this while preserving the experience of playing the original game, by keeping the same core mechanic of deducing where hazards are based on the current cave. While my game does not add new hazards or interactions compared to the original version, it instead explores how this could be extended in more complex and challenging levels, as well as a novel interface.

This project implements a feature set significantly different provided in the support resource. Even without the graphical implementation, **this project implements key features of the textual Hunt the Wumpus not present in the guided walkthrough**, such as the ability for the Wumpus to move between caves, and randomly spawned hazards.

6.4. Documentation

This project has both internal and intrinsic documentation within the codebase.

6.4.1. Intrinsic

Intrinisic documentation makes the code more understandable by using clear structure and naming within the structures of the code itself.

Examples:

```
@dataclass
class Cave:
    location: int
    tunnels: list[int]
    coords: tuple[float, ...]
```

The class, attributes and type annotations within this code make the purpose and structure of the code clear.

```
class Superbats(Hazard):
    def nearby_msg(self):
        return "Bats nearby."

    def on_player_enter(self):
        yield "ZAP -- Super bat snatch! Elsewhereville for you!"
        yield PlayerMoved(choice(list(self.level.values())).location)
```

The class and method names in this code make it clear what each hazard does.

```
class PlayerController:
    def __init__(self, level: Level):
        self.level = level
        self.cave = level.choose_empty_cave()
        self.initial_cave = self.cave
        self.alive = True
        self.win = False
```

The names of the attributes in this class make the player's initial state clear. The `level.choose_empty_cave()` method has a descriptive name to help readers of the code understand its purpose.

6.4.2. Internal

The need for internal documentation is reduced through intrinsic documentation, however it is still important to describe the purpose of code, or to clarify complex procedures.

In Python, internal documentation takes the form of "docstrings" (special comments tied to a specific method, function, class or module) or inline comments.

Examples:

```
class PlayerController:
    """
    Responsibility:
        - Emit events to level like PlayerMoved
        - Handle events such as PlayerKilled
    """
    ...
```

This class-level docstring explains the overall responsibility of the `PlayerController` class, making its role in the system clear.

```
class Hazard:
    """
    Hazards are located in a cave, they can affect the player's location or
    cause the player to lose.
    """
    ...
    def on_player_enter(self) -> Iterator[Event]:
        """Called when the player enters the cave this hazard is in."""
        yield from []
```

Here, both the class and method docstrings describe the purpose and expected behavior, clarifying how hazards interact with the player.

```
def emit_to_level(self, event: Event) -> Iterator[ArrowHit]:
    # This method sends an event to the level and yields any ArrowHit events in response.
    ...
```

This inline comment explains a non-obvious implementation detail about how events are handled and what is yielded.

```
def on_arrow_enter(self) -> Iterator[Event]:
    """
    Called when an arrow hits the cave this hazard is in.

    Should yield an ArrowHit event if the arrow has considered to 'hit` the hazard.
    Most hazards are immune by arrows, so they should not yield this event.
    """
    yield from []
```

This method-level docstring provides additional context about the method's purpose and its expected return value, especially clarifying the default behavior for most hazards.

```python
class BottomlessPit(Hazard):
    """Kills the player when it enters the cave."""
    ...
```

A concise class docstring here makes the effect of the `BottomlessPit` hazard immediately clear to readers.

```
"""
Events are used to decouple the Hazards from the Level.
Instead of directly invoking methods on Level, Hazards yield Events, which
are dispatched to Level. This also prevents circular type imports between Level
and Hazards.
"""
```

This module level docstring, makes it clear the overall purpose of this section of code before readers go into specific classes or methods.

6.5. Evaluation of documentation

This project is thoroughly documented with both internal and intrinsic documentation.

Appendix A. Code structure

The use of LLMs was allowed in this project.

```
wumpus/
├── graphical/     # Graphical user interface: scenes, icons, GUI utilities
├── level_gen/     # Level generation logic and algorithms for different map types
├── text/          # Text-based interface and logic for terminal gameplay
└── wumpus/        # Core game logic: cave structure, events, hazards, player, levels
```

A.1. Commands

1.1.1. Running Unit Tests

```
python -m unittest
```

1.1.2. Playing graphical

```
python -m graphical
```

1.1.3. Playing text

```
python -m text
```

1.1.4. Exporting folio to pdf

```
cd folio
typst compile main.typ
```

Appendix B. Arbitrary Dimension Perspective Projection and Rotation

B.1. Perspective Projection

In this project, the game world is not restricted to the familiar two or three dimensions, but can exist in any number of spatial dimensions. Each point within the world is described by a list of real numbers. For example, a point in four-dimensional space is represented as (x, y, z, w), where each coordinate is a real number indicating position along that axis.

To render such a world on a two-dimensional screen, the system employs a process called perspective projection. The first two coordinates (typically x and y) are selected to define the image plane. The image plane is the flat surface onto which the world is projected, corresponding to the screen itself. The third coordinate (such as z in 3D) is treated as the depth direction, representing the line of sight into the scene. In higher dimensions, the choice of which coordinate acts as depth can be adjusted, allowing the player to explore and visualize the extra dimensions by rotating the view.

Each point, defined by its real-number coordinates, is mathematically mapped onto the image plane. The further a point lies along the depth direction, the smaller and dimmer it appears, creating a convincing sense of perspective. This mimics the way distant objects appear in real life, even when those objects exist in dimensions beyond our direct experience.

B.2. Rotation

Rotation is achieved using a field of mathematics called Geometric Algebra (also known as Clifford Algebra, or shortened as GA). While a full description of the field is out of scope, a brief relevant overview is provided below.

In this field, vectors (visualised as oriented line segments) are generalised to bivectors (visualised as oriented areas), trivectors (visualised as oriented volumes) and in general a k-grade element is a k-vector. In GA, we operate on objects called multivectors, which are the linear combination of different grade elements. For example 3D multivectors are $\mathrm{scalar} + \mathrm{vector} + \mathrm{bivector} + \mathrm{trivector}$, and this generalises to arbitrary dimensions.

In 3D we often think about rotations as occurring around an axis, however generally this is nonsensical. Consider that in 2D there is no orthogonal axis for the rotation to occur “around”, and in 4D there is more than one unique direction perpendicular to any given plane! Thus, it is more correct to treat rotations as occurring *in* a plane. To represent this, we use bivectors which are our 2 dimensional elements in the sum that constitutes multivectors.

Thus we see rotation as an operation involving **a plane and an angle**. It is trivial to obtain a bivector between any two arbitrary vectors, so this is very useful for animating the camera rotation between two caves.

Geometric algebra is extremely powerful, as it allows for concepts such as arbitrary dimension rotation to be expressed very concisely. Below are the equations used in this project. An explanation of these is out of scope but a reference is provided in Appendix C.

Rotate vector by rotor:

$$v' = RvR^{-1}$$

where v' is the rotated vector, v is the initial vector and R is the rotor. This takes a vector and rotates it by the current rotor (which is initialised to the scalar 1).

Update rotor by bivector and angle:

$$R' = e^{B\frac{\theta}{2}}R$$

where R' is the updated rotor, B is the bivector defining the plane of rotation, θ is the angle to rotate in radians and R is the initial rotor. This equation will take the current rotor (representing the current view angle) and rotate it by a plane and an angle. Notice the $\frac{\theta}{2}$, is because this rotor will be applied twice (multivectors are non-commutative). You may also notice that this looks simiar to rotation of complex numbers in 2D or almost identical to rotation of quaternions in 3D. This is because GA provides a generalisation that is isomorphic to these constructs in lower dimensions.

Appendix C. Credits

C.1. Graphics

- Wumpus icon – Hunt the Wumpus box art
- Bat by Gabriele Malaspina from “https://thenounproject.com/browse/icons/term/bat/” (CC BY 3.0)
- Icons – Font Awesome

C.2. Libraries

- Graphics – pygame
- Vector – numpy
- Geometric Algebra – kingdon

C.3. Lessons

- Geometric Algebra – A Swift Introduction to Geometric Algebra - YouTube