CODE NINJAS®

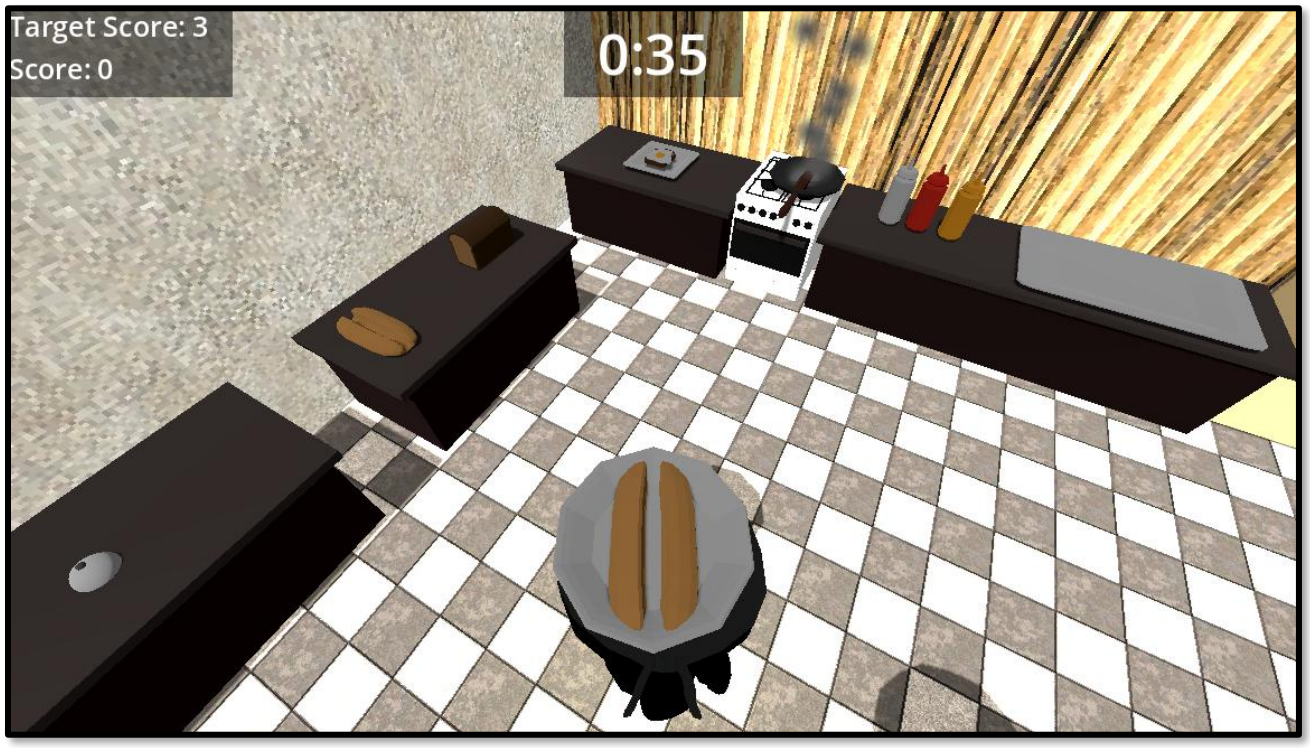
Platinum Belt Ninja Guide

PB Activity 03: Chef Codey

Contents

PB Activity 03: Chef Codey

In this project, you will design a cooking game where the player interacts with objects to serve dishes. You will work with inheritance, signals, UI, and autoloads.



Pro Tips

This project contains many requirements and fewer instructional steps, which may not feel as detailed as you are used to. The requirements should feel familiar and most of the instructional steps contain familiar concepts applied in a slightly new way. This project is set up to feel more like a Quest in IMPACT.

Here are a few problem-solving strategies that you can use if you feel stuck while working through the requirements and instructions:

- Take things one step at a time and break down each instruction or checkbox into manageable chunks.
- Write pseudo code, either with a pen and paper or with code comments in the script.
- Use lots of print statements to see which functions are being called when, and which aren't, or if a variable is updating as expected.
- Tinker with values and playtest often. Don't be afraid to test out code, see if it works and adjust if it doesn't.
- Use the hints as needed, whether you're feeling stuck or to confirm you're on the right track.
- Refer to previous projects or documentation. This will be included in the Resources section for each requirement, but you can refer to any additional projects or documentation that you feel may be helpful.
- Check in with Code Sensei if you're still feeling unsure about a step or a requirement.

Requirement #1: Project Setup

Set up the project and the main scene.

- ☐ Import the Ninja Starter Pack as a new project in the correct installation path.
- ☐ Rename the project to **[YourInitials]ChefCodey** and open the project.
- ☐ Create a new **3D** scene named **chef_codey.tscn**, save the new scene in the **Scenes** folder, and set the main scene to **chef_codey.tscn**.

Requirement #1: Hints

- Which button imports files as a new project in the Project Manager?
- Where is the starter code stored? What file type is the starter pack?

Requirement #1: Resources

- Refer to Activities 06 – 17 in SB for help with importing a project.

Requirement #2: Project Planning

Locate the **project planning documents** with your Code Sensei.

Look through the Food and Item assets, then use the project planning documents to plan the café layout and dishes Codey will cook.

- ☐ Look through the food items in the **Assets > Food** folder and plan the preparation process for the 2 dishes Codey will cook. Think about the individual food scenes that will be needed at the different stages in the preparation process. If the Assets> Food folder is missing an item, check if it can be created by combining other meshes within the folder.
  - Both items must have one ingredient that comes from the fridge.
  - Both items must contain something to cook on the stove.

Example:

Write down the 2 dishes Codey will make and their preparation order below.

Dish #1: Egg Toast

Whole Egg	→	Cooked Egg	→ Bread Slice →
Egg Toast	→		→

Dish #2: Hotdog with Condiments

Uncooked Sausage	→	Cooked Sausage	→ Hotdog Bun →
Hotdog	→	Hotdog with Condiments	→

- Look through the **Assets > Items** folder and plan the café layout. The café must contain the following stations:
    - Fridge
    - Stove
    - Trash
    - Somewhere to plate completed dishes
    - An end point (such as a bell) for Codey to interact with when a dish is plated and ready to be served.
      Note: There is a pre-made bell asset in the Scenes > Interactables folder.
- Plan the **Stove Station** and its cooking process. Codey must be holding an item to pick something up off the stove. This could be a food item or a special dish (like a bowl or a fancy plate).
  Example:

Review the example filled in below, then use the blank lines to plan out the cooking process.

Stove Input	Stove Item	Pickup Item	Stove Output
Whole Egg	Cooked Egg	Bread Slice	Egg Toast
Uncooked Sausage	Cooked Sausage	Hotdog Bun	Hotdog

- Plan the **Interactable Stations** and which items Codey will pick up or put down at each station. Include all stations Codey will interact with, such as the Trash and Bell. The Stove and its items do not need to be included.
  Example:

Station	Pick Up	Put Down
Fridge	Whole Egg	Nothing
Fridge	Uncooked Sausage	Nothing
Bread Counter	Bread Slice	Nothing
Bread Counter	Hotdog Bun	Nothing
Egg Toast Plate	Nothing	Egg Toast
Condiments Counter	Hotdog with Condiments	Nothing
Hotdog Plate	Nothing	Hotdog with Condiments
Trash	Nothing	Anything
Bell	Interact with when dish is complete and plated.	

- Return to Codey's Café layout and label the plan with all the interactable stations.

Pause for **Ninja Stop #1**!

Are all aspects of the planning sheet complete?

Reminder: Save your work!

Requirement #3: Create the Café

Set up the café with a floor, walls, a world environment, directional lighting and the player.

- ☐ Create a **floor.tscn** scene and a **wall.tscn** scene in the **Scenes > Objects** folder with a **StaticBody3D** as the root node for each scene.
  - Complete the scenes by adding the nodes needed to add a mesh to the objects and to prevent Codey from passing through the floor/wall. **Note:** 2D textures can be found in the **Assets > Textures** folder.
- ☐ In the **chef_codey.tscn** scene, use the **floor.tscn** and **wall.tscn** scenes to build a café with 4 walls for Codey to cook in. Add a **Node3D** renamed to **Room** as a child to **ChefCodey** and use it to organize the nodes.
  Note: The café can include 4 walls with the same texture, or multiple textures. If using multiple textures, multiple wall scenes are needed.
- ☐ Add a **WorldEnvironment** node as a child to **ChefCodey**. Create a **New Environment** and set its **Ambient Light Source** to **white**.
  Note: The hex code for white is #FFFFFF.
- ☐ Add a **DirectionalLight3D** node as a child to **ChefCodey**. Adjust the **position** and **rotation** of the light as needed. **Enable shadows** by checking **Shadow** > **Enabled** to **On**.
- ☐ Find the **player.tscn** scene in the **Scenes > Objects** folder. Add the player to the scene as a child of **ChefCodey**.

Pause for **Ninja Stop #2**!

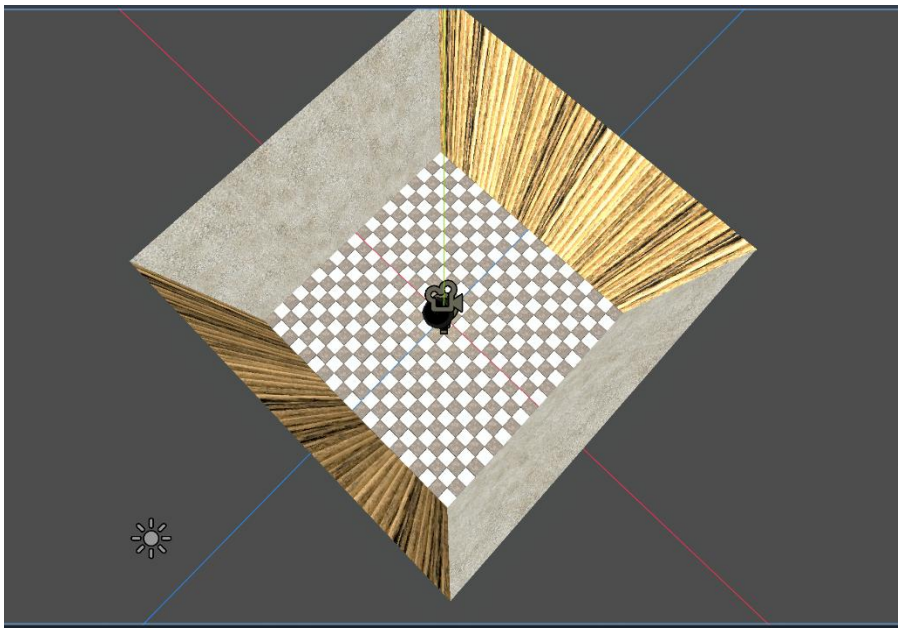
Test your project! Does it have...

- A WorldEnvironment node?
- A DirectionalLight3D node?
- A player?
- A room for the café?

Reminder: Save your work!

Requirement #3: Hints

- ☐ Do the floor and wall scenes have a **MeshInstance3D** and **CollisionShape3D** as children to the StaticBody3D root node?
- ☐ In the **Inspector** for **WorldEnvironment**, is **Environment** set to a **New Environment**?
- ☐ In **Environment** under **Ambient Light**, is **Source** set to **Color** and **Color** set to white?
- ☐ In the **Inspector** for **DirectionalLight3D**, is **Enabled** checked **On** under **Shadow**?



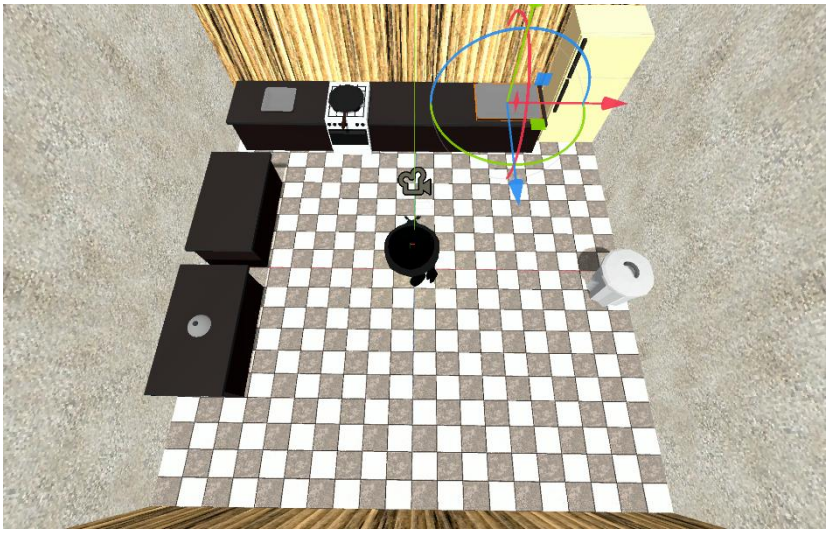
Requirement #3: Resources

- ☐ Refer to **BB Activity 05: Don't Touch the Cubes** for help with **WorldEnvironment** and **DirectionalLight3D** nodes.
- ☐ Refer to Requirement #6 in **PB Activity 02: Codey Raceway** for help with setting up the room's MeshInstance3D nodes.

Requirement #4: Set up the Café

Add objects to the café for Codey to interact with.

- Create **stove.tscn**, **fridge.tscn** and **trash_can.tscn** scenes in the **Scenes > Objects > Interactables** folder with a **StaticBody3D** as the root node for each scene.
    - Complete the scenes by adding the nodes needed to add a mesh to the objects and to prevent Codey from passing through the objects.
      Note: Models can be found in the **Assets > Items** folder.



- In the **chef_codey.tscn** scene, add the stove, fridge and trashcan to the scene. Add a **Node3D** renamed to **Objects** as a child to **ChefCodey** and use the node to organize the objects in the café.
- Customize the scene with additional objects, like plates, counters, lights, etc. Look through the models in the items folder to see what assets can be used to customize the scene.
  Note: Refer to the project planning documents as needed.
    - Create scenes for the additional objects as needed and organize them in the scene under the **Objects** node.
      Note: Not all objects will need to be created as a scene. Items used for decoration, such as a coffee machine or a pan, can be added directly to the **chef_codey.tscn** scene as a mesh. Make sure the node is given a name and parented as needed under the **Objects** node.
- Add the **bell.tscn** scene from the **Objects > Scenes > Interactables** folder to the **chef_codey.tscn** scene.
    - Alternatively, create and add a different object for Codey to interact with when a dish is plated and ready to be served.

Pause for **Ninja Stop #3**!

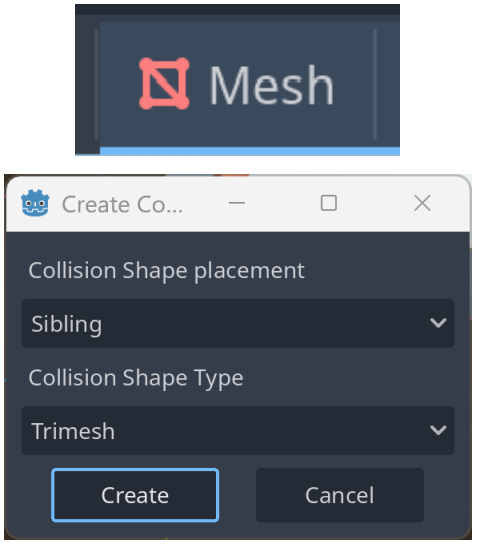
Test your project!

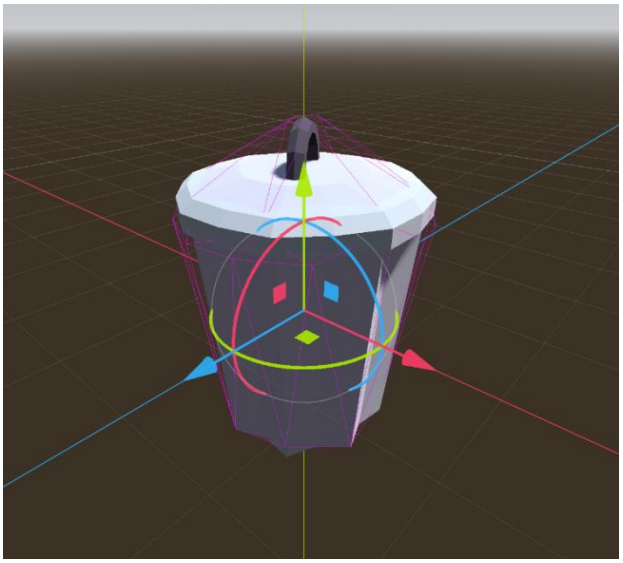
- Can Codey move through any objects (stove, fridge, etc.)?
- Are the objects organized under an Objects node?

Reminder: Save your work!

Requirement #4: Hints

- [ ] Were the CollisionShape3D nodes for the objects created as a **Trimesh Collision Shape** from the **MeshInstance3D**?





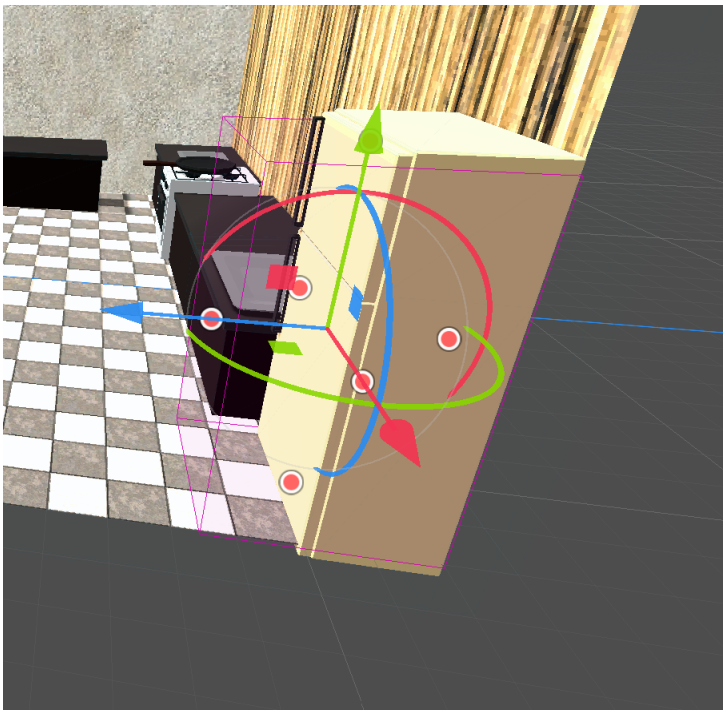
Requirement #4: Resources

- [ ] Refer to platform creation in **BB Activity 04: Sketch Head**
- [ ] Godot 4.4 StaticBody3D Documentation

Requirement #5: Set up the Interactable Stations

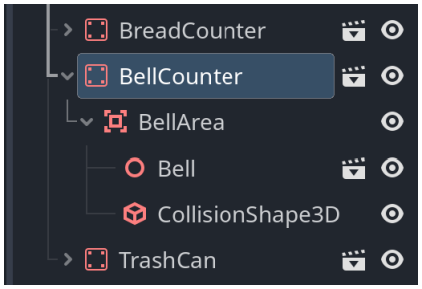
Return to the **project planning documents** and review the **Interactable Stations** section. Codey will need to interact with all these stations.

- [ ] In the **chef_codey.tscn** scene, add the nodes needed to each station to check when Codey is in the object's vicinity. Rename the nodes as needed to keep the game organized.



Requirement #5: Hints

- [ ] Does each Interactable Station have an **Area3D** and **CollisionShape3D** in the **chef_codey.tscn** scene?



- [ ] Is the size of each collision shape large enough for Codey to interact with?
- [ ] Is there any overlap between collision shapes?

Requirement #5: Resources

- [ ] Godot 4.4 Area3D Documentation

Pause for **Sensei Stop #1**!

Check with a Code Sensei and confirm that requirements 1 – 5 are set up correctly.

Reminder: Save your work!

Requirement #6 (Instructions): Code the Interactable Stations

The Player is responsible for... • ~~User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... • Defining enums. • Storing the game state.
All Interactables are responsible for... • Displaying a tooltip when the Player gets close. • Calling interact() when the Player is close and presses the "interact" action.
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... • Defining game constants. • Providing helper functions that are used throughout many scripts.
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #6 (Instructions): Code the Interactable Stations

1. Create a new script **globals.gd** and save the script in the **Scripts** folder. Create an `enum InteractableStation` for the interactable objects in the **chef_codey.tscn** scene. The fridge, stove, trash, and bell objects, as well as any other stations Codey interacts with during the cooking process need to be included in this enum.

2. Set the **globals.gd script** to a **Global variable, Globals**.

   Then, create a new script called **interactable.gd** and save it in a **Scripts > Interactable** folder.

3. Update the code in the **interactable.gd** script to create a `class` named `Interactable` that `extends` the `Area3D` class.

4. Create an `@export` variable `cur_interactable` of type `InteractableStation`. This variable will track what kind of station Codey is interacting with.

   Attach the **interactable.gd** script to all the interactable items in the scene. Set a value for **Cur Interactable** in the **Inspector** for each station.

5. Create a function `get_interactable_prompt()` that takes a parameter `interact_object` of type `InteractableStation` and returns a `String`. The function should check the interactable station and return the name of the station. A match statement may be helpful here.

```
1   class_name Interactable extends Area3D
2
3   @export var cur_interactable: Globals.InteractableStation
4
5   func get_interactable_prompt(interact_obj: Globals.InteractableStation) -> String:
6       # match ???:
7           # ???:
8               # return ???
9           # ???:
10              # return ???
11          # ....
12      # return ???
13
```

6. Create an `on_area_entered()` method to call `get_interactable_prompt()` with the argument `cur_interactable`. Print the string returned by the function to the console to see which item is being interacted with.
7. Connect the `area_entered` signal **with code** to the `on_area_entered()` method.

8 Playtest the game. Does anything print to the console immediately after the game starts, but before Codey has had a chance to interact with any of the stations? If so, stop the playtest and think about why this might happen.

Then, fix the bug.

Continue the playtest by moving Codey around the room. Do the correct strings print to the console when Codey interacts with the different stations?

9 Code the **interactable.gd** script so Codey can choose whether to interact with a station. In **interactable.gd**, create a **new variable** `can_interact` of **type** `bool` and set the variable to `false`.

Use the code completion tool to define the `_input()` method which takes a parameter `event` of type `InputEvent` and returns `void`. Inside the method, print a message to console if the `Input` action `"interact"` is pressed and `can_interact` is `true`.

The **interact event** is included in the **Input Map** as part of the starter code. Codey can interact with stations by pressing the E key.

10 Set `can_interact` to `true` inside the `on_area_entered()` function.

Then, playtest the project. What happens when the E key is pressed at the start of the game? What about when Codey enters a station's Area3D?

What happens when Codey exits a station's Area3D? Can the station still be interacted with?

11 Codey can still interact with a station after exiting its Area3D.

Update the code so Codey can't interact with a station when not inside its Area3D. A new function and signal will need to be created and connected.

12 Playtest the game and test Codey's interactions. Do all the interactions work as expected?

After playtesting, update the interactable prompts in the `get_interactable_prompt()` function to provide better game instruction.

Pause for **Ninja Stop #4**!

Test your project!

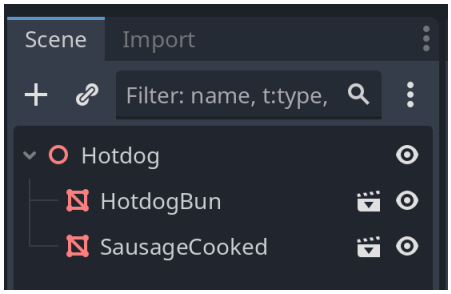
- Can Codey only interact with the stations when inside their Area3D?
- Are the correct prompts printing to the console for each station?

Reminder: Save your work!

Requirement #7: Create Food Items

Set up the food scenes for Codey to cook with. Each food item in the preparation process needs its own food scene. Refer to the **project planning** documents as needed.

- ☐ Create individual scenes for each food item and save each scene in a **Scenes > Objects > Food Items** folder.
  Note: Most food item scenes can use a **MeshInstance3D** as their root node. Food scenes can be created by combining multiple textures if needed.



- ☐ Add any additional food item decorations to the **chef_codey.tscn** scene.
- ☐ Codey will need a way to hold the food items. Add a dish to Codey's head in the **chef_codey.tscn** scene for food items to spawn onto. This will make it easy to see what Codey is holding when navigating the café.

Pause for **Ninja Stop #5**!

Check your project!

- Are all the necessary food scenes created?
- Were food items added to the café?
- Is there something on Codey's head to show what Codey is holding?

Reminder: Save your work!

Requirement #7: Hints

- Do all food items included in the preparation process have their own food scene?
- Are food items added to the scene to help the player navigate the café?
  Example: The bread station has bread items on it. This may hint to where the player can pick up bread items when cooking in the café.

Requirement #7: Resources

- Godot 4.4 MeshInstance3D documentation

Requirement #8 (Instructions): Code the Fridge

The Player is responsible for... • ~~User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... • Defining enums. • Storing the game state.
All Interactables are responsible for... • ~~Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... • Defining game constants. • Providing helper functions that are used throughout many scripts.
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #8 (Instructions): Code the Fridge

13 In the **globals.gd** script, create a new `enum Food` that contains the different food items and cooking stages. Include `NOTHING` at the top of the Food enum. This will help track the different food items and cooking stages each station can produce, or what Codey is holding.

```
12
13  enum Food {
14      NOTHING,
15      SAUSAGE_UNCOOKED,
16      EGG_WHOLE,
17      SAUSAGE_COOKED,
18      EGG_COOKED,
19      HOTDOG_BUN,
20      BREAD,
21      HOTDOG,
22      EGG_TOAST,
23      CONDIMENTS
24  }
```

14 Create a new script **food_spawner.gd** with the `class` name `FoodSpawner` that `extends` the `Interactable` class. Save the script in the **Scripts > Interactables** folder.

Then, define a new function `interact()` that takes no parameters and returns void. Inside the function, write a `print` statement to show when the function is called.

15 In the **interactable.gd** script, define the parent `interact()` function. Inside the function, write a different `print` statement to show when the `interact()` function is called in the parent class `Interactable`.

Then, modify the code so the `get_interactable_prompt()` function is called when Codey enters an Area3D and the `interact()` function is called when the player chooses to interact with a station.

Then, remove the **interactable.gd** script from the fridge, replace it with the **food_spawner.gd** script and update the variables in the Inspector as needed.

Playtest the project. What happens when Codey interacts with the fridge compared to the other stations?

```
Godot Engine v4.4.1.stable.official.49a5bc7b6 - https://godotengine.org
OpenGL API 3.3.0 - Build 27.20.100.8853 - Compatibility - Using Device: Intel - Intel(R) Iris(R) Xe Graphics

Cook on stove?
Interacting with station
```

```
Godot Engine v4.4.1.stable.official.49a5bc7b6 - https://godotengine.org
OpenGL API 3.3.0 - Build 27.20.100.8853 - Compatibility - Using Device: Intel - Intel(R) Iris(R) Xe Graphics

Get ingredient from fridge?
Getting something from Fridge.
```

16 Codey can interact with the fridge, but no food items are spawning on Codey's dish.

In the **interactable.gd** script, create a new `@export` variable `food_outputs` of type `Dictionary` and assign the variable to an`Dictionary`. Then, adjust the variable declaration so only dictionaries that contain **key-value pairs** of `Food` and `PackedScene` can be assigned to the variable. The `food_outputs` variable will track which stations can instantiate which foods and their corresponding scenes.

```
1  class_name Interactable extends Area3D
2
3  @export var cur_interactable: Globals.InteractableStation
4
5  @export var food_outputs: Dictionary[Globals.Food, PackedScene] = {}
6
7  var can_interact: bool = false
8
```

17 Some stations, like the fridge, will spawn an ingredient at random. In the **food_spawner.gd** script, define a function `choose_random_ingredient()` that takes a parameter `ingredients` of type `Dictionary` with `Food-PackedScene` key-value pairs and returns an `int`.

The function should select a random ingredient key and return the value of that key.

```
6
7  func choose_random_ingredient(ingredients: Dictionary[Globals.Food, PackedScene]) -> int:
8      # var ingredient_keys = ???.???
9      # var ingredient = ???(0, ???.??? -1)
10     # return ???[???]
11
```

Reminder:

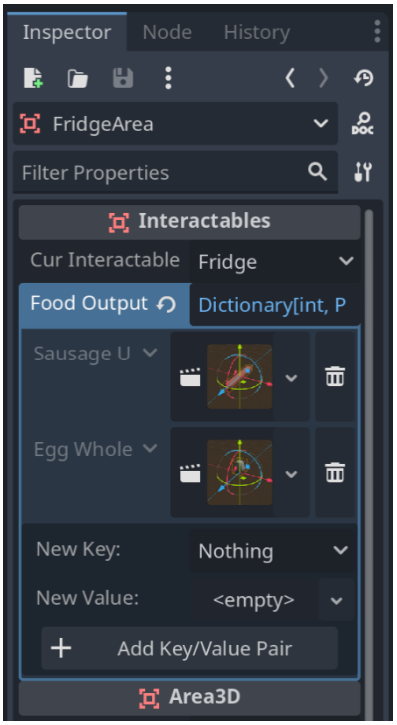
enums can be accessed by their constant (`Globals.Food.EGG_WHOLE`) or their integer value (`2`).

18

In the **Inspector** for the fridge, assign values to **Food Outputs**. Refer to the **project planning documents** as needed.

Then, call the `choose_random_ingredient()` function with `food_outputs` as the argument and print the result.

Playtest the project. Are ingredients selected at random?




```
12
13  enum Food {
14      NOTHING,
15      SAUSAGE_UNCOOKED,
16      EGG_WHOLE,
17      SAUSAGE_COOKED,
18      EGG_COOKED,
19      HOTDOG_BUN,
20      BREAD,
21      HOTDOG,
22      EGG_TOAST,
23      CONDIMENTS
24  }
```

```
Godot Engine v4.4.1.stable.official.49a5bc7b6 - http
OpenGL API 3.3.0 - Build 27.20.100.8853 - Compatibil

Get ingredient from fridge?
1
```

19

Create a new **utils.gd** script and save it in the **Scripts** folder. This script will contain functions and constants for the interactable stations that don't belong in the **Interactable** class.

Add the script as a **global variable**, **Utils**.

20

In the **utils.gd** script, define a constant `Y_OFFSET` of type `float` and a function `spawn_food_item()` to instantiate food scenes in the café. The `spawn_food_item()` function returns `void` and takes 3 parameters: `ingredient` of type `Food`, `location` of type `Node` and `ingredient_scene` of type `Dictionary`.

Use the `ingredient` parameter to access the corresponding PackedScene in the `ingredient_scene` dictionary and instantiate the scene as a child to `location`.

The constant `Y_OFFSET` can be used to adjust the instantiated scene's y position, so it appears "on top" instead of "inside" it's parent node. This constant may need to be adjusted later but assign it a value of `0.2` for now.

```
1   extends Node
2
3   const Y_OFFSET: float = 0.2
4
5   func spawn_food_item(ingredient: Globals.Food, location: Node,
6   ingredient_scene: Dictionary[Globals.Food, PackedScene]) -> void:
7       # var item = ???[???].???
8       # ???.???(item)
9       # var item_position = ???.global_transform.origin + ???.???.basis.??? * Y_OFFSET
10      # item.???.??? = item_position
11      # ???.???.basis = location.???.???
12
```

21

Spawn the food item Codey picks from the fridge.

In the **interactable.gd** script, create an `@onready` variable for the dish above Codey's head. Then, in the **food_spawner.gd** script, call the `spawn_food_item()` function inside the `interact()` function.

Playtest the game. Do random food items spawn on Codey's dish? Does the value of `Y_OFFSET` need to be adjusted? Can Codey get more than one ingredient from the fridge?

Pro Tip:

Make the **Player** node in the **chef_codey.tscn** scene a **Unique Name (%).**

22

Adjust the code so Codey can only grab one item from the fridge.

Inside the **utils.gd** script, define a function `is_nothing()` that takes a parameter `food` of type `Food` and returns a `bool`. The function should return `true` when `food` is `Globals.Food.NOTHING`.

23 In the **globals.gd** script, create a new variable `held_food` of type `int`. This variable tracks what Codey is holding on the dish using the enum's integer value.

Then, create a new script **game_manager.gd** with the `class` name `GameManager` that `extends` the `Node3D` class. Save the script in the **Scripts** folder.

Inside the **game_manager.gd** script, define the `_ready()` method and set the `held_food` variable to `NOTHING` when the game starts.

Then, attach the **game_manager.gd** script to the **ChefCodey** node.

24 Return to the **food_spawner.gd** script and update the `interact()` function to do the following:

- Use the `is_nothing()` function to check if Codey is holding something on the dish.
- Pick a random food item from the fridge if Codey isn't holding something, then update the `held_food` variable.
- Print a message to the console if Codey is already holding something.

Playtest the project. Can Codey still pick multiple items from the fridge?

Pause for **Sensei Stop #2**!

Check with a Code Sensei and confirm that requirements 6 – 8 are set up correctly.

Reminder: Save your work!

Requirement #9: Code the Trashcan

The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... ~~• Defining game constants.~~ • Providing helper functions that are used throughout many scripts.
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #9: Code the Trashcan

Code a script that discards the item Codey is holding in the trashcan.

- Create a script named **trash.gd** and save it in the **Scripts > Interactables** folder. The script should have the `class` name `Trash` that `extends` the `Interactable` class.
- In the **utils.gd** script, define a function `remove_food_item()` that takes a parameter `location` of type `Node` and returns `void`. Code the script to remove the child node from `location`.
- In **trash.gd** script, define an `interact()` function which does the following:
  - Prints a message if Codey tries to discard an item when nothing is on Codey's dish.
  - If Codey is holding an item on the dish, remove the item and update the `held_food` variable to `NOTHING`.
- Remove the interactable script from the trash item and attach the trash script. Update the necessary values in the Inspector, then playtest the project.

Pause for **Ninja Stop #6**!

Test your project! Does...

- The trash remove the food item from Codey's dish?
- Can Codey pick another ingredient from the Fridge after discarding an item in the trash?

Reminder: Save your work!

Requirement #9: Hints

- [ ] What function in **Utils** might be helpful to check whether a value is nothing?
- [ ] Are the `get_child()` and `queue_free()` methods used in the `remove_food_item()` function?

```
13
14 func is_nothing(food: Globals.Food) -> bool:
15     return food == Globals.Food.NOTHING
16
17 func remove_food_item(location: Node) -> void:
18     # var held_item = ???.get_child(0)
19     # ???.???
20
```

Requirement #9: Resources

- [ ] Refer to **Requirement #8: Code the Fridge**
- [ ] Godot 4.4 get_child() documentation

Requirement #10 (Instructions): Place an Item on the Stove

The Player is responsible for...
- ~~User input & movement.~~
- Only allowing movement when the game is running.

The Globals autoload is responsible for...
- ~~Defining enums.~~
- Storing the game state.

All Interactables are responsible for...
- ~~Displaying a tooltip when the Player gets close.~~
- ~~Calling interact() when the Player is close and presses the "interact" action.~~

Each Interactable is responsible for...
- Its own interact() functionality.

The Utils autoload is responsible for...
- ~~Defining game constants.~~
- Providing helper functions that are used throughout many scripts.

The Game Manager is responsible for...
- Setting up the game.
- Updating the countdown timer.
- Ending the game.

The User Interface is responsible for...
- Storing variable references to UI nodes.
- Updating the visible and text properties of the UI nodes.

The Main Menu is responsible for...
- Displaying the game title.
- Navigating to the game or Help Screen.

Requirement #10 (Instructions): Place an Item on the Stove

25 Create a new script **stove.gd** with the `class` name `Stove` that `extends` the `Interactable` class. Save the script in the **Scripts > Interactables** folder.

In the **stove.gd** script, define a new function `cook_on_stove()`. The function takes one parameter `ingredient` of type `Food` and returns `void`.

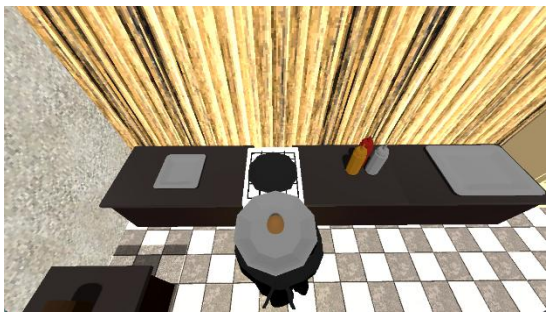
Code the function to spawn a food item on the stove, remove the food item from Codey's dish and update the `held_food` variable to `NOTHING`.

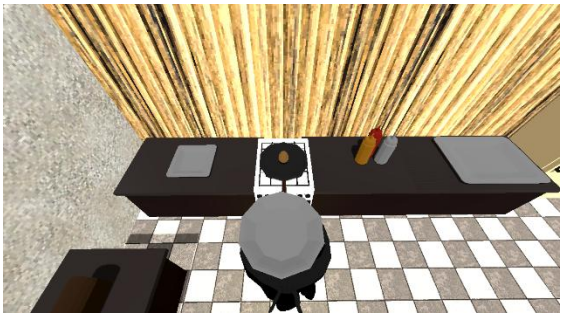
A new variable will need to be created for the pan/pot on the stove.

26

Some food items should not appear on the stove like they do on Codey's dish.

In this example, Codey is holding a whole egg on the dish. If the exact item that Codey is holding is spawned onto the pan, Codey appears to be cooking an egg inside its shell.





Create an `@export` variable `stove_inputs` of type `Dictionary[Food, Food]` and set it to an empty dictionary.

This variable will be used to track what foods spawn in the stove using **input-output key-value** pairs. In this dictionary, the key is the food item on Codey's dish (whole egg) and the value is how the food item appears in the pan (cooked egg). In this example, one entry would look like this:
`[Food.EGG_WHOLE: Food.EGG_COOKED]`

Some items may appear on the stove as they appear on Codey's dish. These items still need to be included in the `stove_inputs` dictionary.

Save the script. Then, update the values for **Stove Inputs** in the **Inspector**. It may be helpful to refer to the **project planning documents**.

27 A function is needed to get a **value** from the `stove_inputs` dictionary so the correct food item can be spawned on the stove.

Return to the **utils.gd** script and define a new function `get_transformed_food()`. The function takes 2 parameters, `input` of type `Food` and `transformations` of type `Dictionary[Food, Food]`, and returns `Food`.

If `input` is a **key** in the `transformations` dictionary, `return` its **value**. Otherwise, the function should `return NOTHING`.

```
22
23  # function: get_transformed_food()
24  # parameters[2]: input (Food), transformations (Dictionary[Food, Food]
25  # return: Food
26      # if ???.has(???)
27          # return ???[???]
28      # else
29          # return ???
30
```

28 Return to the **stove.gd** script and define the `interact()` function. Inside the function, create a new variable `item_to_cook` and set it to the value returned by the `get_transformed_food()` function. What two variables might be passed through the function as arguments?

Then, on the next line, call the `cook_on_stove()` function with `item_to_cook` as the argument.

Save the script and playtest the game. Can Codey get an item from the fridge, then place the item on the stove? Can multiple items be placed on the stove?

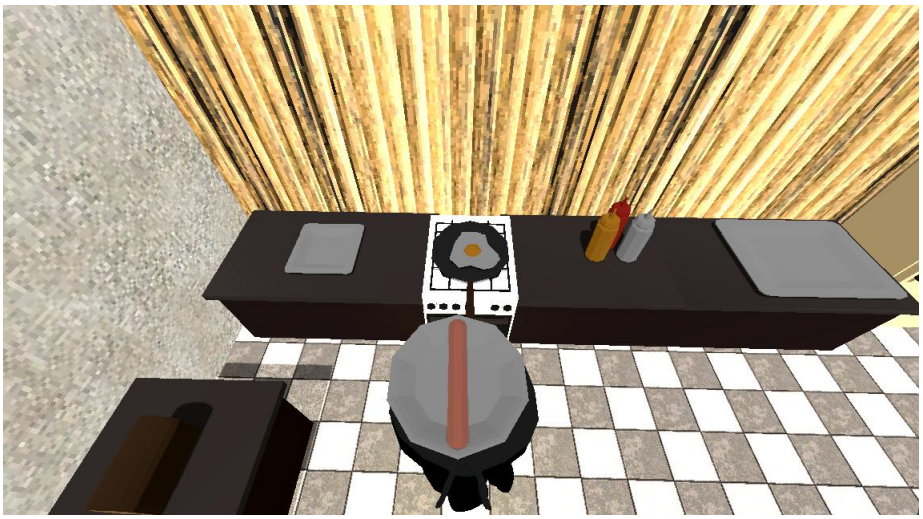
29 Currently, Codey can cook multiple items on the stove at the same time.

In the **globals.gd** script, create a new variable `stove_is_empty` of type `bool`. Then, in the **game_manager.gd** script, set `stove_is_empty` to `true` when the game starts.

Return to the **stove.gd** script and update the `cook_on_stove()` function so the `stove_is_empty` variable is set to `false` when an item is placed on the stove.

Then, in the `interact()` function, write an `if-statement` to call the `cook_on_stove()` function if the stove is empty.

Playtest the game. Can Codey still place multiple items on the stove? What happens if Codey tries to interact with the stove without a food item on the dish?



30

If Codey tries to interact with the stove without a food item on the dish, an error crashes the game.

In the **stove.gd** script, define a new function `try_place_on_stove()` that returns `void`.

Inside the function, write an `if-statement` to check if Codey is holding nothing. If Codey is holding nothing, print a message to the console and exit the function with `return`.

Then, move the lines of code for the `item_to_cook` variable and the call to the `cook_on_stove()` function ***out*** of the `interact()` function and ***into*** the `try_place_on_stove()` function.

```
17 func try_place_on_stove() -> void:
18     # if ???:
19         # print("Nothing to cook!")
20         # return
21
22     var item_to_cook = Utils.get_transformed_food(Globals.held_food, stove_inputs)
23     cook_on_stove(item_to_cook)
```

31

Update the `interact()` function so the `try_place_on_stove()` function is called if the stove is empty, and playtest the game.

What happens now when Codey tries to interact with the stove without a food item on the dish?

Pause for **Ninja Stop #7**!

Test your project!

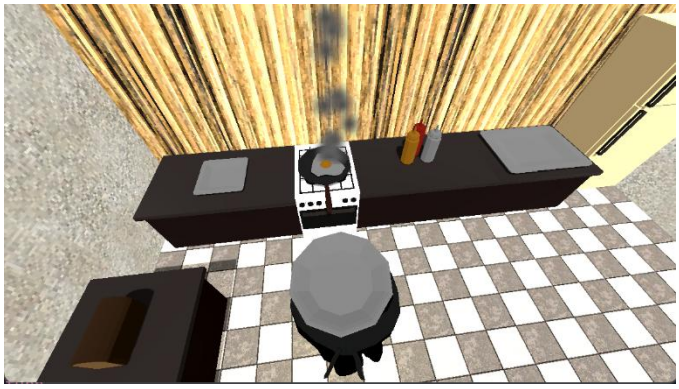
- Can Codey place an item on the stove?
- Does the game crash if Codey tries to place nothing on the stove?

Reminder: Save your work!

Requirement #11: Cook an Item on the Stove

Create a smoke effect using particles to emulate the cooking process on the stove.

- Add a **CPUParticles3D** node as a child to the **Stove** area and rename the node to **SmokeParticles**.
- In the **Inspector** for **SmokeParticles**, update the properties in the **Inspector** to create a smoke effect. It may be helpful to consider the following properties:
  - Amount
  - Lifetime
  - Randomness
  - Gravity
  - Velocity Max
  - Mesh & Material – there is a **smoke.png** texture in the **Assets>Textures** folder for the smoke effect.
- Reposition the particles so they appear in the stove's pan, then turn off emitting.
- In the **stove.gd** script create the following variables:
  - An `@export` variable `cook_time` of type `int`.
  - An `@onready` variable for the **SmokeParticles** node.
- Update the `cook_on_stove()` function to do the following:
  - Start emitting particles when a food item is placed on the stove.
  - Print a message to the console to let the player know the item has started cooking.
  - Create a timer that times out after `cook_time`. The timer uses the `cook_time` variable to determine how long the item cooks on the stove.
  - Stop emitting particles when the item has finished cooking.
  - Print a message to the console to let the player know the item is done cooking.



Pause for **Ninja Stop #8**!

Test your project!

- Do the smoke particles start/stop when an item starts/stops cooking?
- Does a message print to the console to let the player know when an item has started/stopped cooking?

Reminder: Save your work!

Requirement #11: Hints

- ☐ Is a value for `cook_time` set in the Inspector?
- ☐ Is a timer created using `await` and the `cook_time` variable?
- ☐ Is a **positive y-value** used for Gravity to move the particles up?

Requirement #11: Resources

- ☐ Refer to particles in **BB Activity 12: Dropping Bombs Part 6**

Requirement #12: Pickup Item Station

The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... ~~• Defining game constants.~~ • Providing helper functions that are used throughout many scripts.
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #12: Pickup Item Station

To pick up items from the stove, a Pickup Item is required. Set up the Pickup Item station so Codey can grab the Pickup Item while something is cooking on the stove.

It may be helpful to refer to the **project planning documents**.

- ☐ Set up the Pickup Item station to do the following:
  - Spawn a Pickup Item on Codey's dish.
  - Print a message to console if Codey tries to get an item from the station while holding something on their dish.
    Note: Depending on the required functionality, it may be helpful to use the **food_spawner.gd** script.

Pause for **Ninja Stop #9**!

Test your project!

- Can Codey interact with the Pickup station as expected?

Reminder: Save your work!

Requirement #12: Hints

- Does this station work similarly to the fridge? Can the **food_spawner.gd** script be used?

Requirement #12: Resources

- Refer to Requirement #8: Code the Fridge.

Requirement #13 (Instructions): Pick up an Item from the Stove

The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... ~~• Defining game constants.~~ • Providing helper functions that are used throughout many scripts.
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #13 (Instructions): Pick up an Item from the Stove

32 To pick up an item from the stove, the following information is needed:

- The item Codey is holding.
- The current item on the stove.
- If Codey can use the item their holding to pick up the item on the stove.
- The new item Codey will hold after picking up the item on the stove.

Currently, the `held_food` variable provides information about what Codey is holding, but the rest is still unknown.

In the **stove.gd** script, create a variable `on_stove` of type `Food` and set the variable to `NOTHING`.

This variable will track what food item is on the stove.

33 Add the `on_stove` variable to the `cook_on_stove()` function. What variable or parameter can be used in the function to update what is being cooked on the stove?

34 To determine if Codey can use the item on their dish to pick up the item on the stove, a new function is needed.

Define a function `get_required_pickup_item()` that takes parameter `food` of type `Food` and returns `Food`.

Inside the function, use a `match` statement to match the items that the stove can cook with their Pickup Item. The function should return the required Pickup Item, or `NOTHING` if `food` isn't something the stove can cook.

```
40  # function: get_required_pickup_item()
41  # parameters[1]: food (Food)
42  # return: Food
43      # var food_needed := Globals.Food.NOTHING
44      # match ???:
45          # Globals.Food.SAUSAGE_COOKED:
46              # food_needed = Globals.Food.HOTDOG_BUN
47          # ???:
48              # ??? = ???
49      # return ???
```

35

In the `interact()` function, add an `if-else` statement to check if the item on Codey's dish is the required Pickup Item. The value returned by the `get_required_pickup_item()` function will be needed. What variable might be used as the argument for the `get_required_pickup_item()` function?

Inside the `if-else` statement, print a message to console to let the player know if the item on the stove can be picked up with what Codey is holding.

Then, playtest the game. What happens when Codey tries to pick up different food items off the stove? Can incorrect pickup items be used?

36

When picking up an item off the stove, only a single item should appear on Codey's dish. In the Stove Station section of the project **planning documents**, this is the **Stove Output**.

Similarly to the `stove_inputs` dictionary, a `food_transformations` dictionary can be used to track which Pickup Items lead to the Stove Output.

In the **interactable.gd** script, define a new variable `food_transformations` of type `Dictionary[Food, Food]` and set the variable to an empty dictionary.

This dictionary is created in the interactable class because it will be used by other interactable items later.

37 Return to the **stove.gd** script.

In the `interact()` function, return to the `if-else` statement that checks if Codey is holding the correct Pickup Item. Inside the `if`-statement, define a new variable `item_to_pickup` and assign it to the value returned by the `get_transformed_food()` function in the **utils.gd** script.

What two variables might be passed through as arguments?

38 Review the list of information needed to pick up an item from the stove:

- The item Codey is holding.
- The current item on the stove.
- If Codey can use the item their holding to pick up the item on the stove.
- The new item Codey will hold after picking up the item on the stove.

Now, there are functions and variables that provide all the above information.

Define a new function `done_cooking()` that takes a parameter `cooked` of type `Food` and returns `void`. Inside the function, remove the items from the stove and Codey's dish, then spawn the `cooked` item on Codey's dish. What functions in the **utils.gd** script might be called here?

To complete the cooking process, the `held_food`, `on_stove` and `stove_is_empty` variables also need to be updated in the `done_cooking()` function.

39 Inside the **stove.gd** `interact()` function, update the `if-else` statement to call the `done_cooking()` function with `item_to_pickup` as the argument.

Then, in the Inspector, update the **Food Transformations** and **Food Output** dictionaries. Refer to the **project planning documents** as needed.

Playtest the game. The code still contains 3 bugs that need to be fixed, one of which breaks the game. What might they be?

40 The game crashes if Codey tries to place a Pickup Item on the empty stove. Why might this be?

Recall, the `get_transformed_food()` function in the **utils.gd** script returns `NOTHING` if `input` isn't a key in the `transformations` dictionary.

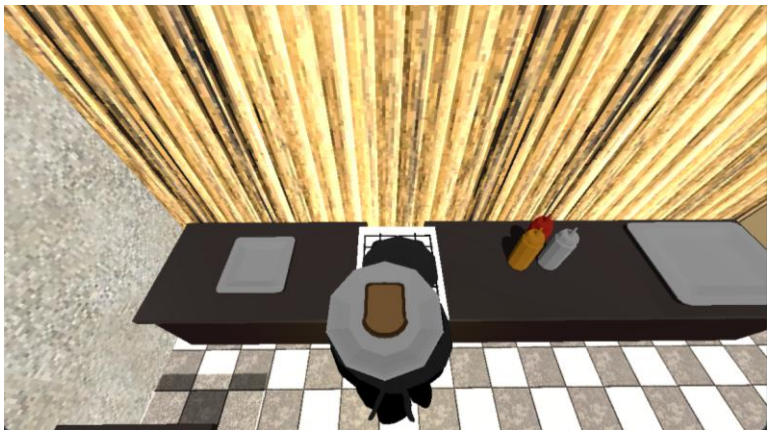
```
func get_transformed_food(input: Globals.Food, transformations: Dictionary[Globals.Food, Globals.Food]) -> Globals.Food:
    if transformations.has(input):
        return transformations[input]
    else:
        return Globals.Food.NOTHING
```

If Codey tries to place an item on the stove that is not in the `stove_inputs` dictionary, then `item_to_cook` is passed though the `cook_on_stove()` function as `NOTHING`. This causes an error since `NOTHING` can't be instantiated on the stove.

```
var item_to_cook = Utils.get_transformed_food(Globals.held_food, stove_inputs)
cook_on_stove(item_to_cook)
```

Add an `if-else` statement to the `try_place_on_stove()` function so `cook_on_stove()` is called if `item_to_cook` isn't `NOTHING` and an error message is printed to the console if Codey tries to cook a "wrong" item on the stove.

41 Playtest test the game. What happens when Codey tries to cook a "wrong" item on the stove? What can be noticed about the console output?



```
Godot Engine v4.4.1.stable.official.49a5bc7b6 - https://godotengine.org
OpenGL API 3.3.0 - Build 27.20.100.8853 - Compatibility - Using Device: Intel - Intel(R) Iris(R) Xe Graphics

Plate the dish?
Get ingredient from fridge?
Throw away dish?
Pick up bread?
Cook on stove?
Can't cook item in pan.
Not holding the correct item
```

42 Multiple messages print to the console when Codey tries to place a "wrong" item on the stove. This is because the **`if-else`** statement inside the **`interact()`** function is executing after the **`try_place_on_stove()`** function.

Inside the if-statement, after the **`try_place_on_stove()`** function is called, add **`return`** to stop the function's execution and prevent further interaction with the stove.

```
14  func interact() -> void:
15      if Globals.stove_is_empty:
16          try_place_on_stove()
17          return
18      if Globals.held_food == get_required_pickup_item(on_stove):
19          var item_to_pickup = Utils.get_transformed_food(Globals.held_food, food_transformations)
20          done_cooking(item_to_pickup)
21      else:
22          print("Not holding the correct item")
23
```

Playtest the project. The game should no longer crash, but what other bugs can be found?

43 Codey can pick up an item from the stove while it's still cooking.

In the **stove.gd** script, define a new variable `is_cooking` of type `bool` and set it to `false`. Update the `cook_on_stove()` function so the variable is set to `true` when the cooking process starts and `false` when the item is finished cooking.

Then, update the `interact()` function so that a message prints to the console if Codey tries to pick up an item off the stove while it's still cooking. Is `return` needed inside the `if`-statement to prevent additional bugs in the game?

44 Inside the `interact()` function, add an if-statement to print a message to the console if Codey tries to pick up an item off the stove without something on their dish. Is `return` needed inside the `if`-statement to prevent additional bugs in the game?

Playtest the game! Does the stove function as expected? Are there any additional bugs that can be found?

45 The stove should now be fully functional. However, several messages print to the console depending on the type of interaction Codey is having. More descriptive messages will help guide the player, especially when the UI is added in later.

In the **utils.gd** script, define a new function `food_to_string()` that takes a parameter `food` of type `Food` and returns a `String`. Inside the function, use a `match` statement to return a string name for each value in the Food enum.

46 In the **interactable.gd** script, define a new variable `interact_tooltip` of type `String` and set it to an empty string. This variable will store the strings printed to the console that will later appear in the UI.

Then, update the print statement inside the `on_area_entered()` function to include the `interact_tooltip` variable.

```
func on_area_entered(_area: Area3D) -> void:
    can_interact = true
    interact_tooltip = get_interactable_prompt(cur_interactable)
    print(interact_tooltip)
```

47 Return to the **stove.gd** script and update the print statements to include the `interact_tooltip` variable and `food_to_string()` function. The messages printed to the console should help guide the player through the game.

```
if Utils.is_nothing(Globals.held_food):
    interact_tooltip = "Can't pick up " + Utils.food_to_string(on_stove) + \
    " without " + Utils.food_to_string(get_required_pickup_item(on_stove)) + "."
    print(interact_tooltip)
    return
```

Pro Tip:

When concatenating long strings, use a **backslash (\)** to break them up on multiple lines without causing errors.

Pause for **Sensei Stop #3**!

Check with a Code Sensei and confirm that requirements 9 – 13 are set up correctly.

Requirement #14: Plate Items

The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... ~~• Defining game constants.~~ ~~• Providing helper functions that are used throughout many scripts.~~
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #14: Plate Items

Set up a station to plate completed dishes. This could be one station for multiple dishes, or specific station for each dish.

- ☐ Create a new script **plate.gd** with the `class` name `Plate` that `extends Interactable` and save the script in the **Scripts > Interactables** folder.
- ☐ Define the `interact()` function inside the script. The function should do the following:
  - Print a message to console if Codey tries to plate `NOTHING`.
  - Print a message to console if Codey tries to plate an incomplete or "wrong" item.
  - Remove the food item from Codey's dish and place it on the plate if Codey tries to plate a complete dish.
  - Prevent Codey from placing more than one dish on the plate with an `plate_is_empty` variable of type `bool`.
- ☐ Use the `interact_tooltip` variable when printing messages to the console.
- ☐ Attach the **plate.gd** script to the necessary stations and update variables in the Inspector as needed.
- ☐ Set up any additional stations required for the cooking process. This may be a station that applies condiments to a hotdog or puts toppings on a hamburger before it's ready to be served.

Pause for **Ninja Stop #10**!

Test your project!

- Can Codey only plate completed dishes?
- Can Codey only place one dish on a plate at a time?
- Are all other interactable stations (minus the completion bell) set up and working?

Reminder: Save your work!

Requirement #14: Hints

- [ ] Is an `@onready` variable created for the plate (or item) that the complete items will appear on?
  Note: if working with multiple plates, make sure the plate/item the complete items will appear on have the same name.

- [ ] Are the `is_nothing()` and `get_transformed_food()` functions in the utils.gd script used as needed?

- [ ] Is the `food_transformations` variable used to see if the item Codey is holding can be plated?

- [ ] Is `return` used in the `interact()` function?

- [ ] Is the `plate_is_empty` variable defined in the **plate.gd** script and updated as needed?

```
func interact() -> void:
    # if ???:
        # interact_tooltip = "Not holding anything!"
        # print(???)
        # return
    # if !???:
        # interact_tooltip = "Something already on the plate!"
        # print(???)
        # return

    # var result = Utils.get_transformed_food(Globals.held_food, food_transformations)
    # if !???:
        # ???
        # ???
        # Globals.held_food = ???
        # plate_is_empty = false
    # else:
        # interact_tooltip = "Can't plate " + Utils.food_to_string(Globals.held_food) + "."
        # print(???)
```

Requirement #14: Resources

- [ ] Refer to requirement #13: Pick Up an Item from the Stove.

Requirement #15: Code the Bell

The Player is responsible for... • ~~User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... • ~~Defining enums.~~ • Storing the game state.
All Interactables are responsible for... • ~~Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • Its own interact() functionality.
The Utils autoload is responsible for... • ~~Defining game constants.~~ • ~~Providing helper functions that are used throughout many scripts.~~
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #15: Code the Bell

Set up the bell station to remove completed dishes from their plates when the bell is hit.

- ☐ Add the completed dish scenes to a **global group** called **Dish**.
- ☐ In the **globals.gd** script, create a new global `signal bell_hit`.
- ☐ Create a new script **bell.gd** with the `class` name `Bell` that `extends` `Interactable` and save the script in the **Scripts > Interactables** folder.
- ☐ In the **bell.gd** script, define the `interact()` function that emits the `bell_hit` signal when Codey interacts with the Bell station.
- ☐ In the **plate.gd** script, define a new function `on_bell_hit()` that returns void. Code the function to do the following:
  - Find all the plate (or item) node's children.
  - Remove the children if they're in a specific group.
  - Update the `plate_is_empty` variable.

```
func on_bell_hit() -> void:
    # var children = ???.get_children()
    # for ??? in ???:
        # if ???.???("Dish")
            # ???
            # ??? = true
```

- ☐ Connect the `bell_hit` signal with code to the `on_bell_hit()` function when the game starts.

Pause for **Ninja Stop #11**!

Test your project!

- Do completed dishes disappear when Codey interacts with the bell?
- Can completed items be placed on the plate(s) after the bell is hit?

Reminder: Save your work!

Requirement #15: Hints

- Is the `_ready()` method defined in the Plate class to connect the `bell_hit` signal to the `on_bell_hit()` function?
- Notice the child class **Plate** and the parent class **Interactable** both contain a `_ready()` method to connect signals with code when the game starts. Is the `super()` method called at the top the Plate class's `_ready()` method so it runs after the `_ready()` method in the Interactable class instead of overriding it?
- Is the `plate_is_empty` variable set to true after an item is removed from the plate?

Requirement #15: Resources

- Refer to coding and connecting signals **in SB Activity 06: Evil Fortress of Doctor Worm.**
- Refer to Requirement #7 and #8 in **PB Activity 02: Codey Raceway** for help with emitting and connecting signals.
- Godot 4.4 Inheritance Documentation

Requirement #16: Create The User Interface

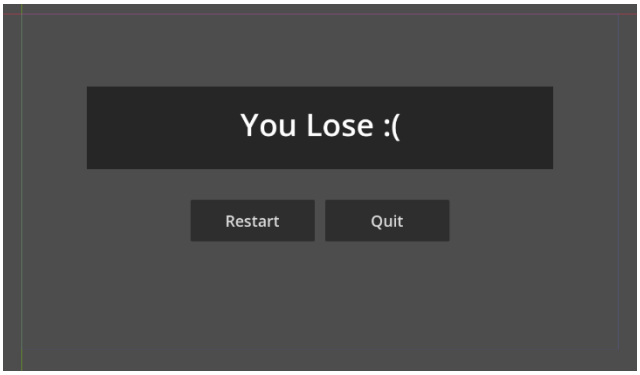
The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... ~~• Its own interact() functionality.~~
The Utils autoload is responsible for... ~~• Defining game constants.~~ ~~• Providing helper functions that are used throughout many scripts.~~
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #16: Create The User Interface

Create a user interface to display game prompts when Codey interacts with a station and a game over screen.

- Create a new **user_interface.tscn** scene with a **CanvasLayer** root node. Save the scene in a **Scenes > UI** folder.
- In the **user_interface.tscn** scene, add two **Control** nodes renamed to **HUD** and **GameOver** to organize and manage the different UI components.
- Set up the **HUD** screen with the following:
  - The player's score and target score.
  - A countdown timer.
  - The interact tooltip prompt.
    Note: it may be helpful to use ColorRect nodes.

- Set up the **GameOver** screen with the following:
  - A game over message.
  - A restart button.
  - A quit/main menu button.



Requirement #16: Hints

- ☐ Are **ColorRect** nodes used to highlight displayed text?
- ☐ Is an **HBoxContainer** used to organize and align the buttons?
- ☐ Is a separate label used for each UI component with text that will be updated?
- ☐ Are nodes renamed to keep the UI organized?

Requirement #16: Resources

- ☐ Refer to creating an HUD **in SB Activity 13: Food Frenzy Part 1.**
- ☐ Refer to setting up ColorRect nodes **in SB Activity 14: Scavenger Hunt Deluxe.**
- ☐ Refer to using HBoxContainer nodes and buttons **in SB Activity 15: Amazing Ninja Worlds Part 3.**
- ☐ Refer to creating a game over screen **in SB Activity 16: Food Frenzy Part 2.**

Requirement #17: Code the User Interface

The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... ~~• Its own interact() functionality.~~
The Utils autoload is responsible for... ~~• Defining game constants.~~ ~~• Providing helper functions that are used throughout many scripts.~~
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... • Storing variable references to UI nodes. • Updating the visible and text properties of the UI nodes.
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #17: Code the User Interface

Code the user interface to update its text and show/hide the different UI screens.

- ☐ Create a new **user_interface.gd** script and save it in the **Scripts > UI** folder. Inside the script, define and code the following functions:
  - A function `update_target_text()` that takes a parameter `target` of type `int` and returns `void`. The function should set the player's target score.
  - A function `update_score_text()` that takes a parameter `score` of type `int` and returns `void`. The function should update the player's score.
  - A function `update_countdown_text()` that takes a parameter `text` of type `String` and returns `void`. The function should update the game countdown.
  - A function `update_tooltip_text()` that takes a parameter `text` of type `String` and returns `void`. The function should make the label visible and update its text.
  - A function `hide_tooltip_text()` that takes no parameters and returns `void`. The function should hide the tooltip label.
  - A function `game_over()` that takes a parameter `game_won` of type `bool` and returns `void`. The function should hide the tooltip label, update the game over text depending on whether the game is won or lost and show the GameOver screen.
  - The `_ready()` method, which hides parts of the UI that aren't needed at the start of the game.
- ☐ Attach the **user_interface.gd** script to the **UserInterface** node, then add the scene as a child to **ChefCodey**.
- ☐ Update **all the scripts** in the **Interactable class** so the prompts are displayed on the UI screen instead of printed to the console. Check that all the scripts assign the string to the `interact_tooltip` variable, then update the UI using `interact_tooltip` as an argument.

Pro Tip:

Make the **UserInterface** node in the **chef_codey.tscn** scene a **Unique Name (%).**

- ☐ In the **interactable.gd** script, update the code to hide the tooltip label when Codey exits an interactable area.

- In the **interactable.gd** script, find the `interact()` function. The `interact()` function in the parent class should not be called anywhere since all child classes define their own version of `interact()`. However, the function is still needed in the parent class. Replace the print statement in the `interact()` function with `pass`.
- In the `get_interactable_prompt()` function, use the `stove_is_empty` variable and the `get_nothing()` function to update the string returned when interacting with the stove. A different interactable prompt should be returned depending on whether Codey is trying to place an item on the empty stove or pick up an item off the stove.

Pause for **Ninja Stop #12**!

Test your project!

- Do the interactable prompts appear and disappear in the UI as expected?
- Do messages no longer print to the console?
- Does the stove display different UI prompts depending on how Codey might interact with it?

Reminder: Save your work!

Requirement #17: Hints

- [ ] Is the **UserInterface** node a **Unique Name**?
- [ ] Is there an `@onready` variable for the **UserInterface** node in the **interactable.gd** script?
- [ ] Is the `on_area_exited()` function updated to hide the tooltip label when Codey leaves an interactable area?
- [ ] Is an `if`-statement used inside the `get_interactable_prompt()` function to update the stove's interactable prompt depending on whether the stove is empty and if Codey is holding something?

```
Globals.InteractableStation.STOVE:
    # if ???:
        # if ???:
            # return "Cook " + Utils.food_to_string(Globals.held_food) + " on the stove?"
        # else:
            # return "Nothing to cook!"
    # else:
        # return "Pick up item off the stove?"
```

Requirement #17: Resources

- [ ] Refer to updating an HUD **in SB Activity 13: Food Frenzy Part 1.**

Requirement #18: Create The Main Menu & Help Screen

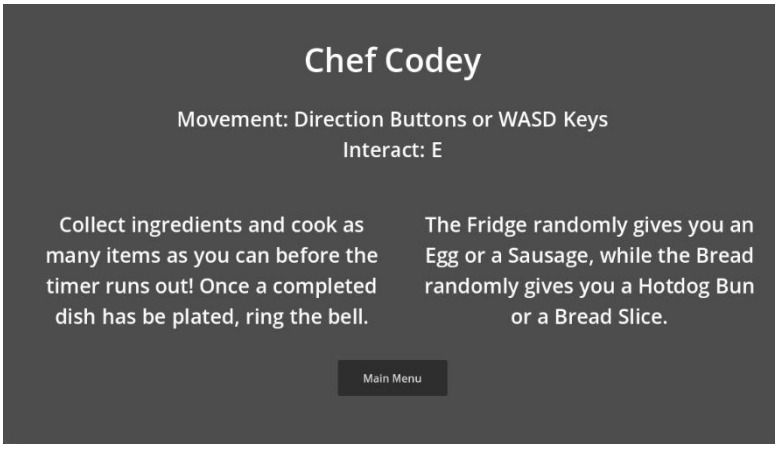
The Player is responsible for... ~~• User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... ~~• Defining enums.~~ • Storing the game state.
All Interactables are responsible for... ~~• Displaying a tooltip when the Player gets close.~~ ~~• Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... ~~• Its own interact() functionality.~~
The Utils autoload is responsible for... ~~• Defining game constants.~~ ~~• Providing helper functions that are used throughout many scripts.~~
The Game Manager is responsible for... • Setting up the game. • Updating the countdown timer. • Ending the game.
The User Interface is responsible for... ~~• Storing variable references to UI nodes.~~ ~~• Updating the visible and text properties of the UI nodes.~~
The Main Menu is responsible for... • Displaying the game title. • Navigating to the game or Help Screen.

Requirement #18: Create The Main Menu & Help Screen

Create a Main Menu and Help Screen for the game. These scenes should have some core functionality but can be customized using textures in the Assets folder as needed.

- Create a new **main_menu.tscn** scene with a **Control** node as the root and save it in the **Scenes > UI** folder. Design the scene to include the game title, **Start** button and **Help** button.

- Create a new **help_screen.tscn** scene with a **Control** node as the root and save it in the **Scenes > UI** folder. Design the scene to provide game instructions for the player, and a button to return to the Main Menu.



- Use the **change_scene_button.gd** script in the **Scripts > UI** folder so the buttons in the **main_menu.tscn**, **help_screen.tscn** and **user_interface.tscn** scenes work as needed. The script connects the `pressed` signal with code, so additional signals do not need to be manually connected in the Godot interface.
- Set the **Main Scene** to the **main_menu.tscn** scene.

Pause for **Ninja Stop #13**!

Test your project! Does it have...

- Do the all the buttons work as expected to move between the Main Menu, Help Screen and Game?
- Is the Main Scene set to the Main Menu?

Reminder: Save your work!

Requirement #18: Hints

- ☐ Is the **change_scene_button.gd** script attached to all the button nodes in the **main_menu.tscn**, **help_screen.tscn** and **user_interface.tscn** scenes?
- ☐ In the **Inspector** for each button, is the **Target Scene** set to the correct scene?

Requirement #18: Resources

- ☐ Refer to using buttons to navigate between game screens **in SB Activity 15: Amazing Ninja Worlds Part 3.**

Requirement #19: Code the Score and Countdown Timer

The Player is responsible for...

- ~~User input & movement.~~
- Only allowing movement when the game is running.

The Globals autoload is responsible for...

- ~~Defining enums.~~
- Storing the game state.

All Interactables are responsible for...

- ~~Displaying a tooltip when the Player gets close.~~
- ~~Calling interact() when the Player is close and presses the "interact" action.~~

Each Interactable is responsible for...

- ~~Its own interact() functionality.~~

The Utils autoload is responsible for...

- ~~Defining game constants.~~
- ~~Providing helper functions that are used throughout many scripts.~~

The Game Manager is responsible for...

- Setting up the game.
- Updating the countdown timer.
- Ending the game.

The User Interface is responsible for...

- ~~Storing variable references to UI nodes.~~
- ~~Updating the visible and text properties of the UI nodes.~~

The Main Menu is responsible for...

- ~~Displaying the game title.~~
- ~~Navigating to the game or Help Screen.~~

Requirement #19: Code the Score and Countdown Timer

Add a score to the game!

- ☐ In the **game_manager.gd** script, create an `@export` variable `target_score` of type `int` and assign a target score for the game.
- ☐ In the **globals.gd** script, create a `score` variable of type `int` to track the number of completed dishes the player has cooked.
- ☐ In the **game_manager.gd** script set the `score` to `0` and update the Score and Target Score labels on the UI screen at the start of the game.
- ☐ Update the code so the score is increased when Codey rings the bell after completing a dish.

Add a timer to the game!

- ☐ In the **game_manager.gd** script, create an `@export` variable `time_remaining` of type `float` and assign a time limit to the game.
- ☐ Define a function `_format_time()` that takes a parameter `time` of type `float` and returns a `String`. The function should format `time` to display how many minutes and seconds remain in the game.
- ☐ Code the `_process()` method to decrease `time_remaining` and update the countdown timer on the User Interface.

Pause for **Ninja Stop #14**!

Test your project!

- Does the countdown timer decrease as time passed in the game?
- Is the score increased when Codey completes a dish?

Reminder: Save your work!

Requirement #19: Hints

- ☐ Does the **game_manager.gd** script contain an `@onready` variable for **UserInterface**?
- ☐ Is the `score` variable increased inside `the plate.gd` script's `on_bell_hit()` function when a completed dish is removed from the plate?
- ☐ Is the score label updated after the score is increased?
- ☐ Is `time_remaining` decreased by `delta`?

Requirement #19: Resources

- ☐ Refer to Requirement #18: Add a Race Countdown **in PB Activity 02: Codey Raceway**.
- ☐ Refer to Requirement #19: Add a Time Limit **in PB Activity 02: Codey Raceway**.
- ☐ Refer to the format time function in **SB Activity 13: Food Frenzy Part 1.**

Requirement #20: Code the Game Over Conditions

The Player is responsible for... • ~~User input & movement.~~ • Only allowing movement when the game is running.
The Globals autoload is responsible for... • ~~Defining enums.~~ • Storing the game state.
All Interactables are responsible for... • ~~Displaying a tooltip when the Player gets close.~~ • ~~Calling interact() when the Player is close and presses the "interact" action.~~
Each Interactable is responsible for... • ~~Its own interact() functionality.~~
The Utils autoload is responsible for... • ~~Defining game constants.~~ • ~~Providing helper functions that are used throughout many scripts.~~
The Game Manager is responsible for... • Setting up the game. • ~~Updating the countdown timer.~~ • Ending the game.
The User Interface is responsible for... • ~~Storing variable references to UI nodes.~~ • ~~Updating the visible and text properties of the UI nodes.~~
The Main Menu is responsible for... • ~~Displaying the game title.~~ • ~~Navigating to the game or Help Screen.~~

Requirement #20: Code the Game Over Conditions

Add a game over function to stop Codey's movement and show the game over screen when the time limit runs out.

- In the **game_manager.gd** script, create a variable `game_is_running` of type `bool` and set the variable to `false`. This variable will be used to stop the player movement when the game ends.
- Define a `game_over()` function that returns `void`. Code the function to display the game win screen if the player reaches the target score before the timer runs out.
- Update the script so `game_is_running` is `true` when the game starts and `false` when the game is over.
- In the **globals.gd** script, create a variable `game_manager` of type `GameManager`. Then, in the **game_manager.gd** script, set `game_manager` to `self` when the game starts.
- In the **player.gd** script, find `TODO 1` inside the `_physics_process()` method. Uncomment the line of code underneath the TODO and modify the function so Codey can only move if `game_is_running` is `true`.

Requirement #20: Hints

- ☐ Is `game_is_running` set to `true` in the `_ready()` method and `false` in the `game_over()` function?
- ☐ Is the `game_over()` function called inside the process method when `time_remaining` reaches 0?
- ☐ Is the `game_over()` function in the **user_interface.gd** script called to update the game over screen as needed?

Requirement #20: Resources

- ☐ Refer to **SB Activity 16: Food Frenzy Part 2.**

Pause for **Sensei Stop #4**!

Check with a Code Sensei and confirm that requirements 14 – 20 are set up correctly.

Then, reflect on the following:

- ☐ Why was inheritance helpful when coding this project?
- ☐ Why might the **globals.gd** script be responsible for the `stove_is_empty` variable but not the `plate_is_empty` variable?
- ☐ Why might `return` sometimes be used without returning a value?
- ☐ What project did you most enjoy in Platinum Belt? Why?