COMP1206/Coursework

From NotesWiki
< COMP1206
Jump to navigationJump to search

Timetable | Labs | Coursework | Exam | Back to COMP1206

Coursework Introduction (https://secure.ecs.soton.ac.uk/notes/comp1206/ofb/Presentation8.pptx) (PDF (https://secure.ecs.soton.ac.uk/notes/comp1206/ofb/Presentation8.pdf)) (Video (https://secure.ecs.soton.ac.uk/notes/comp1206/ofb/Presentation8.php)) Coursework Q&A Session (https://youtu.be/d-HV-HNQ9_I)

Contents

Introduction

This is a major piece of coursework worth 40% and will put into practice all the skills you have developed in the module so far into building a challenging application – but one not too dissimilar to the ECSChat application you have developed already.

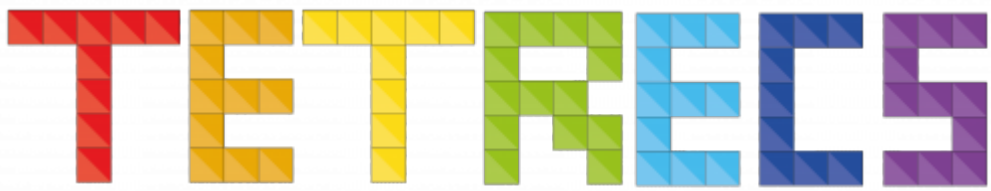
It is expected that you will spend around 60 hours on this coursework.

This coursework builds on everything you have learnt and practiced so far:

- JavaFX
- Custom Components
- Graphics and Animation
- Listeners, Properties and Binding
- Communications
- Media
- Files

If you have any problems, please go back and reference the lecture material and ensure you have worked through the corresponding labs – this coursework uses the same techniques but in a different application – your job is to apply your understanding and what you've learnt so far to solve a similar but different problem: In this case – a game!

TetrECS



Your challenge for this coursework is to build a game called TetrECS:

- A fast-paced block placement game
- You have a 5x5 grid
- You must place pieces in that grid
- You score by clearing lines, horizontally or vertically
- You can rotate pieces
- You can store a single piece to come back to later
- The more lines you clear in one go, the more points you get
- Every piece that you play that clears at least one line increases your score multiplier
- As your score goes up, so does your level, and you get less time to think
- If you fail to place a block, you lose a life

- Lose 3 lives, and you're out of the game

Sections

The Coursework is split into sections, gradually increasing in difficulty:

- Creating the Game Logic
- Building the User Interface
- Adding Events
- Adding Graphics
- Adding a Game Loop
- Adding Scores
- Adding an Online Scoreboard
- Adding Multiplayer
- Extensions

Getting Started

Just like the labs, we will provide you with a basic skeleton application to get you going. We have also provided you with a demo so you know what you're aiming for.

You can find both the skeleton application and the demonstration here:

git clone http://ofb-labs.soton.ac.uk:3000/COMP1206/coursework.git

The Skeleton

While you do not have to use everything inside the skeleton, we strongly recommend that you base your project on it.

The skeleton provides the following:

- Components
- Events
- Game
- Network
- Scene
- UI
- App/Launcher
- Assets

Components

- **GameBlock**
  - Is a JavaFX custom component extending a Canvas
  - Displays an individual block
- **GameBoard**
  - Is a JavaFX custom component extending a GridPane
  - Holds all the GameBlocks in a grid
- **GameBlockCoordinate**

- Holds an x and y column and row number of a GameBlock in the GameBoard

Events

- We supply a couple of Listeners to get you started
  - **BlockClickedListener**: Handle a block being clicked
  - **CommunicationsListener**: Handle receiving a message from the server
- You will need to add more later
- Remember, these are just interfaces with a single method

Game

This package holds the model and game logic

- **Game**: Handles the game logic
- **GamePiece**: Handles the model of a piece
- **Grid**: Handles the model of the grid
  - **This is displayed by the GameBoard**

GamePiece

- Already has the 15 different pieces defined for you
- Every piece holds a 2D block array representing the piece
- Call the static **createPiece** method
- A GamePiece holds a grid of blocks – with a value of 0 for an empty block, or a value greater than 0 representing a block (which is used as it's colour in rendering)
- The centre of a GamePiece is used for placing it (the middle of the grid)

GameGrid

- Holds a 2D array of the grid
- SimpleIntegerProperties are used to represent each block
- The GameBoard binds to these properties

- Update the grid, update the graphical GameBoard representation

Network

- Communicator
  - Works the same way as ECS Chat
  - You can add listeners to receive messages
  - You can send messages

- The server that is connected to is ofb-labs.ecs.soton.ac.uk:9700
  - This is different to the ECS Chat Server
  - This is a dedicated server for TetrECS
  - You need to be on the VPN to reach this server

Scenes

- Each Scene represents a "screen" in the game
- You will be adding more later
  - For example
    - Intro
    - Instructions
    - Multiplayer (Lobby, Game)
- BaseScene
  - Provides a base scene the others inherit from
  - Basic functionality
- ChallengeScene
  - The single player challenge UI
- MenuScene
  - Displayed when the game is launched

UI

Provides useful parts of the User Interface

- You shouldn't really need to change these
- GamePane
  - This is a special pane which will scale all it's internal content, adding padding to ensure correct aspect ratio
- GameWindow
  - The single window that switches scenes to change screen in the game

App/Launcher

- App
  - The JavaFX Application
- Launcher
  - Starts the application
- As with ECS Chat

- You shouldn't need to worry about these

Assets

You are welcome to make the game look like how you want — with your own sound effects, graphics, music, theme. However, for those who want an easy life, we are providing you with all the assets as found in the demo game!

- The font
- The graphics
- The sounds and music
- The CSS

Your Tasks

- Creating the Game Logic
- Building the User Interface
- Adding Events
- Adding Graphics
- Adding a Game Loop
- Adding Scores
- Adding an Online Scoreboard
- Adding Multiplayer
- Extensions

- **Please note**: The documentation below is a suggested guide to implementing it. Some of the concepts are key to follow (e.g. separation of view and model, listeners and properties etc.) but you are free to follow your own implementation if you prefer, as long as you hit the main goals and match the marking scheme. We are not using automated testing so you do not need to follow class or method names or follow the suggested implementation.

Before you Begin

- **Use Logging**: This will be a complicated application, especially as you get further in.
  - Add logging statements at the start of every function you write which outputs what method has been called and with what parameters – this will make it much easier to see what's going on.
  - Add further helpful logging information to highlight what is taking place, how and when
  - The investment will pay off when it comes to working out what's not working!

Game Logic

The first step to implementing the game is to implement the basic game logic. This includes:

- Add the logic to handle placing pieces
  - Can a piece be played?
  - Place a piece onto the grid
- Add the logic to keep track of pieces
  - Keep track of the current piece
  - Create new pieces on demand
- Add the logic to handle when a piece is played
  - Clear any lines

To do this:

- **In the Grid class:**
  - Add a **canPlayPiece** method
    - Which takes a GamePiece with a given ***x*** and ***y*** of the grid will return true or false if that piece can be played
  - Add a **playPiece** method
    - Which takes a GamePiece with a given ***x*** and ***y*** of the grid will place that piece in the grid
    - Tip: Remember, you will need to offset the x and y co-ordinates to ensure a piece is played by it's centre!
- **In the Game class:**
  - Add a **spawnPiece** method to return a GamePiece
    - Create a new random ***GamePiece*** by calling ***GamePiece*.createPiece**
  - Add a **currentPiece** ***GamePiece*** field to the ***Game*** class
    - This will keep track of the current piece
  - When the game is initialised, spawn a new **GamePiece** and set it as the ***currentPiece***
  - Add a **nextPiece** method
    - Replace the current piece with a new piece
  - Update the **blockClicked** method to play the current piece from it's centre if possible, then fetch the next piece
  - Add an **afterPiece** method, and add logic to handle the clearance of lines
    - To be called after playing a piece
    - This should clear any full vertical/horizontal lines that have been made
      - You will need to iterate through vertical and horizontal lines, keeping track of how many lines are to be cleared and which blocks will need to be reset
      - Tip: You may find a HashSet to be helpful here (so a block doesn't get added/cleared/counted twice)
    - Any horizontal and vertical lines that have just been made should be cleared (including intersecting lines – multiple lines may be cleared at once).

Need a bit of help getting started? We go over some of the initial steps in the Q&A session (https://youtu.be/d-HV-HNQ9_I)

Clearing Lines: Additional Help

If you're struggling on how to clear lines, perhaps this example will help:

Consider the grid

```
1 1 1 1 1
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
```

Here, we need to clear two lines and 9 blocks.

We start by iterating through from left to right, top to bottom, and keep track of what we see. (Search for horizontal lines)

We need to store: Number of lines to clear, the blocks to clear (which we hold as a set to avoid duplicates)

Iteration 1: 5/5 = +1 line and 5 blocks to clear (we add those blocks to the set)

Iteration 2,3,4,5: 1/5 = no lines, no blocks to clear

Then we need to iterate from top to bottom, left to right (Search for vertical lines)

Iteration 1: 5/5 = +1 line, and 5 blocks to clear (we add those blocks to the set – the block in the top left corner is already in the set so doesn't get readded)

Iteration 2,3,4,5: 1/5 = no lines, no blocks to clear

Result: 2 lines to clear, 9 blocks to clear (the size of the set), and we have a set of all the blocks to clear

Sort out the score, then iterate through the set of blocks to clear and set them to 0

Build the User Interface

Next, we want to build the basics of the user interface for the game:

- The user interface should keep track of
  - Score
  - Level
  - Lives
  - Multiplier
- Show these in the UI
- Update them appropriately when events happen
  - Implement Scoring
  - Implement Multiplier
- Add Background Music
- Add Appropriate Styles

To do this:

- Add **bindable properties** for the **score**, **level**, **lives** and **multiplier** to the ***Game*** class, with appropriate accessor methods.
  - These should default to 0 score, level 0, 3 lives and 1 x multiplier respectively.
- Add UI elements to show the score, level, multiplier and lives in the ***ChallengeScene*** by binding to the game properties.
  - Tip: Use the .asString method on an Integer Property to get it as a bindable string!
- In the ***Game*** class, add a **score** method which takes the **number of lines** and **number of blocks** and call it in *afterPiece*. It should add a score based on the following formula:
  - number of lines * number of grid blocks cleared * 10 * the current multiplier
  - If no lines are cleared, no score is added
  - For example, if a piece was added that cleared 2 intersecting lines, 2 lines would be cleared and 9 blocks would be cleared (because 1 block appears in two lines but is counted only once) – this would be 180 points with a 1 times multiplier (compared to 200 points if 2 non–intersecting lines were cleared at the same time)
- Implement the multiplier
  - Every time you clear *at least one line* with a piece, the multiplier increases by *exactly 1*
  - The multiplier *resets* as soon as you play a piece or a piece expires without clearing *any* lines
  - The multiplier is increased by 1 if the next piece also clears lines. It is increased **after** the score for the cleared set of lines is applied
  - The multiplier is reset to 1 when a piece is placed that doesn't clear any lines
  - Example: If you clear 4 lines in one go, the multiplier increases once (now at 2x). If then clear 1 line with the next piece, the multiplier increases again (now at 3x). If you then clear 2 lines with the next piece, it increases again (now at 4x). The next piece you play clears no lines (multiplier resets to 1x)
- Implement the level
  - The level should increase per 1000 points (at the start, you begin at level 0. After 1000 points, you reach level 1. At 3000 points you would be level 3)
- Create a ***Multimedia*** class

    - Add two MediaPlayer fields to handle an audio player and music player
        - These need to be fields, not local variables, to avoid them being garbage collected and sound stopping shortly after it starts
    - Add a method to play an audio file
    - Add a method to play background music
        - The background music should loop
- Implement background music on the Menu and in the Game using your new class
- Ensure everything is styled in some way
    - You may create your own aesthetics with as much freedom as you want, or use the default styles and assets as provided, but we expect you to show you can apply them

- Important: Use listeners to link your Game (the game model and state) and the GameBoard (the UI) via the ChallengeScene. Do not directly include the GameBoard inside your Game (read the FAQ as to why!)

Enhance the User Interface

Now we have the basics down, we want to enhance the user interface further:

- Make a better Menu
- Add an Instructions Screen
- Make a custom component to show a specific piece
- Add a listener for handling a next piece being ready
- Use the component the upcoming piece in the UI

To do this:

- Create a **PieceBoard** as a new component which extends ***GameBoard***
    - This can be used to display an upcoming piece in a 3x3 grid
    - It should have a method for setting a piece to display
    - Add it to the ***ChallengeScene***
- Update the **MenuScene**
    - Add pictures, animations, styles and a proper menu
    - Add appropriate events by calling the methods on ***GameWindow*** to change scene
- Create a new **InstructionsScene**
    - This should show the game instructions
    - Add an action from the Menu to the Instructions
- In the **InstructionsScene**, add a dynamically generated display of all 15 pieces in the game

-
    - You can create a GridPane of **PieceBoards**
    - You do not need to worry about handling more than the 15 default pieces
- Add keyboard listeners to allow the user to press escape to exit the challenge or instructions or the game itself
    - Tip: You want to listen for keyboard input on *the scene* – if you try to add it to a control, how would it know which control should receive the event?
    - Tip: You will need to add a method to shutdown the game in the ChallengeScene to end and clean up all parts of the game, before going back – or it'll keep playing!

- Create your own **NextPieceListener** interface which a **nextPiece** method which takes the next ***GamePiece*** as a parameter
    - What we are aiming for: The ChallengeScene (UI) will register a NextPieceListener on the Game (model), so that when a new piece is provided by the game, it can update the PieceBoard with that piece (so the ChallengeScene acts as the bridge between the model – game – and the pieceboard component)

- Add a **NextPieceListener** field and a **setNextPieceListener** method to ***Game***. Ensure the listener is called when the next piece is generated.
- Create a **NextPieceListener** in the ***ChallengeScene*** to listen to new pieces inside game and call an appropriate method.
    - In this method, pass the new piece to the **PieceBoard** so it displays.

Events

Now we need to handle some of the main events and actions in the game:

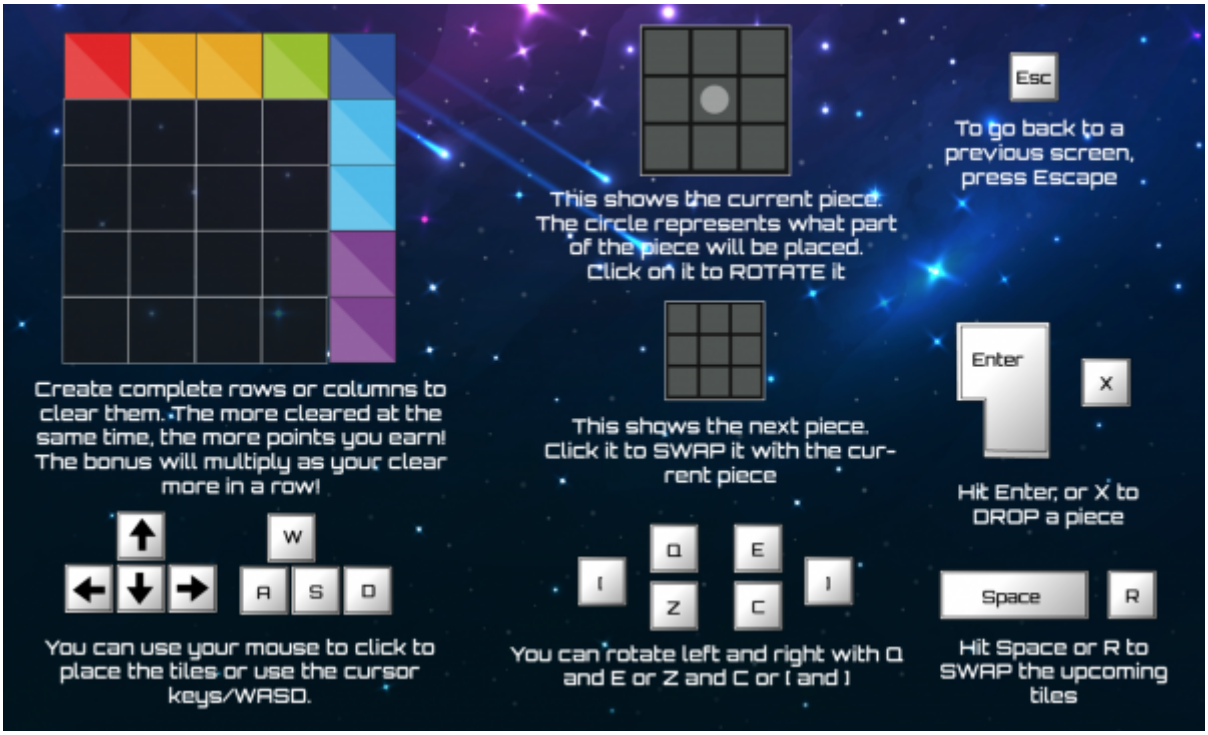
- Add the next tile in advance
- Add piece rotation
- Add piece swapping
- Add sound effects
- Add keyboard support

To do this:

- Add a **rotateCurrentPiece** method in ***Game*** to rotate the next piece, using **GamePiece's** provided rotate method
- Add a **followingPiece** to ***Game***. Initialise it at the start of the game.
- Update **nextPiece** to move the following peice to the current piece, and then replace the following piece.
- Add another, smaller ***PieceBoard*** to show the following peice to the ***ChallengeScene***
- Update the ***NextPieceListener*** to pass the following piece as well, and use this to update the following piece board.
- Add a **swapCurrentPiece** method to swap the current and following pieces
- Add a **RightClicked** listener and corresponding setOnRightClicked method to the ***GameBoard***
- Implement it so that right clicking on the main **GameBoard** or left clicking on the current piece board rotates the next piece
- Add sounds on events, using your ***Media*** class, such as placing pieces, rotating pieces, swapping pieces.
- Add keyboard support to the game, allowing positioning and dropping pieces via the keyboard.
    - You will need to keep track of the current aim (x and y)
    - Drop the piece on enter

- Move the aim when relevant buttons are pressed

Suggested Controls



Graphics

Let's now enhance the graphics by working with our canvas:

- Add tiles to the game, not just squares
- Add hovering
- Add animations on clearing to show tiles cleared

To do this:

- Update the ***GameBlock*** drawing to produce prettier filled tiles and empty tiles
- Update the ***PieceBoard*** and ***GameBlock*** to show an indicator (e.g. a circle) on the middle square
  - Ensure that any pieces placed on the board are placed relative to this.
- Add events and drawing code to update the ***GameBoard*** and ***GameBlock*** to highlight the block currently *hovered* over
- Create a new **fadeOut** method on the ***GameBlock***
  - By implementing an AnimationTimer, use this to flash and then fades out to indicate a cleared block
  - Tip: You could do this by painting the block empty, then filling them in with a semi-transparent (gradually getting more transparent) fill each frame
- Create a new **fadeOut** method on the ***GameBoard*** which takes a ***Set*** of ***GameBlockCoordinates*** and triggers the **fadeOut** method for each block
- Create a **LineClearedListener** which takes a ***Set*** of ***GameBlockCoordinates*** (that hold an x and y in the grid of blocks cleared) and add it to the **Game** class to trigger when lines are cleared.

- Use the **LineClearedListener** in the ***ChallengeScene*** to receive blocks cleared from the ***Game*** and pass them to fade out to the ***GameBoard***

Game Loop

Next we need to handle the game progression, which is done via the timer – when no piece is played, we move on to the next piece. We need to:

- Add a timer to count down how long there is until the piece must be placed
- When the timer runs out, move on to the next piece and lose a life
- Show the timer in the game UI

To do this:

- Add a **getTimerDelay** function in ***Game***
  - Calculate the delay at the maximum of either 2500 milliseconds or 12000 – 500 * the current level
  - So it'll start at 12000, then drop to 11500, then 11000 and keep on going until it reaches 2500 at which point it won't drop any lower
- Implement a ***Timer*** or ***ScheduledExecutorService*** inside the ***Game*** class which calls a **gameLoop** method
  - This should be started when the game starts and repeat at the interval specified by the **getTimerDelay** function
  - When **gameLoop** fires (the timer reaches 0): lose a life, the current piece is discarded and the timer restarts. The multiplier is set back to 1.
  - The timer should be **reset** when a piece is played, to the new timer delay (which may have changed)
- Create a **GameLoopListener**
  - and a setOnGameLoop method to link it to a listener
  - Use the GameLoopListener to link the timer inside the game with the UI timer
- Create and add an animated timer bar to the ***ChallengeScene***.
  - Use Transitions or an AnimationTimer to implement the timer bar
  - The ChallengeScene should use the GameLoopListener to listen on the GameLoop starting and reset the bar and animation
  - The timer bar should change colour to indicate urgency.
  - Tip: You may want to use a Timeline or create a Transition to animate the bar, which provides a smooth animation using interpolation
  - Tip: The demo uses a Rectangle
- When the number of lives goes below 0, the game should end.

Scores

Now we've got the game working, we need a reason to play the game – to get a high score! We'll add a new Scores screen:

- Create a new Scores Screen
- Save and read high scores to a scores file
- Prompt for a name on getting a high score

To do this:

- Create a new ***ScoresScene***
  - Add the relevant method to start it into GameWindow
  - Switch to it when the game ends.

    - You should pass through the Game object, containing the final game state
- Set up an observable list property to hold the scores
    - Add a **localScores** ***SimpleListProperty*** to hold the current list of scores in the Scene
    - Use **Pair**<String,Integer> to represent a score
    - Create an ArrayList to hold those Pairs
    - Use **FXCollections.*observableArrayList*** *to make an observable list* out of that ArrayList
    - Create a SimpleListProperty as a wrapper around the list, exposing it as a property
- Create a new **ScoresList** custom UI component and add it to the ***ScoresScene***, which will hold and display a list of names and associated scores
    - Use a ***SimpleListProperty*** inside the ScoresList
    - Update the scores when the list is updated
    - Bind the ScoresList scores to the ScoresScene scores list
    - Add a **reveal** method which animates the display of the scores
    - Tip: You are binding the ScoresList UI component to an observable list property. When the list property is updated, the scores list UI should update it's display of those scores
- Write a **loadScores** method to load a set of high scores from a file and populate an ordered list
    - A simple format of newline separated name:score will suffice
    - Update the ScoresScene score list with the loaded scores (which will update the ScoresList)
- Write a **writeScores** method to write an ordered list of scores into a file, using the same format as above.
- If the scores file does not exist, write a default list of scores to the file.
- When the ***ScoresScene*** starts
    - Load the ordered list list of scores and link the scores to the ScoreList
    - If the score contained inside the ***Game*** beats any of the scores, prompt the user for their name, insert it into the list at the correct position, display the scores and update the saved file.
    - Reveal the high scores
- Add a **getHighScore** method to the ***ChallengeScene***
    - Get the top high score when starting a game in the ChallengeScene and display it in the UI
    - If the user exceeds it, increase the high score with the users high score.

Online Scores

Next, let's integrate with the online server and compare our scores with other people:

- Receive online scores from the server
- Include these in the Score Screen
- If the new score beats the current scores, submit it to the server

To do this:

- Add a **remoteScores** SimpleListProperty in the Scene
- Add **loadOnlineScores** method to ScoresScene
    - Use the ***Communicator*** to request HISCORES when entering the ScoresScene
- Use a ***CommunicationsListener*** within the ***ScoresScene*** to parse the high scores and create a second ordered list of online high scores.
    - Update the **remoteScores** list
    - You will need to ensure you only trigger displaying the high scores and checking the high scores occurs after the scores list arrives!
- Add a **writeOnlineScore** method
    - Use the ***Communicator*** to submit a new HISCORE if the user has beaten the previous high scores, as well as inserting their score into the list.

- Add another ***ScoresList*** component bound to the remoteScores property to display the online scores list alongside the local high scores list.

Network Protocol

The Network Protocol that you need is:

- Send: **HISCORES**
  - Receive: **HISCORES <Name>:<Score>\n<Name>:<Score>\n...**
  - Description: Get the top high scores list
- Send: **HISCORES UNIQUE**
  - Receive: **HISCORES <Name>:<Score>\n<Name>:<Score>\n...**
  - Description: Only include each unique player name for a more varied high score list
- Send: **HISCORES DEFAULT**
  - Receive: **HISCORES <Name>:<Score>\n<Name>:<Score>\n...**
  - Description: Include a default high score list. Good for testing
- Send: **HISCORE <Name>:<Score>**
  - Description: Submit a new high score. Do not cheat – we will know!
  - Receive: **NEWSCORE <Name>:<Score>**

Testing

We suggest you test by calling HISCORES DEFAULT initially and then sending a HISCORE and ensuring you receive a NEWSCORE response

Multiplayer

Now, let's take the multiplayer aspect a little further by adding multiplayer play itself:

- Implement the Lobby System
  - Find all games
  - Create a new Game
  - Join a game that exists already
- Chat
- Implement the Multiplayer Gameplay
  - Create a leaderboard against the people you're playing with
  - Send and receive blocks from the server
  - Send and receive game updates

The Lobby

First, we want to implement a lobby system to browse, jon and create games:

- Create a Multiplayer menu option and create a corresponding **LobbyScene**.
  - The LobbyScene on opening should start a repeating timer requesting current channels using the Communicator from the server.
  - Add a listener to the Communicator to handle incoming messages
  - You will need to handle the different commands that are receiveved
  - When channels are received, the LobbyScene UI should be updated to show available channels.
- Add a button to start a new channel,
  - Prompt the user for a channel name
  - Submit it using the Communicator
- Allow the user to JOIN a channel by sending the corresponding command to the Communicator.
  - Update the UI to show the channel and the users currently inside that channel.

- Add a chat box to allow sending and receiving chat messages with people in that channel.
- Add buttons to start the game (if the user is the host) and leave the channel
  - Send appropriate messages when these buttons are clicked to the server
- Add error handling on receiving ERROR messages from the server.

Network Protocol

To complete this section, we expect you to have completed all the protocol below

- Note: Game names must be unique. The server will not allow creating a game with the same name that already exists.

At any time:

- Send: **LIST**
  - Description: Get a list of all current channels
  - Receive: **CHANNELS <Channel>\n<Channel>\n<Channel>...**
- Send: **CREATE <Channel>**
  - Receive: **JOIN <Channel>**
  - Description: Create a new channel
- Send: **JOIN <Channel>**
  - Receive: **JOIN <Channel>**
  - Description: Request to join the given channel, if not already in a channel
- Send: **QUIT**
  - Description: Disconnect from the server. No further communication will be possible. Send when exiting
- Receive: **ERROR <Message>**
  - Description: Received if an action was not possible

When in a channel:

- Send: **MSG <Text>**
  - Description: Send a message to the channel
- Send: **NICK <NewName>**
  - Description: Change nickname. The new nickname, if successful, will be returned. This may be different to the submitted nickname (e.g. special characters removed).
  - Receive: **NICK <NewName>**
- Send: **START**
  - Description: Request to start a game if host
  - Receive: **START**
- Send: **PART**
  - Description: Leave a channel
  - Receive: **PARTED**
- Send: **USERS**
  - Description: Request the list of users in the channel
- Receive: **USERS <User>\n<User>\n<User>...**
  - Description: User list in channel
- Receive: **NICK <Name>:<NewName>**
  - Description: Name has changed to NewName. You can safely split on the : as this is not allowed in usernames.
- Receive: **START**
  - Description: The game is starting
- Receive: **MSG <Player>:<Message>**
  - Description: Received a chat message from the player

- Receive: **HOST**
  - Description: You are the host of this channel and can start a game

The Multiplayer Game

Now we've got the lobby, the actual multiplayer game aspect itself isn't much different – the main things are getting the pieces from the server and updating other players with our state and receiving their state updates – primarily scores.

To do this:

- Create a custom **Leaderboard** component which extends the **ScoreList**
- Create a **MultiplayerScene** which extends the ***ChallengeScene***.
- Update the UI to include chat and the leaderboard and remove less important elements if needed.
- Create a **MultiplayerGame** which extends the ***Game***.
  - Using a queue or similar, request pieces as needed from the server and use these to populate the current and upcoming pieces.
  - The server will give every player the same ordered stream of pieces, so every player has the same gameplay
  - Piece order: If the server sends Line, Square, X, Line, then they should have Line, Square, X, Line in that order
- The **MultiplayerGame** should listen for incoming pieces, score updates and chat messages.
  - When score updates are received, update the list which should be bound to the leaderboard and update automatically.
- Send appropriate updates using the **Communicator** on board changes, score changes, live changes, getting a game over and leaving the game.
- Update the leaderboard to track number of lives and when people get a game over, crossing them off as they are eliminated.
- Update the **ScoresScene** to substitute showing local scores for the multiplayer game scores, held in the ***MultplayerGame*** object
- If the player runs out of lives, or aborts, make sure you send the DIE message

The server will give the **same ordered stream of pieces for every player**. Every player should get those pieces in that order. If you are using a queue, they should be queued in the order from the server.

To complete this section, we expect you to have completed all the protocol below, with the exception of processing incoming BOARD messages, which is an extension.

Network Protocol

- Send: **SCORE <Score>**
  - Description: Update player current score when changed
  - Receive: SCORE <Player>:<Score>
- Send: **SCORES**
  - Description: Request current scores from all players
  - Receive: **SCORES <Player>:<Score>:<Lives|DEAD>\n<Player>:<Score>:<Lives|DEAD>\n<Player>:<Score>:<Lives|DEAD>\n...**
- Send: **PIECE**
  - Description: Request the next piece
  - Receive: **PIECE <Value>**
- Send: **BOARD <Value of Block 0,0> <Value of Block 0,1> <Value of Block 0,2>... <Value of Block columns,rows>**
  - This is a space separated list of block values, iterated over columns (0,1 to 0,rows then 1,0 to 1,rows up to columns,rows)
  - Description: Update player current board when changed

    - Note: Sending this is important as it is used to verify against anti-cheating
- Send: **LIVES <New Number>**
    - Description: Update the number of lives player has
- Send: **DIE**
    - Description: This player has run out of lives and got a game over, or has chosen to quit the game
    - This will also cause the player to leave the channel - you do not need to **PART** separately.

- Receive: **BOARD <Player>:<Value of Block 0,0> <Value of Block 0,1> <Value of Block 0,2>... <Value of Block columns,rows>** (Handling receiving this is an extension)
    - Format: Space separated list of block values, iterated over columns (0,1 to 0,rows then 1,0 to 1,rows up to columns,rows)
    - Description: Update the board of a particular player
- Receive: SCORE <Player>:<Score>
    - Description: A player has updated their score
- Receive: **SCORES <Player>:<Score>:<Lives|DEAD>\n<Player>:<Score>:<Lives|DEAD>\n<Player>:<Score>:<Lives|DEAD>\n...**
    - Description: Get a current status update on all players, their scores and their number of lives (or DEAD if they've game overed)
- Receive: **MSG <Player>:<Message>**
    - Description: Received a chat message from the player

Extensions

There are up to 5 marks available for extensions — you can do a small number or even one extension well to get all 5 marks, or work through multiple extensions. Marks will be awarded based on the level of challenge or number of extensions and the technical achievement.

- Add further polish, animations and effects to the user interface
- Add statistics to the ScoreScene and save and retrieve them between games
- Allow customising the single player challenge with different options
- Add a settings menu to change volume, resolution and other settings. Store and retrieve this from a config file
- Add custom theme support to change the appearance, sounds and music
- Display other peoples boards in multi player
- Implement powerups in single player
- Implement a duel mode where you can play against a bot
- Implement your own compatible multiplayer server in Java
- Introduce your own new server-side multiplayer features
- **Anything else you think will impress us!**
- **Have fun!**

If you need any changes on the server to support your extensions, get in touch - requests are accepted.

Marking

This coursework is worth 40% of the module mark, across 80 marks:

- The Basics: 2 marks
- Creating the Game Logic: 8 marks
- Building the User Interface: 6 marks
- Enhancing the User Interface: 8 marks
- Adding Events: 6 marks

- Adding Graphics: 7 marks
- Adding a Game Loop: 6 marks
- Adding Scores: 6 marks
- Adding an Online Scoreboard: 4 marks
- Adding Multiplayer: 12 marks
- Extensions: 5 marks
- Code Quality: 5 marks
- Understanding: 5 marks

The coursework will be marked objectively – for each mark, either the coursework performs the function (full marks), partially performs the function (half marks) or doesn't perform the function (no marks), using the following marking scheme:

Category	Requirement	Mark
Basics	Does it compile with mvn compile?	1
Basics	Does it run with mvn javafx:run?	1
Creating the Game Logic	Play a piece in a valid location	2
Creating the Game Logic	Reject a piece in an invalid location	2
Creating the Game Logic	A new random piece each time	2
Creating the Game Logic	Lines are cleared	2
Building the User Interface	Score, Level, Lives shown in the UI	2
Building the User Interface	Scoring works	1
Building the User Interface	Multiplier works	1
Building the User Interface	Level works	1
Building the User Interface	Background music	1
Enhancing the User Interface	Menu screen present and functioning	1
Enhancing the User Interface	Menu screen uses animations and visual effects	1
Enhancing the User Interface	Menu screen works: launches other scenes	1
Enhancing the User Interface	Instructions screen	1
Enhancing the User Interface	Instructions screen shows dynamically generated blocks	1
Enhancing the User Interface	Challenge screen shows current piece	1
Enhancing the User Interface	Current piece updates on play	1
Enhancing the User Interface	Escape navigates back	1
Adding Events	Challenge screen shows next piece	1
Adding Events	Piece can be rotated	1
Adding Events	Piece can be swapped	1

Adding Events	Sound effects added	1
Adding Events	Keyboard support for move and drop	2
Adding Graphics	Blocks have updated graphics and are not just a single colour square	1
Adding Graphics	Placement indicator (e.g. circle) shown on current piece	1
Adding Graphics	Blocks are placed on their centre	1
Adding Graphics	Hover effect on blocks	2
Adding Graphics	Fade out effect when blocks cleared	2
Adding a Game Loop	Countdown shown in UI	1
Adding a Game Loop	Countdown animates	1
Adding a Game Loop	Countdown changes when time nearly out	1
Adding a Game Loop	Life lost and new piece when timer expires	2
Adding a Game Loop	Game over when no more lives left	1
Adding Scores	Game goes to Scores screen on Game Over	1
Adding Scores	Scores read from file	1
Adding Scores	Scores saved to file	1
Adding Scores	New score prompts for name	1
Adding Scores	New score inserted into score list	1
Adding Scores	High Score to beat shown in Challenge screen	1
Adding an Online Scoreboard	Online scores shown in Score screen	4
Adding Multiplayer	Lobby updates games as new ones created	1
Adding Multiplayer	Can create a game	1
Adding Multiplayer	Can join a game	1
Adding Multiplayer	Can send a chat message	1
Adding Multiplayer	Can receive a chat message	1
Adding Multiplayer	Can start a game (if host)	1
Adding Multiplayer	Game starts when started by another player (when not host)	1
Adding Multiplayer	Error states handled	1
Adding Multiplayer	Can play a multiplayer game with a friend/second client	1
Adding Multiplayer	Scores update during multiplayer game	1
Adding Multiplayer	Eliminated players shown during multiplayer game	1

Adding Multiplayer	Scores shows multiplayer game scores	1
Extensions	Good level of challenge, number of extensions, technical achievement	5
Code Quality	Full javadoc must be generated with mvn javadoc:javadoc. Classes and comments are commented with javadoc and inline comments, good use of logging and code is of good quality and structure	5
Understanding	Clear understanding of code and application demonstrated in video	5

FAQ

Can we modify the skeleton?

Answer: Please do! It's there to get you started – change it as you wish! Remove/replace/modify methods to your hearts content...

Can we use our own artwork, music, styles, gameplay ideas etc.?

Answer: Yes, please do! We provide assets but you don't have to use them, and we love to see what you come up with. Just make sure you match the mark scheme!

Should we use the SceneBuilder to help build our GUI?

Answer: No! This will bring you a world of pain – A game is really not the right place to use a GUI builder with. You can mock up things in it and translate that into code, but you really don't want this as FXML, which is intended for simpler dialog boxes and windowed applications, not graphically intensive games.

Do we need to use the exact class names and methods suggested? Are you using automatic test harnesses

Answer: We are not using automated testing so you do not need to follow class or method names or follow the suggested implementation. This is why we ask you to demonstrate your game in submission, so we can see it in action and mark it against the objectives.

Do we need Javadoc/Comments

Answer: Yes – you must have Javadoc comments on all classes and functions and inline comments to explain your code within functions

Can we create additional folders/packages?

Answer: Yes – this is fine, as long as it is possible to fully build your project with maven from your submission

Why don't I want the GameBoard (UI) inside my Game (Model)?

The reason we use things like listeners is to avoid tight coupling of classes – e.g. where a class is dependent on another and can't exist without it.

Ideally, you want classes to be responsible for their thing, and interactions to happen loosely.

The Game class should be responsible for the game and game state – it should have no cares about the UI The GameBoard should be responsible for displaying a grid on the UI – it should have no cares about the game itself The ChallengeScene is responsible for linking them together, receiving events, updating the game, which updates the game board.

These are all in separate packages too

Properties and listeners allow you to link all this together but without any of the classes having to reference each other inside the class itself.

Imagine that we wanted to make a web-based version of the game. If your Game class is completely independent of the GameBoard, it'd be simple – we'd make a GamePage or similar class responsible for the web UI and a WebGameBoard or similar, but be able to reuse the Game class /exactly/ as it is – we'd just bind to it and listen to it in a different way.

If you've hard coded the GameBoard into the Game and accessing it directly, you can't do that anymore – you've tied it to JavaFX, you've tied it to one GameBoard implementation, you've required JavaFX to be a dependency which means you can make a web app and you've removed that single job, single responsibility of the Game and munged it with the UI = bad

If I said now, when the game starts up, you should be able to specify on the command line to run a command line version of it with a TUI or the normal JavaFX version, you should only have to modify the UI code and not touch a single bit of the workings of the game. If you would have to, you've probably coupled them together badly.

Can we re-use code from ECSChat, including code we didn't write?

Answer: Beyond the skeleton which was provided, you should be writing new code for TetrECS to avoid any issues of AI

My sounds or music stop or cut short?

This happens if your MediaPlayer is a local variable in a method, which gets garbage collected when the method ends, before it has finished playing. Make sure it is a field in the class or exists outside of the method itself. Typically, we add MediaPlayer static fields into the Utility class so they won't be garbage collected and we can easily access them to stop them at a later point.

Can I have an example of how to use an ObservableList properly?

Here is a simple example you should be able to reproduce yourself:

```
        //We create an ArrayList to hold the scores - we can fill it with things if we want
        ArrayList<String> names = new ArrayList<>();

        //We create an ObservableArrayList around this list
        ObservableList<String> namesList = FXCollections.observableArrayList(names);

        //We create a SimpleListProperty which represents the ObservableArrayList as a property
 (and allows binding and listeners)
        ListProperty<String> wrapper = new SimpleListProperty<String>(namesList);

        wrapper.addListener(new ListChangeListener<String>() {
            @Override
            public void onChanged(Change<? extends String> change) {
                System.out.println("This element changed: " + change);
            }
        });
        wrapper.add("Hello");
```

Submission

- You will be required to submit a fully working **Maven project** (tetrecs.zip)
  - **It must be possible for us to compile and build your project using the pom.xml file**
  - This should as a minimum contain the pom.xml and src folder and in the same structure as the skeleton was provided in
  - If you want to test your submission, unzip your tetrecs.zip file somewhere else on your computer and ensure you can mvn clean compile javafx:run it
- Your code must be fully Javadoc documented – we will be generating this and verifying this with Maven mvn javadoc:javadoc

- You will be required to demonstrate your game in the lab session to a demonstrator, who will mark your functionality

- You will also be required to submit a **5 minute video** walkthrough of your game showing it meeting all the criteria provided (tetrecs.mp4)
  - You should capture your footage and then *edit* the video to 5 minutes to form a demonstration of the marking points *in order*
  - Any videos longer than 5 minutes will be penalised!
  - This must be submitted in MP4 format
  - The suggested free, cross-platform tool to prepare your video is OBS (https://obsproject.com/)
  - You must walk through the mark scheme **in order**, demonstrating your game running to meet each requirement
  - You should narrate the video and/or have text on the video to make it clear what mark(s) of the mark scheme are being shown
  - The video MUST be submitted in addition to the in-lab demonstration – it will be used to moderate the marking, to handle any issues or problems, and to ensure people can be marked if they are unable to attend due to illness or other reasons.

- You will be required to submit a **self-assessment mark sheet** (assessment.txt)
  - Mark off what you have and have not completed. Everything completed must be demonstrated in the video
  - Other than ticking the boxes, you must not make any other changes to this file or you will risk losing marks

- Deadline: 28/04/2022 (after Easter) – Thursday

Marking Process

The handin for TetrECS is now live here: https://handin.ecs.soton.ac.uk/handin/2122/COMP1206/7/

When you submit, you should get an email to tell you everything is fine, or that your code doesn't compile, or you didn't fill out your assessment.txt correctly. In the case of the latter two, make sure to correct that and resubmit, or you won't be marked and will receive 0.

You must then attend a marking session to get your coursework marked – the coursework is not marked automatically, but by demonstrators in the lab sessions. You can choose either:

- **Marking Session 1**: Tuesday 26th April (9am or 11am, depending on your timetabled slot)
- **Marking Session 2**: Tuesday 3rd May (9am or 11am, depending on your timetabled slot)

I would strongly recommend you attend Marking Session 1 – this slot will be quieter, you will have more time to talk with your demonstrator and show what you've made, and if you hit any significant issues or problems, it is before the deadline, so you will have time to correct them!

In these marking sessions, you must come in and get your application ready and running, ready to demonstrate to the marker. You'll then need to walk them through everything you have done so they can tick off the mark scheme (https://secure.ecs.soton.ac.uk/noteswiki/w/COMP1206/Coursework#Marking) – which you already have, so hopefully you already know what your mark will likely be!

You must attend the slot you are assigned to on your timetable so 9am or 11am! If you attend the wrong slot, you will not be marked in that session!

It is **your responsibility** to get your coursework marked – if you do not get it marked, you will receive a 0 for the coursework. If you are unable to make it to the marking sessions, you must submit a [special considerations request https://www.southampton.ac.uk/quality/assessment/special_considerations.page] and let us know as soon as possible. If such arrangements need to be put into place, they must be sorted **before the deadline**.

Support

Do not panic! There is lots of support and help available for you to help you through this coursework – we are here to help! Support is available:

- Use **HELPDESK** — 1:1 support for your coursework!
  - On Discord in the #helpdesk channel (open 10am to 12pm and 2pm to 4pm daily)
  - Or email helpdesk@ecs.soton.ac.uk
- Use the Discord #coursework channel
  - We are here to help!
- We will run a coursework tutorial and help session before the handin

Debugging

With IntelliJ

A quick way is if you've built it already with Maven, you can just right click a class method (e.g. main) and start the debugger

An alternative way is set up a Remote Debugger in IntelliJ on port 8000

Then add the following config in your pom.xml:

```
<plugin>
   <groupId>org.openjfx</groupId>
   <artifactId>javafx-maven-plugin</artifactId>
   <version>0.0.8</version>
   ...
   <executions>
       <execution>
           <id>debug</id>
           <configuration>
               <mainClass>uk.ac.soton.comp1206/uk.ac.soton.comp1206.App</mainClass>
               <options>
                   <option>-
agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:8000</option>
               </options>
           </configuration>
       </execution>
   </executions>
</plugin>
```

Then in IntelliJ in the terminal, do mvn javafx:run@debug (or create a specific run configuration for this)

Then run your remote debug setup

Retrieved from "https://secure.ecs.soton.ac.uk/noteswiki/index.php?title=COMP1206/Coursework&oldid=42595"

- This page was last edited on 28 April 2022, at 08:34.