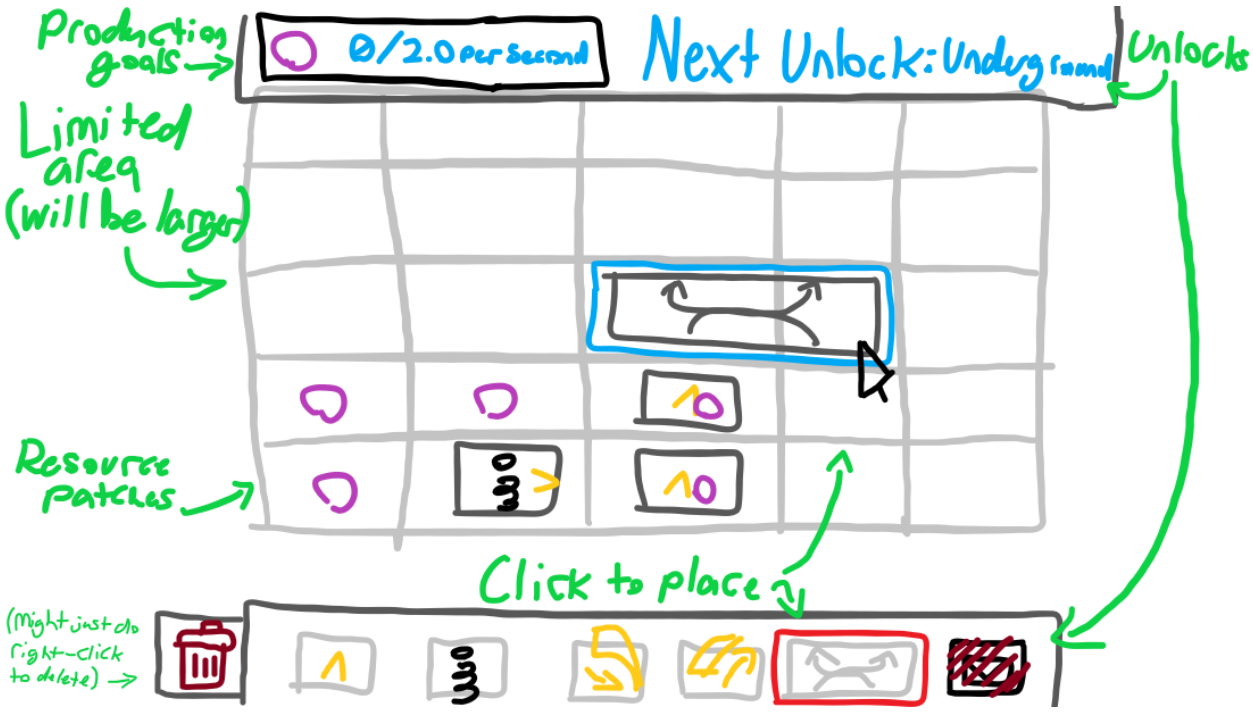
Factory Builder

Evan Daveikis 991 721 245



Limited area to build a factory to produce the maximum output

Reaching certain "goals" (units of output per second) unlocks new buildings

Buildings (order of unlock): Outbox, miner, conveyor, assembler/combiner, underground conveyor, splitter

Toolbar to place buildings at the bottom

UI for stats (output rates, goals, building unlocks) at the top

See Assignment/Planning/Planning.txt for more details

Milestones:

1. Get the world and world tiles working (placement, graphics, etc)

2. Create the building system (buildings made of tiles, temporary building placement)

3. Get placement working (toolbar, removing buildings, building rotation)

4. Complete the Products system

5. Get conveyors, miners, etc working (simple subclasses)

6. Get assemblers working (recipes, recipe select menu, etc)

7. Get goals, hud, building unlocking working

8. Polish

World.cs
- Stores the state of the world (placed buildings, where resources are, etc)
- Creates the WorldTile objects
- No other components - empty GameObject
- Pseudocode:
  - Variables: worldTilePrefab, worldTileSize, worldSize (vec2int), resourceLocations, worldTiles, buildings, tileToBuilding
  - Start => create world tiles, store in worldTiles dict
  - fn CanPlaceBuilding(position, building) => returns true if the necessary tiles are clear
  - fn PlaceBuilding(position, building) => checks if it can be built, adds building to list, adds building tiles to tileToBuilding dict, sets building position and sets it as active
  - fn RemoveBuilding(building) => removes building and tiles from list/dict, destroys object

WorldTile.cs
- Displays the state of one tile in the world (really just if any resources are there)
- Updates its graphics to match the state when it is created
- Has a SpriteRenderer attached
- Pseudocode:
  - Variables: spriteRenderer, tileType
  - Start => set sprite to match the tileType (get the sprites from FactoryManager)

TileDescriptor.cs
- Used to store tile information in the inspector
- Stores inputs, outputs
- Raw C# class
- Pseudocode:
  - Variables: inputs, outputs
  - fn CreateTile(building) => returns a new tile with the inputs and outputs

Tile.cs
- Represents a part of a building (occupies one WorldTile)
- Stores a list of inputs and outputs
- Notifies the building when an item it trying to enter the building
- Raw C# class
- Pseudocode:
  - Variables: inputs, outputs, building, rotation
  - Tile(building) => set reference to building, init inputs and outputs, set rotation
  - fn HasInput(direction) => returns true if we have an input in that direction (accounts for rotation)
  - fn HasOutput(direction) => returns true if we have an output in that direction (accounts for rotation)
  - fn TryGetInput(direction, out input) => returns the input if we have it
  - fn TryGetOutput(direction, out output) => returns the output if we have it

TileInput.cs
- Represents a place where an item can enter a building
- Raw C# class
- Pseudocode:
  - Variables: direction, tile, building
  - fn Init(tile) => set reference to tile and building (direction set in inspector)
  - fn CanInput(product) => returns true if the building will accept the product
  - fn Input(product) => calls the building's OnInput() fn

TileOutput
- Represents a place where an item can leave a building
- No components - raw class
- Pseudocode:
  - Variables: direction, tile, building
  - fn Init(tile) => set reference to tile and building (direction set in inspector)
  - fn CanOutput(product) => returns true if there is a place to output the item (checks buildings from World for inputs)
  - fn Output(product) => puts the product into the attached input

BuildingDescriptor.cs
- Stores the basic information of a building (size and tiles)
- ScriptableObject
- Pseudocode:
  - Variables: size, tiles (list of TileDescriptors), buildingType, sprite
  - No functions or anything else: basic data container

FactoryBuilding.cs (see subclasses at bottom)
- Meat and bones of the game - represents a building (assemblers, conveyors, anything the player can place)
- Used to mine, transport, process, and outbox items
- Has a SpriteRenderer and BoxCollider2D (sprite set in inspector)
- Stores the tiles that make up the building
- Stores a list of products that are inside the building
- Pseudocode:
  - Variables: descriptor, tiles, created, products
  - Properties: inputs, outputs, buildingType
  - virtual Start => create tiles from descriptor.tiles
  - virtual fn Place(position) => set position, set created = true
  - virtual fn Tick()
  - Update => if created call Tick() (only called once the building is actually placed)
  - virtual fn OnInput(product) => adds the product to the list
  - virtual bool WillAccept(product) => returns true by default
  - virtual fn OnMouseDown() => may open menus for certain subclasses (i.e. recipe selector for assemblers)
  - virtual fn CanBePlacedOn(worldTiles) => returns true if all tiles have no buildings on them

Product.cs
- Represents an item that is mined, produced, etc.
- Raw C# class, essentially just a ProductID and an amount
- Pseudocode:
  - Variables: id, amount
  - Not much else for it to do...

ProductObject.cs
- Used to display a Product in the world
- Has a SpriteRenderer
- Pseudocode:
  - Variables: productID
  - Start => get SpriteRenderer, set its sprite using the list in FactoryManager

Recipe.cs
- Stores what inputs will produce what outputs in an assembler
- ScriptableObject
- Pseudocode:
  - Variables: inputs, outputs, craftingTime
  - Not much else, maybe one util function
  - fn CanBeCraftedBy(assembler) => returns true if the assembler's inventory has all required components
  - ^^^ but that function would likely be in the assembler itself, single responsibility and all

Goal.cs
- Represents a production goal needed to unlock the next building
- ScriptableObject
- Pseudocode:
  - Variables: products, unlockedBuildingType
  - Nothing else really

Toolbar.cs
- Shows building icons for players to click to place buildings
- More buildings appear as they are unlocked
- Will have another small script that handles clicks and sends that info to this script
- Pseudocode:
  - Variables: buildingButtons, buildingPlacer
  - Start => EnableCurrentlyUnlockedBuildings()
  - fn EnableCurrentlyUnlockedBuildings() => turn on the buttons that are unlocked
  - fn BuildingButtonPressed(buildingType) => call buildingPlacer.StartPlacement(buildingType)

BuildingPlacer.cs
- Code used to handle the placement of buildings
- Shows the "ghost" of the building to be placed
- Verifies that there is room for the building
- Pseudocode:
  - Variables: currentGhost
  - fn StartPlacement(buildingType) => set currentGhost to building prefab from FactoryManager dict
  - Update => if ghost != null: move ghost to mouse pos, check for left click (TryPlaceGhost()), check for right click (destroy ghost/cancel)
  - fn TryPlaceGhost() => Check World.CanPlaceBuilding, then call World.PlaceBuilding if it is possible

HUD.cs
- Shows the current production goals and what building it will unlock
- Has various UI components (text elements for production stats, icons for building sprites, etc)
- Pseudocode:
  - Variables: nextUnlockImage, currentGoalText[], nextUnlockText, currentGoal
  - static fn SetCurrentGoal(goal) => update currentGoal, nextUnlockText, nextUnlockImage
  - static fn SetProduct(product) => update the corresponding currentGoalText (check the currentGoal to get the index)

FactoryManager.cs
- Manages lists and data for other scripts (sprites, ScriptableObjects, goals, etc)
- Pseudocode:
  - Variables: goals, tileSprites, buildings, productSprites, recipes, instance, [product counts]
  - Awake => set singleton instance
  - static fn OnProductOutboxed(product) => increment appropriate counts
  - Needed functions tbd as development progresses

RecipeSelectMenu.cs
- Shows available recipes for the assembler
- Pops up when the assembler is clicked
- Unique instance of menu per assembler (part of prefab, makes it a bit easier)
- Will have another small script that handles clicks and sends that info to this script
- Pseudocode
  - Variables: target, menu
  - Start => get reference to target (Assembler script on same object)
  - fn Open() => turn the menu GameObject on
  - fn RecipeChosen(recipe) => set target's current recipe to the clicked recipe and close the window (call Close())
  - fn Close() => turn the menu off

SUBCLASSES

Outbox, miner, conveyor, assembler/combiner, underground conveyor, splitter

Virtual fns:
-Start
-Place(position)
-Tick()
-OnInput(product)
-WillAccept(product)
-OnMouseDown()
-CanBePlacedOn(worldTiles)

Outbox.cs
- 3x3 tiles
- Inputs on all sides
- Adds to the FactoryManager product counts
- Pseudocode:
  - override fn OnInput(product) => FactoryManager.OnProductOutboxed(product)

Miner.cs
- 2x2 tiles
- 1 output
- Outputs the resource it is placed on
- Has a SpriteRenderer for showing the output
- Pseudocode:
  - Variables: mineDelay, outputSpriteRenderer
  - override fn Place(position) => base + start Mine coroutine, set outputSpriteRenderer to the resource we are on
  - coroutine Mine() => infinite loop that outputs a resource and waits for mineDelay
  - override fn CanBePlacedOn(worldTiles) => return base fn + check if at least one tile has a resource

Conveyor.cs
- 1 tile
- 1 output, 3 inputs
- Just transports an item forward
- References a SpriteRenderer to display the item
- Pseudocode:
  - Variables: transportDelay, itemSpriteRenderer
  - override fn WillAccept(product) => returns true if there are no current products

  - override fn OnInput(product) => base + starts TransportProduct coroutine
  - virtual coroutine TransportProduct(product) => sets itemSpriteRenderer using FactoryManager dict, waits for GetTransportTime(), waits for valid output, outputs product, remove product from products list, sets itemSpriteRenderer to nothing
  - virtual fn GetTransportTime() => returns transportDelay by default
  - virtual fn HasValidOutput() => returns our output's CanOutput() by default

Assembler.cs
- 2x2 tiles
- 1 output, 4 inputs
- Recipe can be set with a GUI
- Takes in items and creates a new item according to the recipe
- Has SpriteRenderers for the current recipe's inputs and output
- Has an attached RecipeSelectMenu
- Pseudocode:
  - Variables: currentRecipe, inputSprites, outputSprite, const_maxIngredientMultiplier, craftCoroutine
  - fn UpdateRecipe(newRecipe) => set currentRecipe to newRecipe, update recipe sprites (input and output), destroy current products that dont match the new recipe (wrong type or too many), stop the craftCoroutine if it is running
  - override fn WillAccept(product) => returns true if the currentRecipe uses that product and there isnt too much of that product (multiply how much the current recipe uses by const_maxIngredientMultiplier)
  - override fn OnInput(product) => base + check if we have enough products to start crafting - if so start the coroutine
  - coroutine Craft() => waits for the currentRecipe's craft time, waits for valid output, outputs product, removes used products from products list

UndergroundConveyor.cs
- 1 tile per end (input and output)
- Input has an input, output has an output (shocking) and inputs on the sides
- Allows products to pass under other buildings and conveyors
- Split into two buildings (unlocked at the same time)
- Input end inherits from Conveyor with different behaviour for transporting (has no output)
- Output end inherits from Conveyor (standard conveyor)
- Input end overrides:
  - Variables: transportDelayPerTile, maxTilesBetweenEnds

  - override fn GetTransportTime() => gets the current output (GetOutput(out distance)), and multiplies the distance by transportDelayPerTile
  - override fn HasValidOutput() => returns whether or not GetOutput() is null and whether that output is accepting input
  - fn GetOutput(out distance) => walks forward for maxTilesBetweenEnds + 1 (to include the output), checking for an output tile facing the correct direction. returns it if found, null otherwise

Splitter.cs
- 2x1 tiles
- 2 inputs at the back, 2 outputs at the front
- Swaps which output it uses each time
- Pseudocode:
  - Variables: currentOutput
  - override fn WillAccept(product) => returns true if there are no current products
  - override fn Tick() => if we currently have a product stored, call Output(), removing the product from the products list if successfully output
  - override fn OnInput(product) => calls Output(), calling the base function to stores the product if it returns false
  - bool Output() => calls TryOutput(), and if output fails it SwapOutputs() and tries again. Returns whether or not the item was output successfully
  - bool TryCurrentOutput(product) => checks if the current output can output, outputting the product, swapping the output and returning true, if output is possible. Returns false otherwise
  - fn SwapOutputs() => changes the currentOutputIndex from 0 to 1 and vice versa