Classic City Builder Kit

SoftLeitner

read manual online at
https://citybuilder.softleitner.com/manual

for support please contact
softleitner@gmail.com

Overview

Welcome

This is the manual for Classic City Builder Kit by SoftLeitner. Classic City Builder Kit is quite the mouthful so i will be referring to it using the shorthand CCBK.

Oldschool walker based City Builders are very systems heavy games that require lots of tweaking and balancing to get right. CCBK was created with the aim of providing all the common tropes of the genre and making them easily accessible in the Editor.

The kit was created with the classic evolve housing, build monuments type of game in mind and most of the systems are commonly seen in that genre. That being said large parts like Buildings, Items, Walkers and Layers are general enough to create a wide variety of games.

Project Structure

CCBK is separated into multiple assemblies. Which ones you need depends on how you are planning to use the Kit. For your first exploration create a new project and remove the example assets and materials, then import the entire asset.

- CityBuilderCore
  The Core Framework of CCBK.
  Always import this one.

- CityBuilderCore.Tests
  Tests some core functionality of CityBuilderCore.
  Only import this if you plan to change CityBuilderCore and want to check if the basic systems still function.

- CityBuilderThree
  Contains the 3D City Builder demo
  Import if you want to start by adapting this demo.
  For the menu to work add the Menu, StageOne, StageTwo, StageThree scenes in Build Settings.

- CityBuilderDefense
  Contains the 2D Tower Defense demo
  Import if you want to start by adapting this demo.

- CityBuilderManual
  Holds completed samples of the HowTo section of this manual.
  Import if you're stuck on a how to or exploring.

Both demos depend on either layers, tags or sorting layers. **If these settings were not set during import please copy the contents of TagManager.txt from the demo folder to the TagManager.asset in you ProjectSettings.**

Dependencies

- The demos depend on TextMeshPro (http://docs.unity3d.com/Packages/com.unity.textmeshpro@2.1/index.html) for their UI
- CCBK generally depends on 2D Tilemap Editor (https://docs.unity3d.com/Packages/com.unity.2d.tilemap@1.0/manual/index.html)

Usage

After familiarizing yourself with CCBK you should be ready to start your project. I'll outline three basic levels of usage for CCBK here. The first two don't require any coding, implementing custom systems assumes basic knowledge of C#.

- Adapting one of the Demos
  Is one of the demos close enough to what you are trying to achieve? Great!
  Import it along with CityBuilderCore and the Settings and start tweaking. The content of the demos is explained in detail in the 3D City Builder and 2D Tower Defense manual pages.

- Starting from Scratch using Core Logic
  If none of the demos gets close enough to what you want to make or you want to make sure you understand every piece of your project starting from scratch is the way to go. Continue Here for a step to step guide to creating a city builder purely in Unity Editor or get a head start using the **Setup Wizard** in Windows/City Builder/Setup.

- Implementing Custom Logic
  Need some special sauce in your game and know basic C#. Continue Here for a step to step guide to extending CCBK with your own components and systems.

Release Notes

1.1.0

ADDED

- **Setup Wizard** window
  creates a clean project template with placeable buildings
  configure display modes, axis, sizes, ... in a simple dialog
- **Expandable Buildings**
  building size is determined during building
  expand building in one or two dimensions
  showcased in the new bridge building of THREE demo
- enhanced **Full 3D** support
  showcased in new 3D demo scene in THREE demo
  support for perspective camera and 3D roads
  height parameter in walkers can modify their display
  (move along terrain or change to different layer in tunnel)
- **Road Switching**
  allows destination walkers to move between road networks
  showcased in SwitchRoadDebugging scene in tests and tunnel scene in urban demo

IMPROVED

- **Building Requirements** now support various modes and configurations
- **Road Requirements** can now be configured to amend missing roads
- **Cell Size** of grid is taken into account in default and isometric map

CHANGED

- RoadBuilding functionality has been merged into regular Buildings

FIXED

- incorrect camera area when camera was pitched enough to look up
- structure level editor not working inside arrays
- monument pathing when using worker pooling
- ...

1.0.0

ADDED

- **urban demo project**
  simple city sim in isometric 2d
  showcases various 1.0 features
- **connections** core system
  build a network of feeders and passers
  enables water and electricity networks
- **connected tiles** that switch sprite based on their neighbors
  simple base class and variants for rectangle and isometric grids

IMPROVED

- **walkers** can now be pooled using ObjectPool
- **roads** now support multiple networks
  simply use MultiRoadManager instead of DefaultRoadManager

  > *rails and roads in urban demo*

- **structures** now support multiple levels
  easily control which levels a structure inhabits using the level toggles

  > *a single point in the urban demo can have a pipe, road and power line; pumps inhabit all three levels*

- new **view** type that displays building efficiency

CHANGED

- demo materials have been switched from urp to legacy shaders
- sort axis moved to camera controller

FIXED

- blockers not working after save/load
- building walkers not triggering on spawn
- panning in xy isometric maps
- speed controls in historic demo
- ...

0.9.0

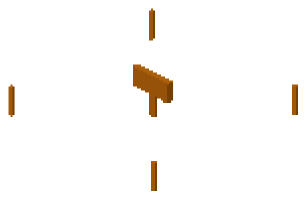
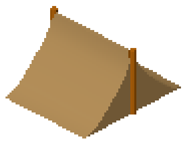
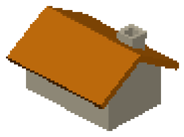
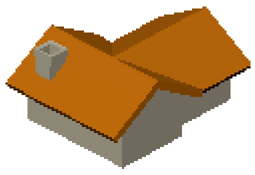
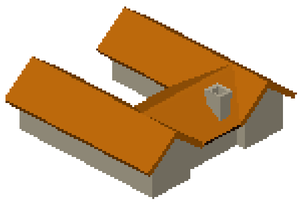
Initial Release

3D City Builder

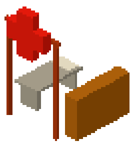
Play it here, uses pretty much every system in CCBK except attacks.

Buildings

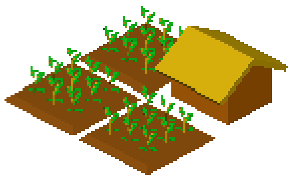
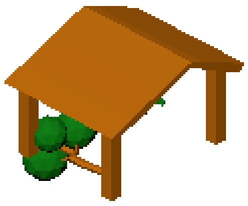
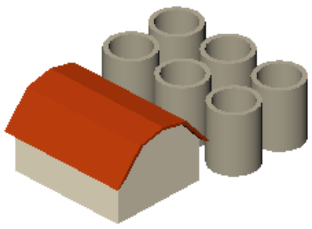
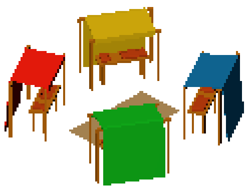
Housing

	**Property**	HousingPlaceholderComponent	Prefab: Tent
	**Tent**	HousingComponent EvolutionComponent RiskerComponent	Capacity: 5 Plebs Evolution: Housing Collapse: 1/s Disease: 3/s
	**Hut**	HousingComponent EvolutionComponent RiskerComponent	Capacity: 10 Plebs Evolution: Housing Collapse: 3/s
	**House**	HousingComponent EvolutionComponent RiskerComponent	Capacity: 15 Plebs Evolution: Housing Collapse: 3/s
	**Villa**	HousingComponent EvolutionComponent RiskerComponent GenerationComponent	Capacity: 15 Plebs Evolution: Housing Collapse: 3/s Interval: 10 Items: 20 Gold

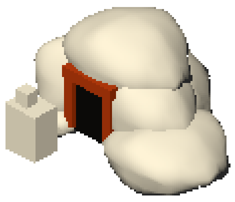
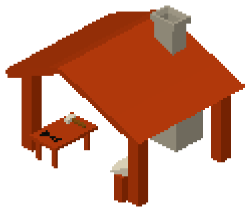
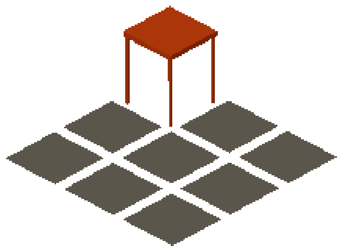
Risks&Services

	Well	EmploymentComponent ServiceWalkerComponent	Group: Services Needed: 5 Plebs Downtime: 5 Prefab: WaterWalker
	Workshop	EmploymentComponent RiskWalkerComponent	Group: Services Needed: 5 Plebs Downtime: 5 Prefab: CollapseWalker
	Firefighter	EmploymentComponent RiskWalkerComponent	Group: Services Needed: 5 Plebs Downtime: 5 Prefab: FireWalker
	**Doctor**	EmploymentComponent RiskWalkerComponent	Group: Services Needed: 5 Plebs Downtime: 5 Prefab: DiseaseWalker
	Treasurer	EmploymentComponent CollectionComponent	Group: Services Needed: 5 Plebs Downtime: 5 Prefab: TreasureWalker Storage: Global

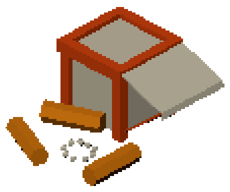
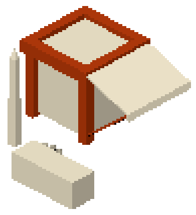
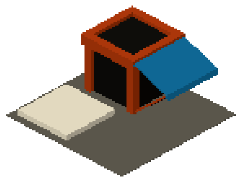
Food

	**Farm**	EmploymentComponent ProductionComponent LayerEfficiencyComponent	Group: Food Needed: 15 Plebs Interval: 10 Produces: 100 Potatoes Layer: Water Min: 0.2 Max: 100
	**Hunter**	EmploymentComponent ItemsRetrieverComponent	Group: Food Needed: 10 Plebs Retriever: HunterWalker Downtime: 10
	**Silo**	EmploymentComponent StorageComponent	Group: Logistics Needed: 10 Plebs Storage: Stacked StackCount: 6 Capacity: 4 Orders: Accept Potato, Meat
	**Market**	EmploymentComponent DistributionComponent	Group: Services Needed: 10 Plebs Storage: Unit Capped Capacity: 5 MinimumOrder: 1 Orders: Fill Potato, Meat

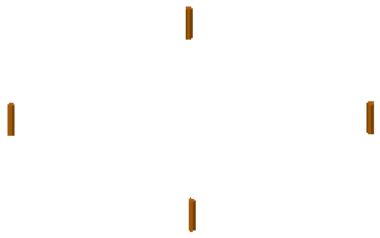
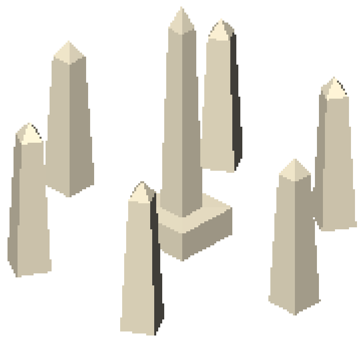
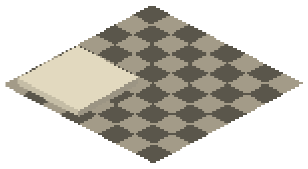
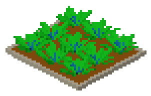
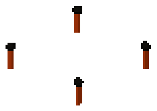
Production

	Iron Mine	EmploymentComponent ProductionComponent	Group: Production Needed: 10 Plebs Interval: 20 Produces: 1 Iron
	**Granite Mine**	EmploymentComponent ProductionComponent	Group: Production Needed: 10 Plebs Interval: 20 Produces: 1 Granite
	**Smith**	EmploymentComponent ProductionComponent	Group: Production Needed: 10 Plebs Interval: 10 Consumes: 1 Iron Produces: 10 Tools
	**Yard**	EmploymentComponent StorageComponent	Group: Logistics Needed: 10 Plebs Storage: Stacked StackCount: 8 Capacity: 4 Orders: Accept Iron, Tools, Granite Refuse Potato, Meat

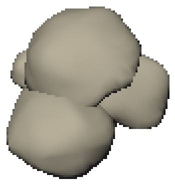
Workers

	**Labour Camp**	CyclicWorkerProviderComponent ItemEfficiencyComponent	Prefab: LabourWalker Downtime: 5 Item: Tools Interval: 10
	**Mason Camp**	PooledWorkerProvider	Prefab: MasonWalker Count: 2 Downtime: 3
	**Entertainer Camp**	CyclicWorkerProviderComponent RiskerComponent LayerAffector	Prefab: EntertainerWalker Risk: Fire Increase: 5/s Layer: Desirability Value: -10 Range: 3 Falloff: -3

Other

	**Obelisks Site**	MonumentSiteComponent	Stages: CirclyPits, CenterPit, CircleObelisk, CenterObelisk various steps with different workers per stage
	**Obelisks**		
	**Stage**	WorkerUserComponent LayerAffectorWorking	Worker: Entertainer Quantity: 1 Queue: 2 Duration: 20 Layer: Entertainment Value: 100 Range: 10
	**Garden**	LayerAffector	Layer: Desirability Value: 10 Range: 3 Falloff: 3
	**Road Blocker**	HousingComponent	Health: 100

Structures

	Tree	StructureCollection LayerAffector	Key: TRE Destr: 1 Deco: 0 Walk: 0 Size: 1 Layer: Desirability Value: 5 Range: 2 Falloff: 2
	Ore	StructureCollection LayerAffector	Key: ORE Destr: 0 Deco: 0 Walk: 0 Size: 2 Layer: Iron Value: 10 Range: 1
	**Rubble**	StructureCollection	Key: RUB Destr: 1 Deco: 1 Walk: 0 Size: 1

Walkers

Migration

ImmigrationWalker	ImmigrationWalker	Path: Map Capacity: 5
EmigrationWalker	EmigrationWalker	Path: Map Capacity: 5
HomelessWalker	HomelessWalker	Path: Road Capacity: 1

Roaming

WaterWalker	ServiceWalker	Path: RoadBlocked Service: Water Amount: 100/s
CollapseWalker	RiskWalker	Path: RoadBlocked Risk: Collapse Amount: 100/s
DiseaseWalker	RiskWalker	Path: RoadBlocked Risk: Disease Amount: 100/s
SickWalker	RiskWalker	Path: Road Risk: Disease Amount: -10/s
FireWalker	RiskWalker	Path: RoadBlocked Risk: Fire Amount: 100/s
TreasureWalker	CollectionWalker	Path: RoadBlocked Items: Gold
SaleWalker	SaleWalker	Path: RoadBlocked Storage: 1 Unit
EmploymentWalker	EmploymentWalker	Path: RoadBlocked Return: 1

Logistics

DeliveryWalker	DeliveryWalker	Path: Road Storage: 1 Unit Return: 1
StorageWalker	StorageWalker	Path: Road Storage: 1 Unit Return: 1
PurchaseWalker	PurchaseWalker	Path: Road Storage: 2 Unit

Workers

LabourWalker	WorkerWalker	Path: Road Worker: Laborer Storage: 1 Unit
MasonWalker	WorkerWalker	Path: Road Worker: Mason Storage: 1 Unit
EntertainerWalker	WorkerWalker	Path: Road Worker: Entertainer Storage: 1 Unit

Items

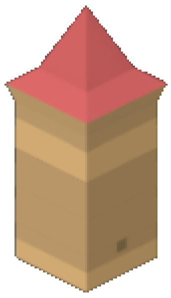
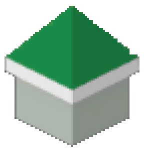
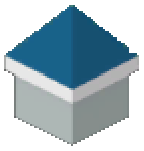
	Gold	Key: GLD Unit Size: 1
	Potato	Key: PTT Unit Size: 100
	Meat	Key: MEA Unit Size: 100
	Iron	Key: IRN Unit Size: 1
	Tools	Key: TOL Unit Size: 10
	Granite	Key: GRT Unit Size: 1

2D Tower Defense

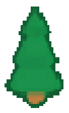
Art assets have been adapted from packs made by Kenney (https://kenney.nl)

Play it here, primarily uses the attacks and items retriever resource systems.

Buildings

	**Center**	AttackableComponent	Health: 100
	Tower	DefenderComponent	Distance: 2 Cooldown: 1.5 Damage: 2
	**Lumberjack**	ItemsRetrieverComponent	Storage: Global Walker: LumberjackWalker
	**Stone Quarry**	ItemsRetrieverComponent	Storage: Global Walker: StoneQuarryWalker

Structures

	Walls	StructureTiles	Key: STO Replicates to WallObstacles
	**Trees**	StructureCollection Tree > ReloadingItemsDispenser	Key: TRE Prefab: Tree Key: TRE Charges: 1 Reload: 5
	Rocks	StructureCollection Rock > ReloadingItemsDispenser	Key: ROK Prefab: Rock Key: STO Charges: 1 Reload: 5

Walkers

	AttackWalker	AttackWalker	Path: MapGrid Speed: 3 Health: 10 Damage: 5 Rate: 5
	LumberjackWalker	ItemsRetrieverWalker	Path: Map Speed: 2 Distance: 10 DispenserKey: TRE
	StoneQuarryWalker	ItemsRetrieverWalker	Path: Map Speed: 2 Distance: 10 DispenserKey: STO

Items

	Stone
	Wood

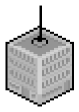
2D City Sim

Art assets have been adapted from sprites made by opengameart user pixel32 (https://opengameart.org/users/pixel32)

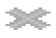
Play it here.

Demonstrated Connections, Multi-Road-Networks and Multi-Level-Structures.

Buildings

	Railstation	RailwayComponent	Goods	OXX
	House	ConnectionPasserComponent LayerEfficiencyComponent HouseComponent	Power Water Van	OXX
	**Shop**	ConnectionPasserComponent LayerEfficiencyComponent ShopComponent	Power Water Money	OXX
	Water Pump	ConnectionPasserComponent ConnectionFeederComponent LayerEfficiencyComponent	Power Water Water	XXX
	Power Station	ConnectionFeederComponent	Power	OXX

Structures

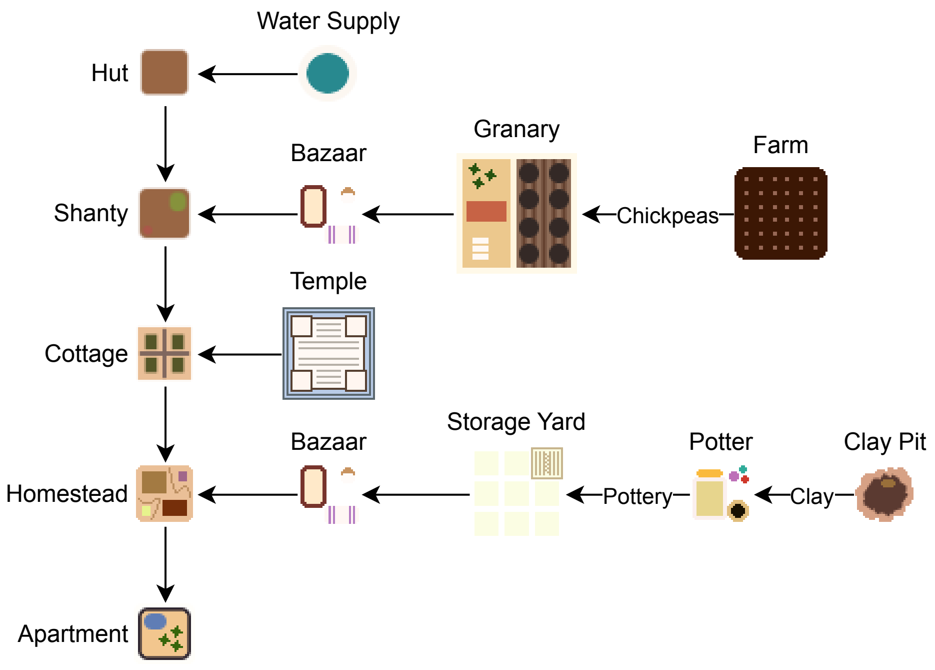
	Power Lines	StructureTiles ConnectionPasser	POW Power	OOX
	**Pipes**	StructureTiles ConnectionPasser	PIP Water	XOO

Walkers

	Train	TrainWalker	Rail
	Train	TruckWalker	Road 100 Goods
	Train	VanWalker	Road 5 Goods

Start from Scratch

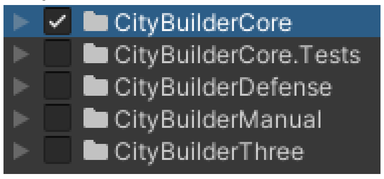
Throughout the following steps we will create a very simple oldschool city builder from scratch using CCBK. Please forgive the horrible programmer art.



This page covers only the first two evolutions, water and food. A completed version with all stages can be found in the CityBuilderManual project. I believe you will be able to figure the rest out after following the steps laid out here. Religion is very similar to water and pottery has just a slightly longer production chain than food.

Setup

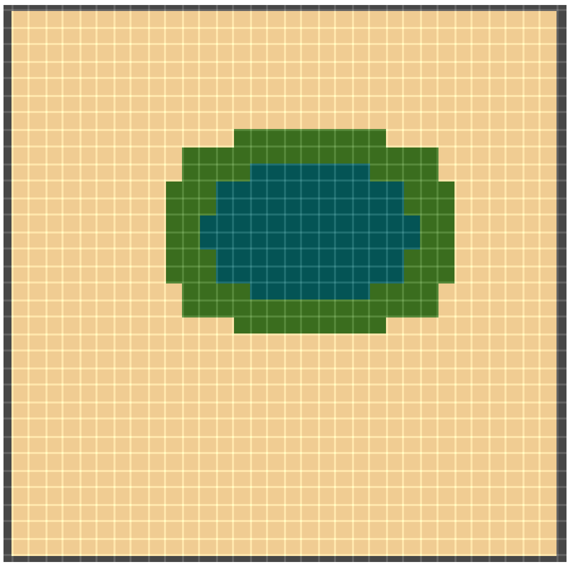
- Create a new unity project using the 2D template
- Import the CCBK Core Framework



- Place the Manual Assets (http://localhost:4000/assets/downloads/manualAssets.zip) inside your Assets folder

Map

- Create a Gameobject at 0,0,0 and call it 'Map'
- Add a Grid Component
- Add a DefaultMap Component [1]
  - Set its Size to 32x32
  - In Walking- and BuildingBlockingTiles add the water tile [2]
  - Create a child Gameobject at 0,0,0 and call it 'Ground'
    - Add Tilemap and TilemapRenderer to it
  - Assign it to the Ground field on your map
- Move the Main Camera Object to the center of the map (16,16,-10)
  - Add a Camera Controller Component to it
- Open the Tile Palette Tab found in Windows/2D
  - Make sure your active Tilemap is 'Ground' and your Palette is 'historic'
  - Draw your map inside the guide lines of the map(turn on gizmos if you cant see it)



We'll later add a water layer that spreads out two tiles from water. I' ve drawn grass tiles there to communicate that to the player.

With all that done you can start the game for the first time. You should be able to move the Camera around the map while being confined to its bounds.

Roads

Roads are generally pretty simple but we'll have to do some setup on the Tools and Logic side to be able draw them so please bear with me.

Tilemap

- Create a child Gameobject in Map at 0,0,0 and call it 'Roads'
  - Add Tilemap and TilemapRenderer to it
    - Set OrderInLayer to 1
  - Add a DefaultRoadManager Component
- Create a child Gameobject in Map at 0,0,0 and call it 'Highlights'
  - Add Tilemap and TilemapRenderer to it
    - Set OrderInLayer to 2
  - Add a DefaultHighlightManager to it
    - Assign the Highlight Tiles of the same name

Logic

- Create a Gameobject at 0,0,0 and call it 'Logic' and add the following Components
    - DefaultGameManager
    - ObjectRepository
    - DefaultStructureManager

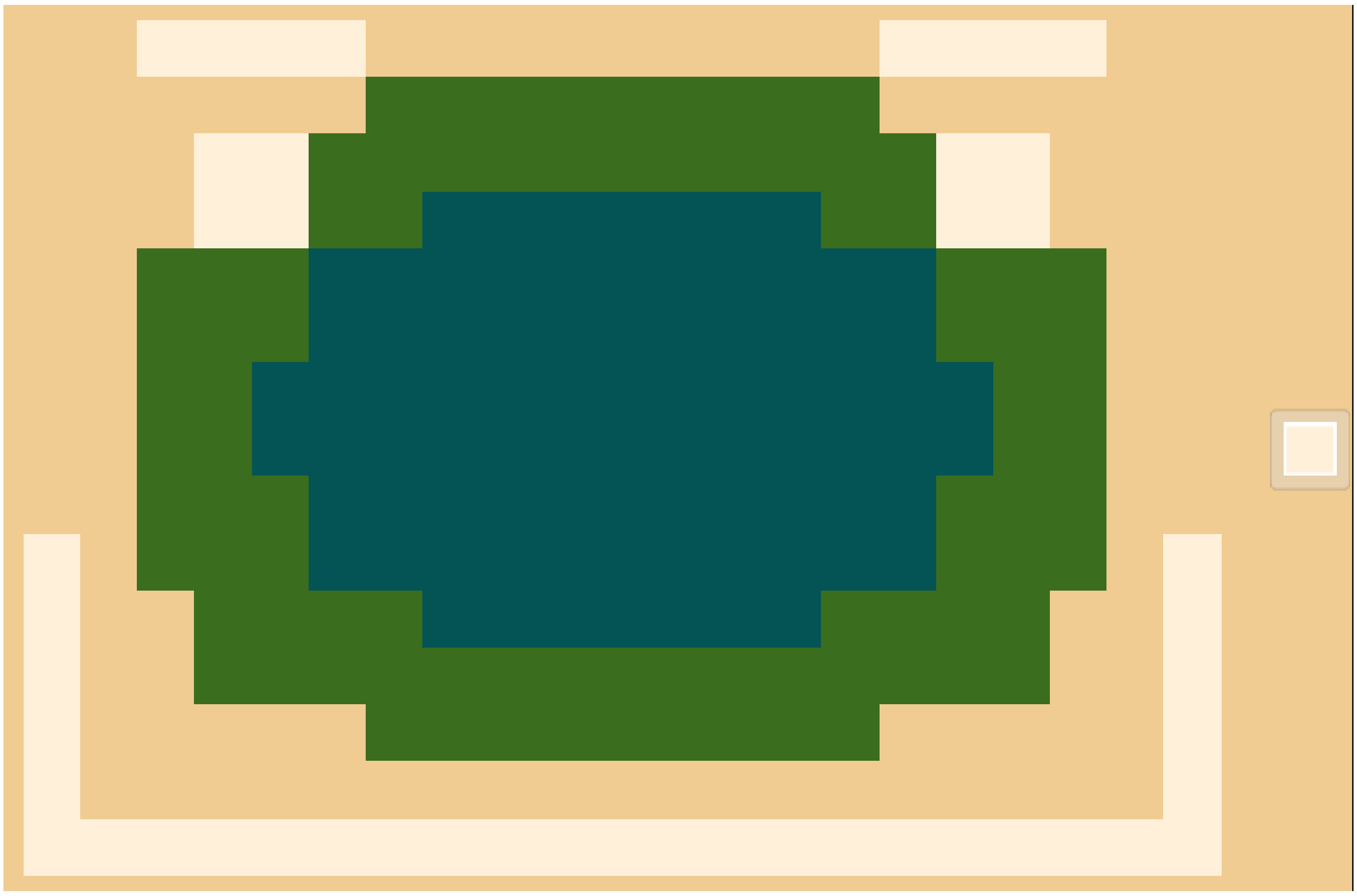
Road Object

- Create a Folder in your Assets called City
    - Create a Road Object
      Found in the ContextMenu under Create/CityBuilder/Road
    - Add a single stage to it by assigning 1 to Stages-Size
        - Assign 'ROD' to the Key [3]
        - Assign the 'road' tile to Tile
- On ObjectRepository assign Road

UI Tools

- Create a UI/Panel and call it 'Tools'
    - Move it to middle-right with a width of 50
    - Add a VerticalLayoutGroup and a ContentSizeFitter with verticalFit=Preferred
    - Add a ToggleGroup with AllowSwitchOff=1 and a ToolsManager Component
        - Assign the ToggleGroup in the ToolsManager
    - Create a UI/Toggle in Tools and call it 'RoadTool'
        - Delete the Text
        - Resize the Toggle to 50,50
        - Assign the ToggleGroup
        - Set the Toggles IsOn to false
        - Move Background and Checkmark to middle-center with size 32,32
        - Assign toolsBorder as the Checkmark image and blank as the Background
        - Add a RoadBuilder and ToolsActivator Component to RoadTool
            - Assign the Road Object you just created in the Road field
        - Add ToolsActivator.SetToolActive to the Toggles OnValueChanged event

After selecting the tool and starting up the game you should now be able to draw roads on the map.



Buildings

Prefab

- Create a Gameobject at 0,0,0 and call it 'WaterSupply' [4]
    - Create a 2D Object/Sprite at 1,1,0 and call it 'sprite'
        - Assign the waterSupply sprite and OrderInLayer 10
    - Add a Building Component to WaterSupply and assign sprite as its Pivot
- Create a Folder called 'Buildings' inside the City Folder
    - Drag the WaterSupply to the Folder to create a Prefab
    - Remove the WaterSupply from the Scene

Buildinginfo

- Create a BuildingInfo Object called 'WaterSupply' in Buildings
  Found in the ContextMenu under Create/CityBuilder/BuildingInfo)
    - Assign Key='WAT' Name='water supply' and Size=2,2
    - Assign the Prefab we just created
- On the WaterSupply Prefab assign the BuildingInfo

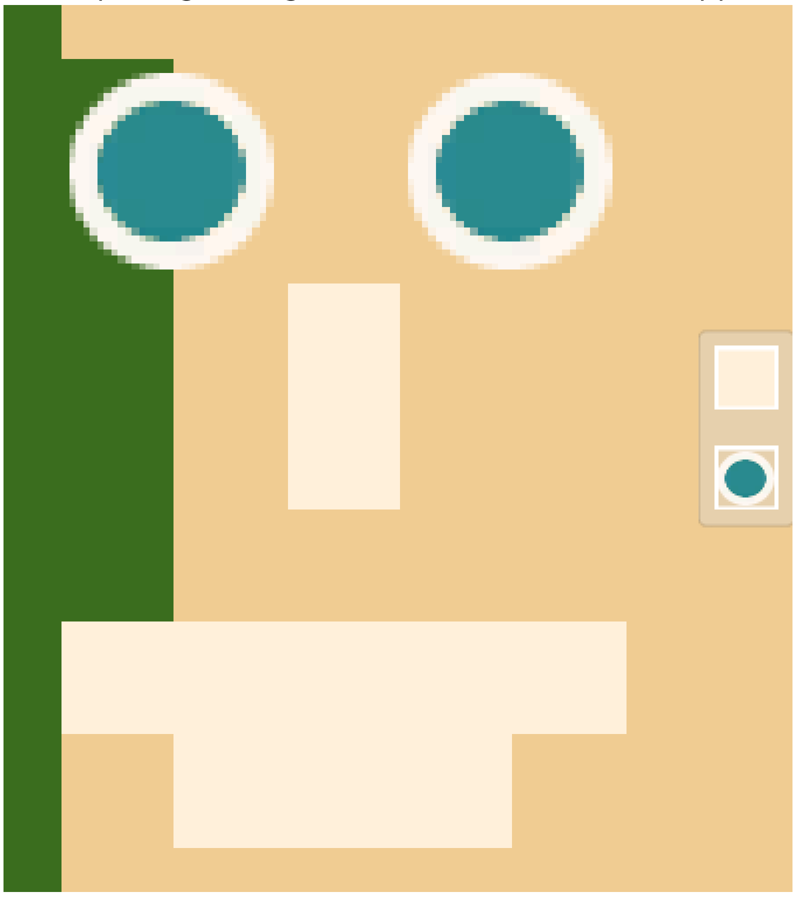
Tool

- Duplicate the RoadTool we've created in the previous chapter and call it 'WaterTool'
    - Replace the RoadBuilder with a BuildingBuilder [5]
        - Assign the WaterSupply BuildingInfo to the BuildingBuilder
    - Set the Background image to waterSupply and reset the color to white

Logic

- Add a DefaultBuildingManager to Logic

Start up the game again and build some water supplies!



Walkers

Prefab

- Create a Gameobject at 0,0,0 and call it 'WaterCarrier'
    - Create a child Gameobject at 0.5,0.5,0 and call it 'pivot'[6]
        - Create a 2D Object/Sprite at 0,0,0 and call it 'sprite'
            - Assign the waterCarrier sprite and OrderInLayer 20
            - Set Z Rotation to 270
    - Add a ServiceWalker Component to WaterCarrier and assign pivot as its Pivot
        - Set PathType to Road
- Create a Folder called 'Walkers' inside the City Folder
    - Drag the WaterCarrier to the Folder to create a Prefab
    - Remove the WaterCarrier from the Scene

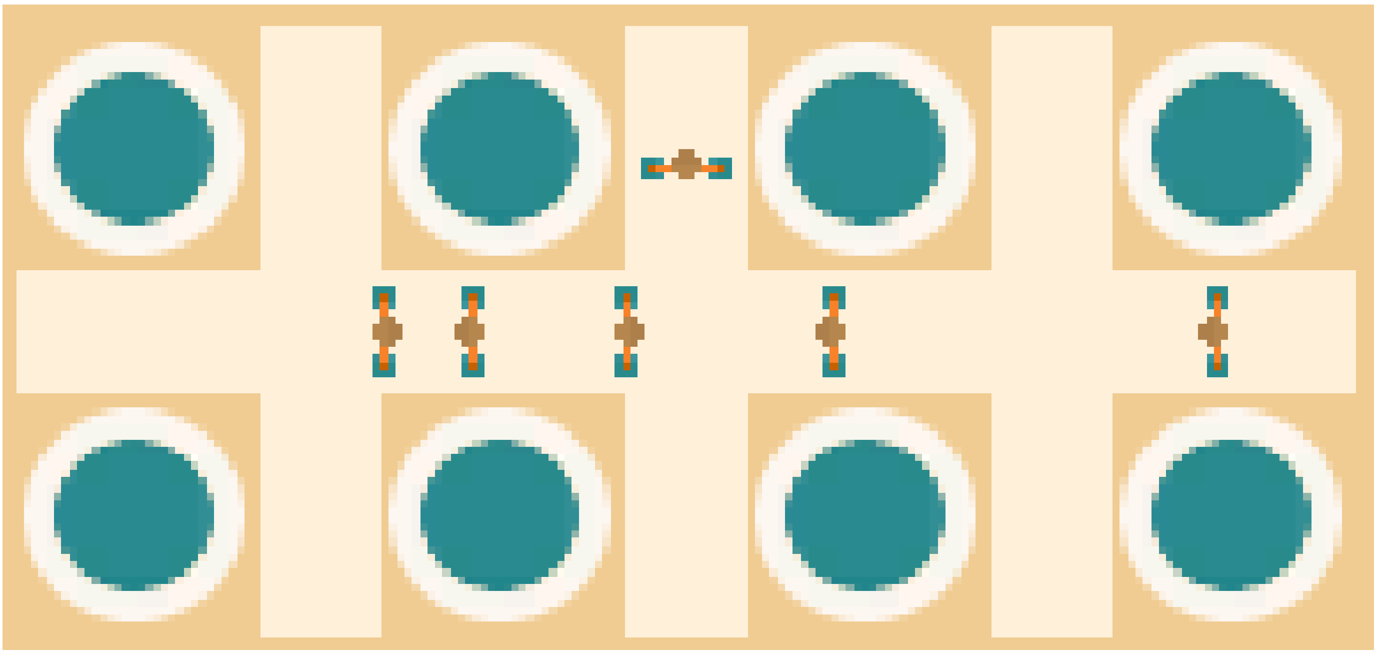
Walkerinfo

- Create a WalkerInfo Object called 'WaterCarrier' in Walkers (found in the ContextMenu under Create/CityBuilder/WalkerInfo)
    - Assign Name='water carrier'
- On the WaterCarrier Prefab assign the WalerInfo

Spawning

- On the WaterSupply Prefab add a ServiceWalkerComponent Component[7]
    - Assign Count=1 Reuse=1 Downtime=5

As soon as a WaterSupply is connected to a road it should now spawn a WaterCarrier that roams around for a bit and then returns.



- Change Variance on the Map to 0.2 and start the game again, walkers should now be offset a little bit to prevent overlapping

Templates

Since we will be creating a number of Buildings and Walkers in the following Chapters let’s create blank templates for them that we can easily duplicate.[4]

- Duplicate WaterSupply Prefab and Info
  - rename them both to ‘Building’
  - clear out key, name and prefab in info
  - clear out info and sprite in prefab
  - remove the ServiceWalkerComponent
- Duplicate WaterCarrier Prefab and Info
  - rename them both to ‘Walker’
  - clear out name in info
  - clear out info and sprite in prefab
  - remove the ServiceWalker

From now on whenever i mention creating a new building do the following.

- Duplicate Building Prefab and Info
- Set prefab, key and name in info
- Set info and sprite in prefab

Whenever i mention creating a new walker do the following.

- Duplicate Walker Prefab and Info
- Add whatever Walker Component is needed
- Set name in info
- Set info and sprite in prefab

Housing

Population Object

- Create a Population Object in your City Folder
  Found in ContextMenu under Create/CityBuilder/Population
    - Set Key='POP' Name='Population'
- On ObjectRepository assign Population

Buildings

- Create Buildings

name	info-key	info-name	sprite
H0	HNO	housing	housing
H1Hut	HUT	hut	hut
H2Shanty	SHT	shanty	shanty
H3Cottage	CTG	cottage	cottage

- Add a HousingPlaceholderComponent to H0
    - Assign H1 as the Prefab
- Add a HousingComponent to H1,H2,H3
    - In PopulationHousings add a single entry that uses the Population Object and the counts 20, 35, 50

Walkers

- Create Immigrant Walker
    - Add ImmigrationWalker Component
        - Name: immigrant Sprite: migrant
        - Set PathType to MapGrid
        - Set Capacity to 10

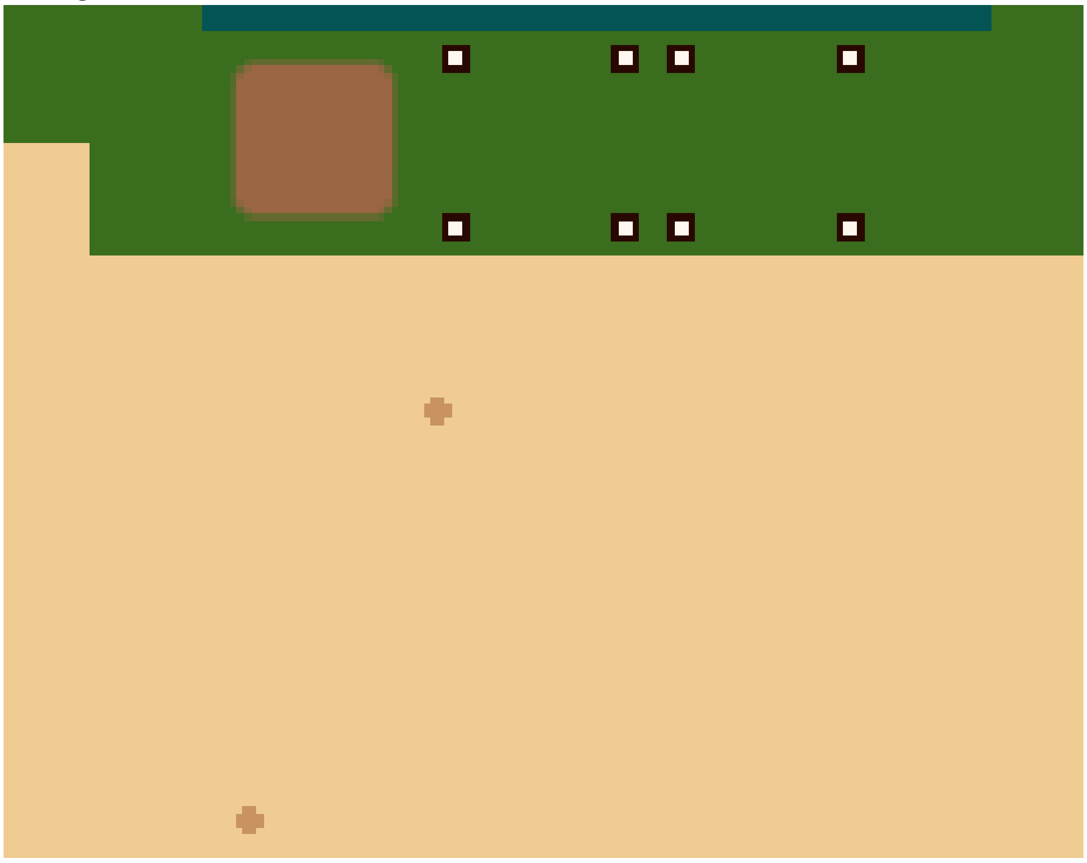
Logic

- Add DefaultPopulationManager to Logic
- Add a Migration to Logic
    - Assign the ImmigrationWalker and set the count to 100
    - Set Logic as the Entry
    - Assign the Population Object

Tool

- Duplicate the WaterTool and name it HousingTool
- Assign H0 as the Building
- Use the hut sprite as the Background
- Move it between RoadTool and WaterTool

Start the game, the HousingTool now builds placeholders that are turned into huts as soon as immigrants arrive.



Food

Items

- Create a Folder called ‘Items’ inside the City Folder
- Create an Item Object in the Items Folder
  Found in ContextMenu under Create/CityBuilder/Item
    - Set Key=’CHK’ Name=’chickpeas’ UnitSize=100 Icon=chickpeas
- On ObjectRepository assign Item

- Create a Gameobject at 0,0,0 and call it ‘Chickpeas’
    - Set scale to 0.65,0.65,1
    - Create 4 Sprites with OrderInLayer 30 and assign the chickpeas1-4 sprites
    - Add a StorageQuantityVisual
        - Set Swap to true
        - Add the sprites to the Objects array
- Drag Chickpeas to the Items Folder and remove it from the scene
- Add it to the Chickpeas-Item Visuals

Walkers

name	info-name	sprite	component	settings
CartPusher	cart pusher	cartPusher	DeliveryWalker	PathType:Road
BazaarBuyer	bazaar buyer	bazaarBuyer	PurchaseWalker	PathType:Road Capacity:3
BazaarSeller	bazaar seller	bazaarSeller	SaleWalker	PathType:RoadBlocked

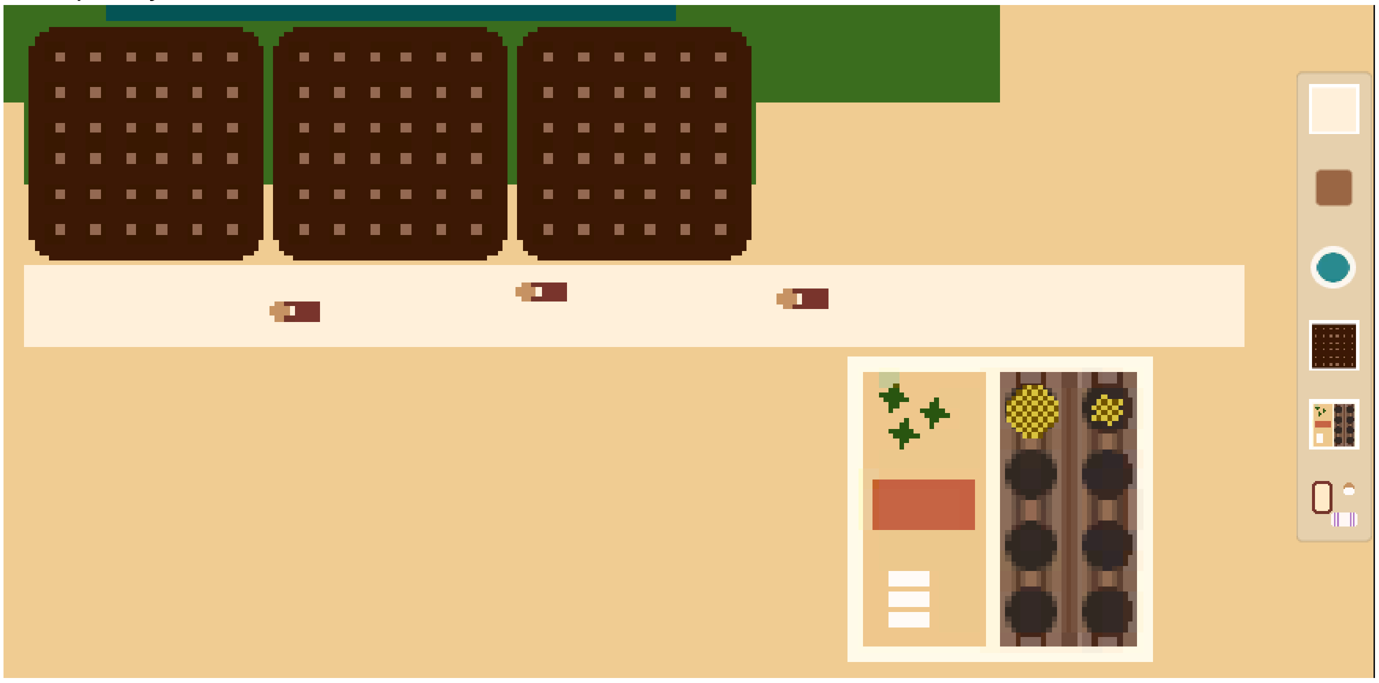
Buildings

name	info-key	info-name	sprite
Farm	FAM	farm	farm
Granary	GRN	granary	granary
Bazaar	BAZ	bazaar	bazaar

- Granary and Farm are larger buildings so we'll have to adjust size and pivot position
    - Farm size:3x3 pivot:1.5,1.5,0
    - Granary size:4x4 pivot:2,2,0
- Farm
    - Add a ProductionComponent
        - Set Interval to 20
        - Assign the CartPusher as Prefab in DeliveryWalkers
        - Create an entry in ItemsProducers
            - Set Item to Chickpeas and Quantity to 100
            - Set Storage to FreeUnityCapped with Capacity:1
    - Add a ProgressThresholdVisualizer
        - Create 4 Sprites under sprite with OrderInLayer:100 and assign farmProgress1-4 sprites
        - Create 4 entries in ProgressThresholds with the above sprites and the thresholds: 0.2, 0.4, 0.6, 0.8
- Granary
    - Add a StorageComponent
        - Set Storage to Stacked Mode with 8 Stacks and 4 Capacity
        - Create an entry in Orders for Chickpeas with Ratio:1
    - Add a StorageQuantityVisualizer to Granary
        - Create 8 Transforms and move them over the centers of the holes in the sprite
        - Add them to the Origins array
- Bazaar
    - Add a DistributionComponent
        - Set Storage to FreeUnitCapped with Capacity 5
        - Create an entry in Orders for Chickpeas with Ratio:1
        - Assign the BazaarBuyer as Prefab in PurchaseWalkers
        - Assign the BazaarSeller as Prefab in SaleWalkers with Reuse:1 and Downtime:5

- Add DefaultItemManager to Logic and set Storage to Free Mode
- Duplicate the WaterTool 3 times and assign the sprites and BuildingInfos of Farm, Granary and Bazaar

Build some farms and watch them grow. Connect them to a granary for the chickpeas to be stored there. The Bazaar wont do anything because the seller wont find anyone in need of chickpeas yet.



Layer

- Create a Layer Object in the City Folder and name it Water
  Found in ContextMenu under Create/CityBuilder/Layer
- On Logic Add DefaultLayerManager
  - Create an Entry in AffectingTiles
    assign the water tile, water layer and set value:10 Range:2 Falloff:0
- In the BuildingInfos of Farm and WaterSupply add a LayerRequirement entry
  - Set Layer:Water Min:5 Max:100

Farms and WaterSupplies can now only be built close to water.

Evolution

Water Service

- Create a Folder called 'Services' inside the City Folder
- Create a Service Object in the Services Folder and name it Water
  Found in ContextMenu under Create/CityBuilder/Service
- In the WaterCarrier Walker Prefab assign the Water Service

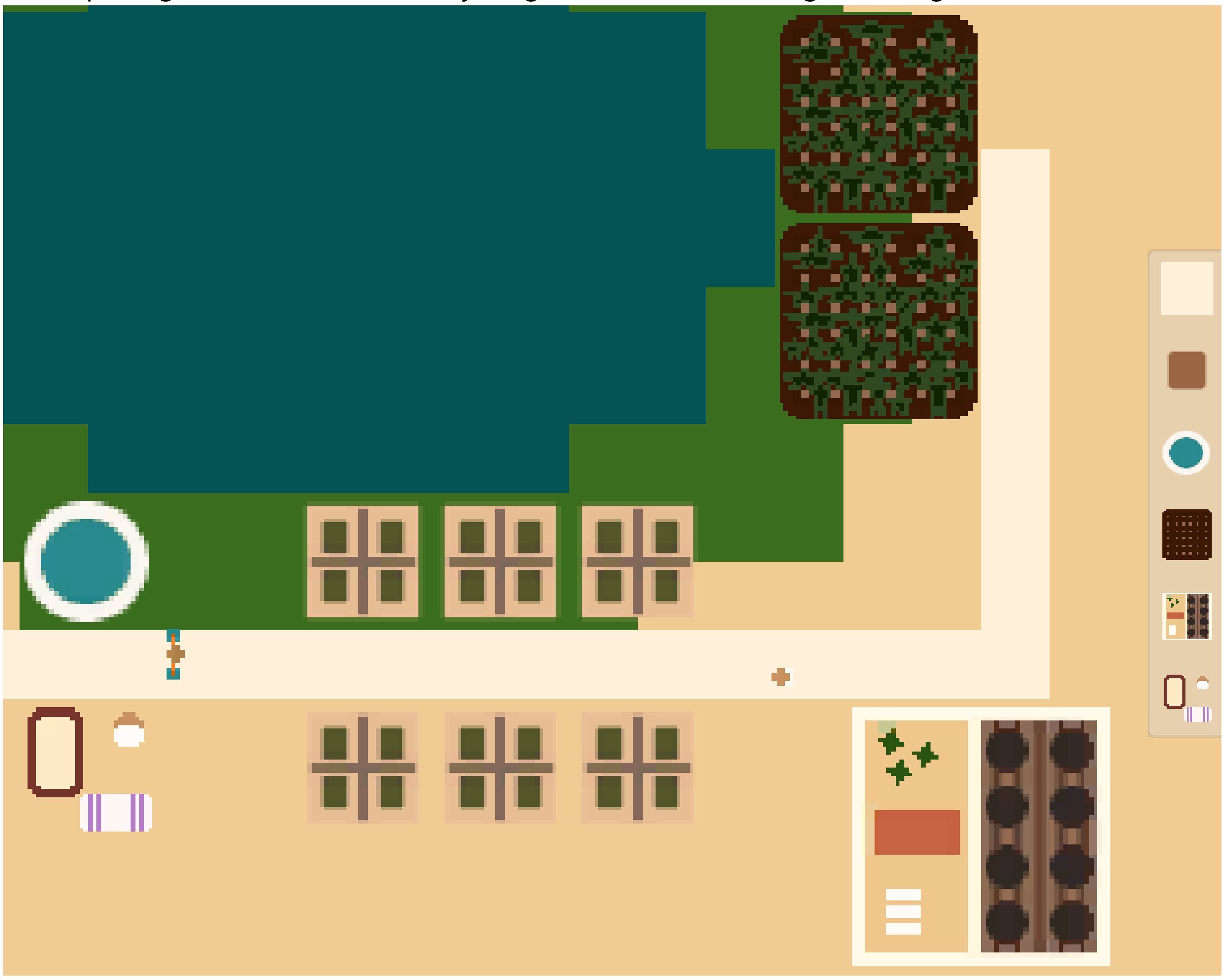
Evolution Sequence

- Create an EvolutionSequence Object in the City Folder
  Found in ContextMenu under Create/CityBuilder/Service
- Add the following Stages
  - BuildingInfo: H1Hut
  - BuildingInfo: H2Shanty Services: Water
  - BuildingInfo: H3Cottage Items: Chickpeas

Evolution Component

- Add an EvolutionComponent to all 3 Housing Buildings(H1, H2, H3)
- Assign the EvolutionSequence
- Add a ServiceRecipient for Water to all 3 with a Loss of 5
- Add an ItemsRecipient for Chickpeas to H2 and H3
  Set Storage to FreeItemCapped Mode with Capacity 10 and ConsumptionInterval to 2

Start up the game and combine everything to evolve the housing to Cottages



Saving

- Create a BuildingInfoSet Object in the Buildings Folder
  Found in ContextMenu under Create/CityBuilder/Sets/BuildingInfoSet
- On ObjectRepository assign Buildings with it.

Thats all we were missing for saving and loading. In the absence of any UI you can use the In-Editor Buttons on Logic/DefaultGameManager to save and load.

1. CCBK supports XYZ and XZY swizzles, the default XYZ is good for 2D ↩
2. blocking and ground are needed later to prevent walking and building on water ↩
3. keys have to be unique within their category, they are mostly used for saving ↩
4. the completed project uses a tree of prefab variants for buildings(skipped here for brevity) ↩ ↩[2]
5. switch the inspector to debug and select the other script, switch inspector back to normal ↩
6. sprite is a separate object in this one because is drew the walkers in the wrong direction and they have to be rotated ↩
7. i know ↩

Custom Systems

In this section i will walk you through creating a custom system in CCBK. The system itself is not especially useful but it contains implementations of all the most important parts of the framework.

A **completed, playable** version can be found in the Custom folder of the CityBuilderManual project within the CCBK asset.

Building Component

We will start off with a pretty basic building component. All it will do is keep a building effective for a certain duration and then disrupt it until it is reset.

Before actually creating the component let's add an interface for the component so we can easily add other components for the same purpose or switch out the implementation. This step can be skipped, using the component directly instead of an interface works too.

```
public interface ICustomBuildingComponent : IBuildingComponent
{
    void DoSomething();
}
```

Next add a script that inherits from BuildingComponent and implements the interface.

```
public class CustomBuildingComponent : BuildingComponent
{
    public override string Key => "CCO";

    public GameObject GoodVisual;
    public GameObject BadVisual;

    public float Duration;

    private float _time;

    private void Update()
    {
        if (_time > 0)
        {
            _time -= Time.deltaTime;
        }

        GoodVisual.SetActive(IsWorking);
        BadVisual.SetActive(!IsWorking);
    }

    public void DoSomething()
    {
        _time = Duration;
    }
}
```

As you can see i've added the reset logic and some GameObjects for visualization. The Code is mandatory for building components and is used to identify the component in save/load. We'll add the save logic next, put the following at the end of your script.

```
#region Saving
[Serializable]
public class CustomComponentData
{
    public float Time;
}

public override string SaveData()
{
    return JsonUtility.ToJson(new CustomComponentData()
    {
        Time = _time
    });
}
public override void LoadData(string json)
{
    var data = JsonUtility.FromJson<CustomComponentData>(json);

    _time = data.Time;
}
#endregion
```

As you can see saving usually contains a data class that stores all the runtime data of the component. The building component base class provides overridable save and load methods that are called by the building. Saving is optional, if the methods are not overridden or return empty not data will be saved.

The last thing we need is for the component to influence the buildings efficiency. We do this by implementing IEfficiencyFactor

```
public class CustomBuildingComponent : BuildingComponent, ICustomBuildingComponent,
IEfficiencyFactor
{
    public override string Key => "CCO";

    public bool IsWorking => _time > 0;
    public float Factor => _time > 0 ? 1 : 0;
...
```

Building Trait

A BuildingTrait is a special BuildingComponent that always creates a Reference an is stored in a special collection within the BuildingManager. This is done so they can be retrieved without having to iterate through all the buildings on the map. The interface for it looks like this.

```
public interface ICustomBuildingTrait : IBuildingTrait<ICustomBuildingTrait>
{
    float CustomValue { get; }

    void DoSomething();
}
```

The minimal implementation for the Trait is similar to a normal component.

```
public class CustomBuildingTrait : BuildingComponent, ICustomBuildingTrait
{
    public override string Key => "CTR";

    public float CustomValue => 1;

    public BuildingComponentReference<ICustomBuildingTrait> Reference { get; set; }

    public override void InitializeComponent()
    {
        base.InitializeComponent();

        Reference = registerTrait<ICustomBuildingTrait>(this);
    }
    public override void OnReplacing(IBuilding replacement)
    {
        base.OnReplacing(replacement);

        var replacementTrait = replacement.GetBuildingComponent<ICustomBuildingTrait>();

        replaceTrait(this, replacementTrait);
    }
    public override void TerminateComponent()
    {
        base.TerminateComponent();

        deregisterTrait<ICustomBuildingTrait>(this);
    }

    public void DoSomething()
    {
        Debug.Log("Hello!");
    }
}
```

The major difference to normal components is that we have to manage a reference throughout the lifecycle of the component. It has to be created when the building is build, passed on when it is replaced and removed when the building gets demolished.

The purpose of the trait is to spawn walkers so we will return to it after creating some custom walkers.

Walker

The first walker we will add is a roaming walker that resets every ICustomBuildingComponent it passes. CCBK has a base class called BuildingComponentWalker for exactly this purpose.

```
public class CustomRoamingWalker : BuildingComponentWalker<ICustomBuildingComponent>
{
    protected override void onComponentEntered(ICustomBuildingComponent buildingComponent)
    {
        base.onComponentEntered(buildingComponent);

        buildingComponent.DoSomething();
    }
}
```

Thats it, whenever the walker passes a ICustomBuildingComponent onComponentEntered will be called. The base class also provides onComponentRemaining in case you want to do something as every frame the component is in reach instead of just the first one.

The second most prevalent kind of walker in city builder are destination walkers. The one we'll add is fairly simple since it gets its path passed to it from outside. It will follow that path it is given and call a method on its target when it is finished.

```csharp
public class CustomDestinationWalker : Walker
{
    public enum CustomDestinationWalkerState
    {
        Inactive = 0,
        Walking = 1
    }

    private CustomDestinationWalkerState _state = CustomDestinationWalkerState.Inactive;
    private BuildingComponentReference<ICustomBuildingTrait> _target;

    public void StartWalker(BuildingComponentPath<ICustomBuildingTrait> customPath)
    {
        _state = CustomDestinationWalkerState.Walking;
        _target = customPath.Component;
        walk(customPath.Path);
    }

    protected override void onFinished()
    {
        if (_target.HasInstance)
            _target.Instance.DoSomething();

        base.onFinished();
    }

    #region Saving
    [Serializable]
    public class DeliveryWalkerData
    {
        public WalkerData WalkerData;
        public int State;
        public BuildingComponentReferenceData Target;
    }

    public override string SaveData()
    {
        return JsonUtility.ToJson(new DeliveryWalkerData()
        {
            WalkerData = savewalkerData(),
            State = (int)_state,
            Target = _target.GetData()
        });
    }
    public override void LoadData(string json)
    {
        var data = JsonUtility.FromJson<DeliveryWalkerData>(json);

        loadWalkerData(data.WalkerData);
```

```
            _state = (CustomDestinationWalkerState)data.State;
            _target = data.Target.GetReference<ICustomBuildingTrait>();

            switch (_state)
            {
                case CustomDestinationWalkerState.Walking:
                    continueWalk();
                    break;
            }
        }
        #endregion
    }
```

The _state variable is only added so the walker can be correctly restored when loading. The _target is stored so the walker can call DoSomething on it when it is reached.
The WalkerData field in the save data contains all the basic data of the walker like current position and path which enables you to just call continueWalk in Load.

WalkerSpawner

To actually spawn walkers from buildings CCBK uses something called in WalkerSpawner. These come in Cyclic, Pooled and Manual varieties and are serializable field that are added to building components. In Unity versions prior to 2020.1 generic fields do not get serialized which which is why you will see concrete spawner implementations at the end of all the walker classes in CCBK. If you are using such a version add the spawners to the walkers.

```
/// <summary>
/// concrete implementation for serialization, not needed starting unity 2020.1
/// </summary>
[Serializable]
public class ManualCustomRoamingWalkerSpawner : ManualWalkerSpawner<CustomRoamingWalker> { }
/// <summary>
/// concrete implementation for serialization, not needed starting unity 2020.1
/// </summary>
[Serializable]
public class CyclicCustomRoamingWalkerSpawner : CyclicWalkerSpawner<CustomRoamingWalker> { }
/// <summary>
/// concrete implementation for serialization, not needed starting unity 2020.1
/// </summary>
[Serializable]
public class PooledCustomRoamingWalkerSpawner : PooledWalkerSpawner<CustomRoamingWalker> { }
```

Now add the spawner to the building trait we created earlier.

```
public CyclicCustomRoamingWalkerSpawner CustomRoamingWalkers;
public CyclicCustomDestinationWalkerSpawner CustomDestinationWalkers;

private void Awake()
{
    CustomRoamingWalkers.Initialize(Building);
    CustomDestinationWalkers.Initialize(Building);
}
private void Update()
{
    if (Building.IsWorking)
    {
        CustomRoamingWalkers.Update();
        CustomDestinationWalkers.Update();
    }
}
```

Initialization and updating of the spawners is handled by the owning building component. You might have noticed that StartWalker on the destination walker is not called here. While that would be the easier thing to do we'll move that responsibility out into a manager for the sake of this example.

Manager

Managers are central scripts meant to orchestrate different systematic behaviors. Once again we'll start by creating an interface.

```
public interface ICustomManager
{
    float GetTotalValue();
    BuildingComponentPath<ICustomBuildingTrait> GetPath(BuildingReference home, PathType pathType);

    void Add(BuildingComponentReference<ICustomBuildingTrait> trait);
    void Remove(BuildingComponentReference<ICustomBuildingTrait> trait);
}
```

The manager will be responsible for calculating a total from the CustomValue field of our traits. It will also provide the paths for our destination walkers. Lastly it will be notified when oue of our traits gets built or demolished. Let' look at the implementation.

```
public class CustomManager : MonoBehaviour, ICustomManager
{
    public float Multiplier;

    private void Awake()
    {
        Dependencies.Register<ICustomManager>(this);
    }

    public float GetTotalValue()
    {
        return Dependencies.Get<IBuildingManager>().GetBuildingTraits<ICustomBuildingTrait>().Sum(t
=> t.CustomValue) * Multiplier;
    }
    public BuildingComponentPath<ICustomBuildingTrait> GetPath(BuildingReference home, PathType
pathType)
    {
        foreach (var other in Dependencies.Get<IBuildingManager>
().GetBuildingTraitReferences<ICustomBuildingTrait>())
        {
            if (other.Instance.Building == home.Instance)
                continue;

            var path = PathHelper.FindPath(home.Instance, other.Instance.Building, pathType);
            if (path == null)
                continue;

            return new BuildingComponentPath<ICustomBuildingTrait>(other, path);
        }

        return null;
    }

    public void Add(BuildingComponentReference<ICustomBuildingTrait> trait)
    {
        Debug.Log("Custom Trait Added!");
    }
    public void Remove(BuildingComponentReference<ICustomBuildingTrait> trait)
    {
        Debug.Log("Custom Trait Removed");
    }
}
```

To calculate the total value the manager pulls all the traits from the BuildingManager and applies a Multiplier. As mentioned before BuildingManager keeps track of all traits for quick retrieval. For the path we just loop all traits and return the first one that PathHelper is able to calculate. The Add and Remove Methods don't have and function in this demonstration so we'll just log to check if they work.

Score

To be able to use the total value calculated by the manager in visuals and win conditions we will now create a score. Scores are pretty straightforward ScriptableObjects that are implemented like this.

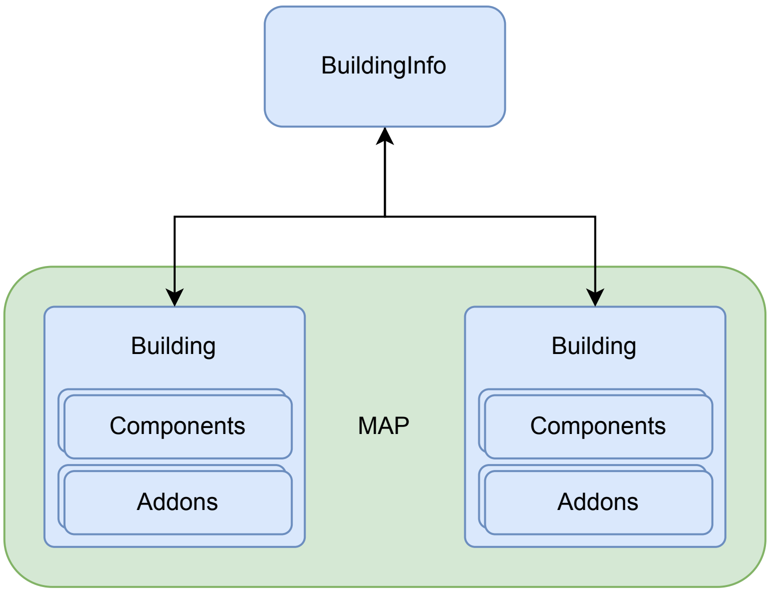
```
[CreateAssetMenu(menuName = "CityBuilder/Scores/" + nameof(CustomScore))]
public class CustomScore : Score
{
    public float Multiplier;

    public override int Calculate()
    {
        return Mathf.RoundToInt(Dependencies.Get<ICustomManager>().GetTotalValue() * Multiplier);
    }
}
```

To see the score add some kind of text to the scene in combination with a ScoreVisualizer. For the score to be calculated you need a ScoreCalculator in the scene. The ScoreCalculator pulls the Scores it has to calculated from a set of scores it retrieves from the Dependencies. To register your custom score for that add an ObjectRepository to the scene and give it a ScoreSet that contains the instance of score used in the visualizer.

Buildings

The bread and butter of every good city are, of course, buildings.
In CCBK Buildings do not define any actual behaviour.
Instead they act as containers for components and addons that can be placed on the map.

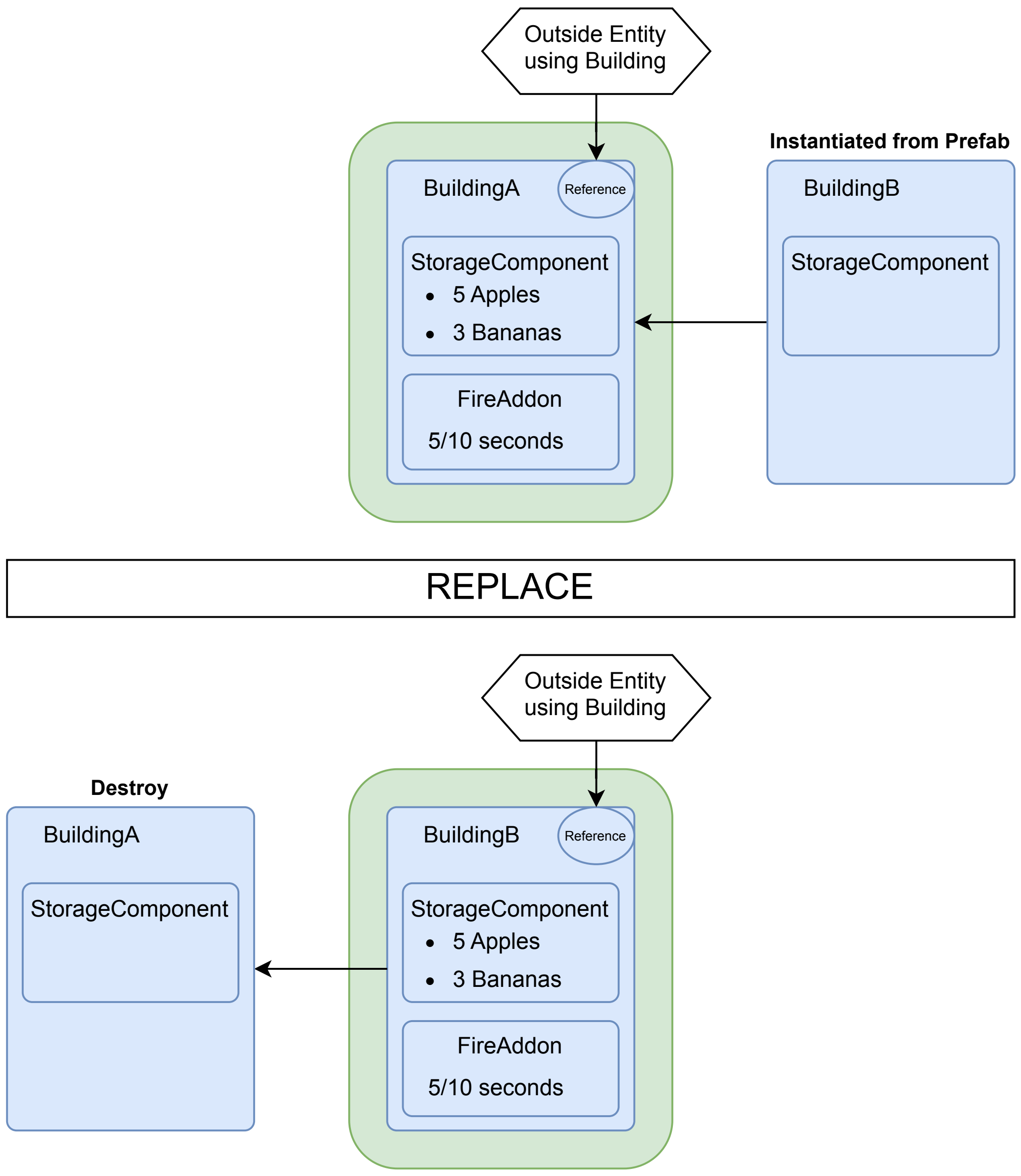


Global properties of buildings like name, size, prefab and access type are stored in a ScriptableObject called BuildingInfo.

> *give buildings that spawn roamers a preferred access point to avoid annoying scenarios where roamers spawn wrong because a road was added*

A system deeply embedded in buildings is the ability to replace them.
This can be used to change a buildings appearance and behaviour by replacing it with another.

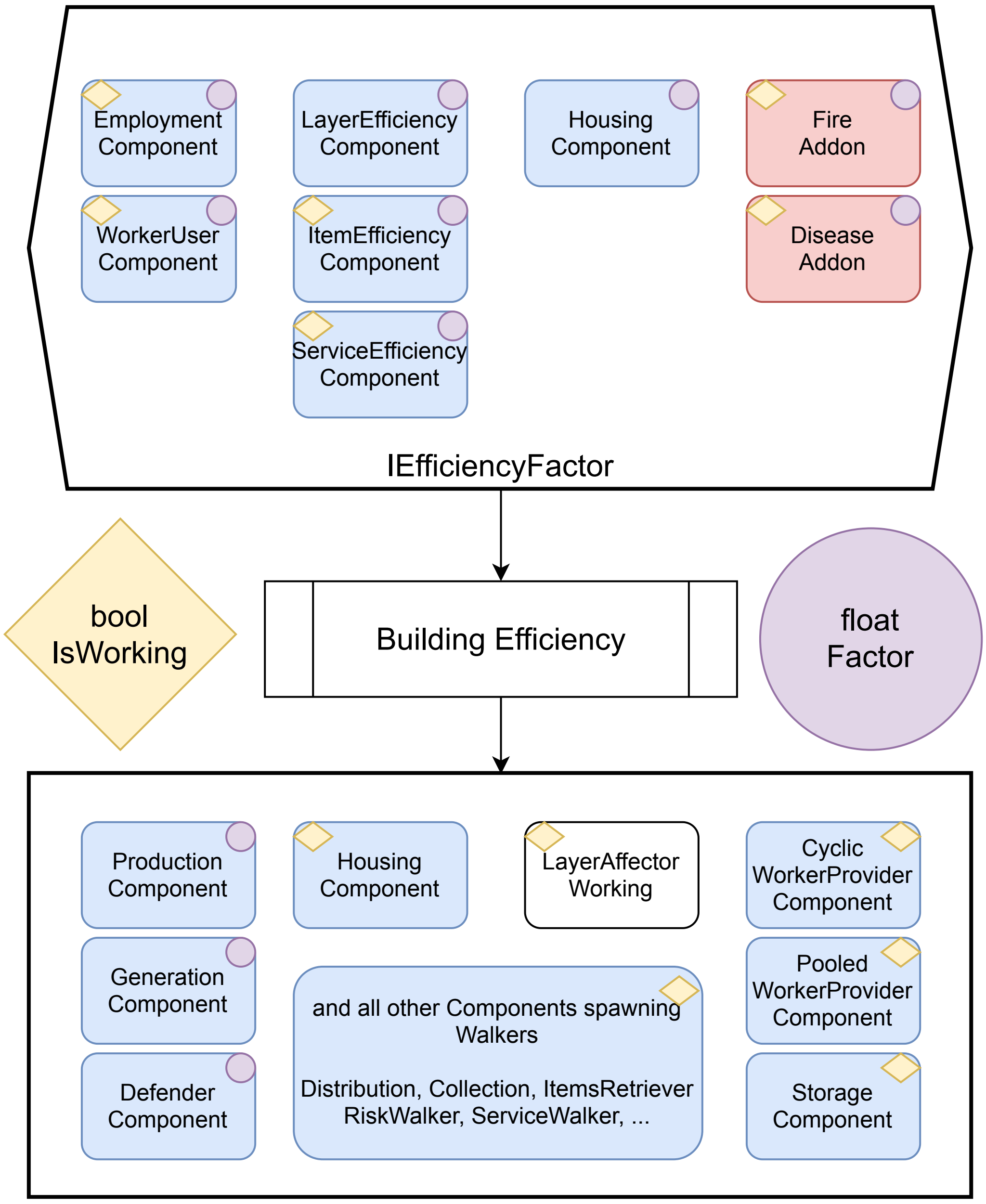
> *useful for evolving housing, upgradable towers, ...*



Anything interacting with a Building or one of its parts should not directly carry a reference to it. Instead a BuildingReference can be used which updates its reference when the building is replaced.

The different building parts can either interact directly of through the buildings efficiency.

> *eg farms growing wheat quicker based on land fertility or storages not accepting wares without enough workers*



The efficiency is expressed through two values. IsWorking[bool] is meant to indicate whether the building is generally working or somehow disrupted. The Efficiency[float] expresses how well the building is working from 0-1.

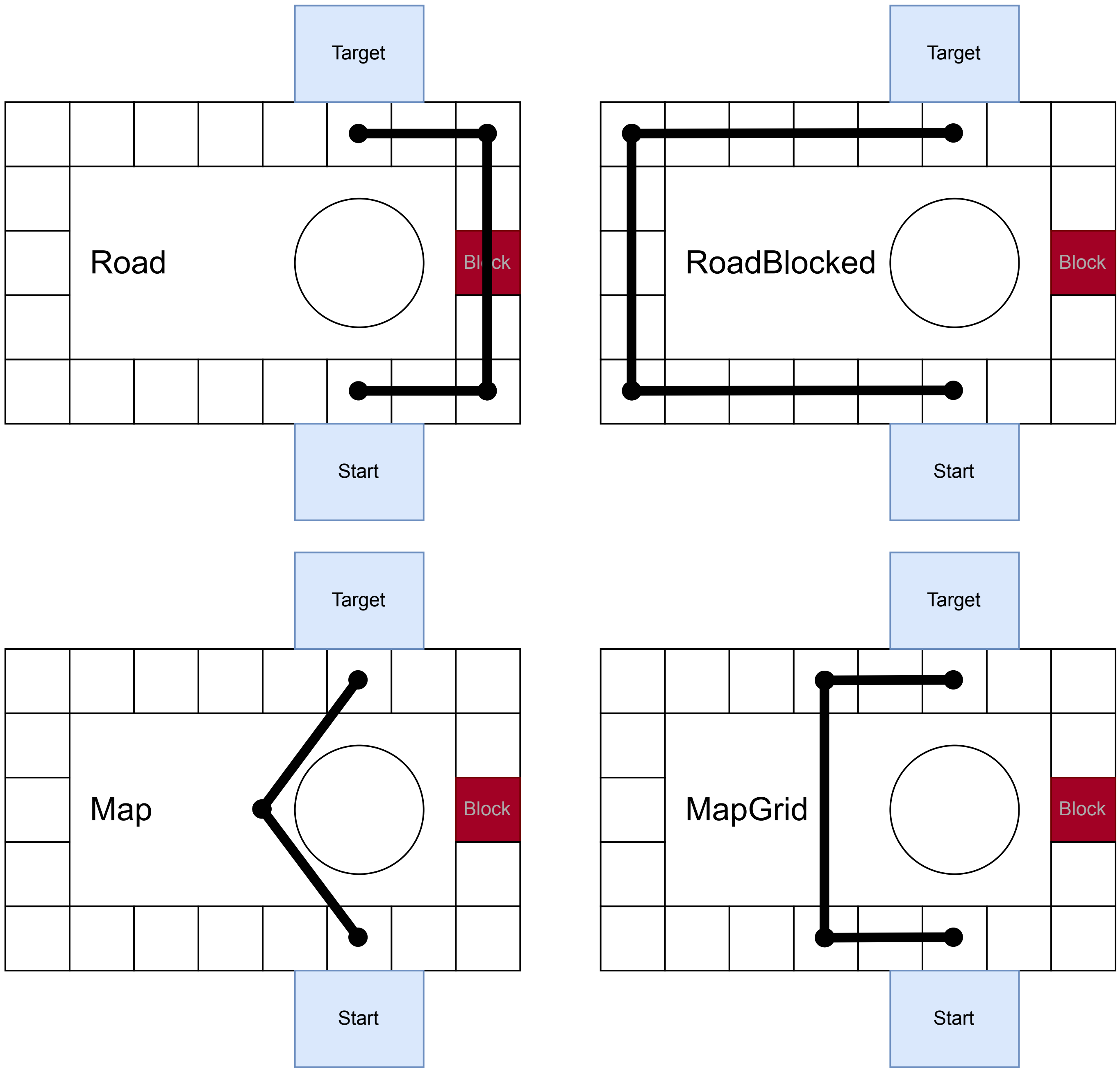
Walkers

The primary way to move things about in CCBK.

> *storage workers moving items, hunters collecting meat, architects inspecting buildings*

Path Types

Among other base functions Walkers have a setting for PathType which specifies the kind of pathfinding it will use to get to its destination.



Road

Simple A* Pathfinding inside the road network on the map. Usually used for all kinds of destination walkers.

> *carriers delivering goods, workers walking to a monument site, ...*

RoadBlocked

Same as Road but points with RoadBlockers are removed. Usually used for all kinds of roaming walkers.

> *water carrier distributing out water,*

Map

This Pathfinding uses Unity NavMesh. All demo scenes have a gameobject called Navigation that is used purely to generate the NavMesh. To change the NavMesh follow these steps.

- activate Navigation gameobject
- adjust the objects inside Navigation
  - make sure Navigation Area is correctly set to Walkable/Not Walkable
- bake the NavMesh
- deactivate Navigation gameobject

  > *immigrants walking to their new homes, hunters walking to pray, ...*

MapGrid

Simple A* pathfinding of all points on the map that are not occupied. Only feasible for small maps, on larger maps use Map pathfinding instead.

Walker Modes

The walker base class includes helper methods for various modes of walking. These save their states automatically and can easily be continued on loading.

Walk

Directly pass a WalkingPath for the Walker to follow it immediately.

> *used when the path is checked before the walker is even spawned like storage walkers*

TryWalk

Tells the walker to try to find a way to a target or give up after a certain time. The walker can be given a target building or position to check pathfinding by itself or a function that returns one.

> *deliver walkers spawned by production are spawned and have to figure out a path afterwards*

Roaming

The walker randomly walks around while trying to avoid points already visited. It does for a set number of steps specified in Range, how many positions it remembers is defined in Memory.

> *well workers handing out water, bazaar workers selling food*

Wander

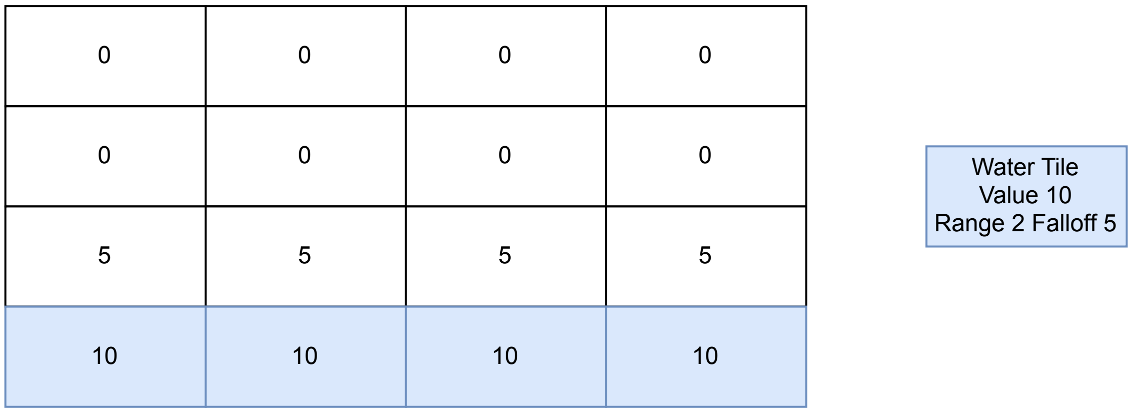
Moves to a random adjacent point

> *homeless walkers*

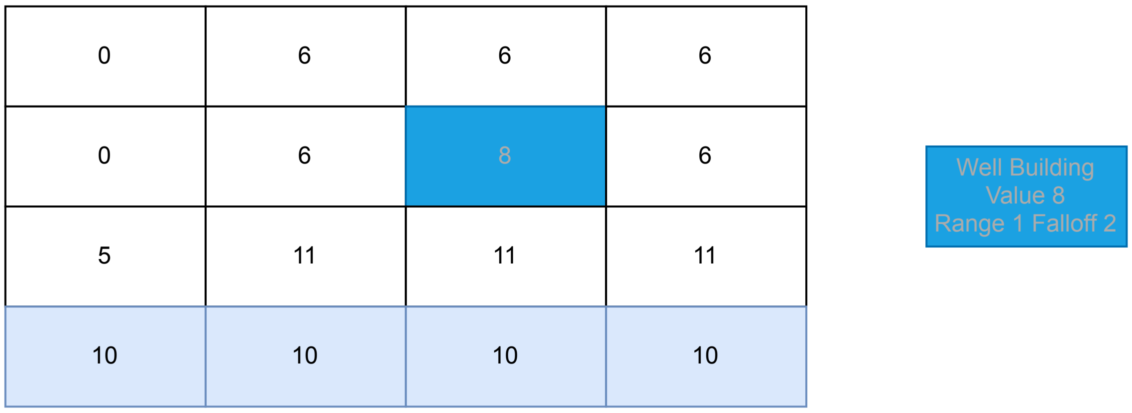
Layers

Layers define numeric values for every point on the map.

> *resources like water and ore, desirability of buildings or even the spreading heat from fires*



0	0	0	0
0	0	0	0
5	5	5	5
10	10	10	10



0	6	6	6
0	6	8	6
5	11	11	11
10	10	10	10

At the start of the game a layers base values are calculated using predefined tiles. After that LayerAffecters can register with the layer system to add their own values.

Here are some of the ways Layers interact with other systems

- requirement for building or evolution

> *wells need a minimum amount of water, housing needs desirability to evolve*

- (working-)affecter on building

> *eg gardens affecting desirability, working stages affect entertainment*

- layer efficiency component affecting building efficiency

> *fertility on farm buildings,*

- multiplier for risk increase and service decrease

> *fires happen faster in hotter positions and more water is consumed*

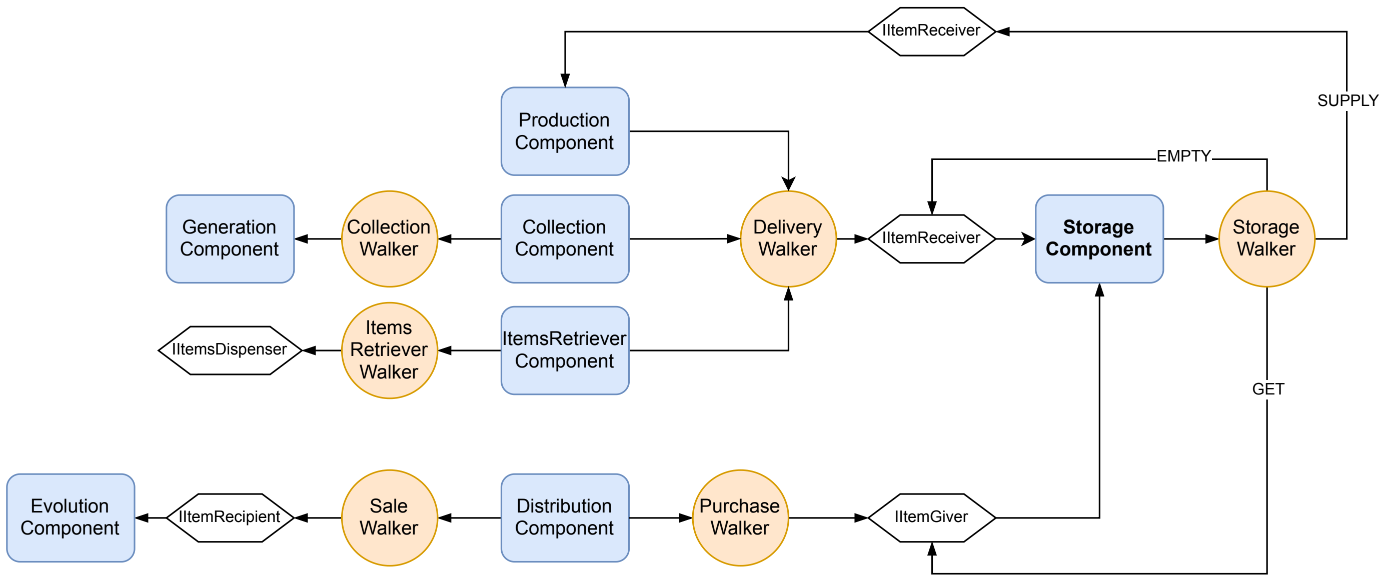
Resources

The different resource systems deal with creating, moving and consuming items. Items are ScriptableObjects that specify how the item is visualized in storage and UI among other settings.

> *wood, meat, clay, iron, ...*

Items are stored in the ItemStorage class which can limit how many items they fit by item quantity, item units or stacks. ItemStorages are either part of building components or a global manager. ItemsStorages on Buildings can be set to global mode so their items go to global storage instead.

> *gold is stored in global storage, food is stored in buildings*



The two essential interfaces regarding resources are IItemReceiver and IItemGiver.

- Item receivers are building component that accept items from others.
  Other components deliver items to receivers.
- Item givers are building components that provide certain items for others.
  Other components acquire items from givers.

Storage

Storage Components act as both receivers and givers. They need a StorageOrder for every Item they store. Storage components performs actions in the following order.

- get items with StorageOrderMode GET from other givers
- supply receivers with a higher priority with items(production)
- empty items with StorageOrderMode EMPTY to other receivers

> *silos, storage yards, ...*

Production

As long as it is supplied with items specified in ItemsConsumers a production component will periodically use up those and produce the items specified in ItemsProducers. It does not actively get the consumer items but instead registers as an IItemsReceiver. It does however spawn delivery walkers to move produced items to other receivers.

> *barley farm, brewery, clay pit, potter, ...*

Distribution

These components send out roaming Sale Walkers that take note of the items IItemRecipients it passes need. They then look for IItemGivers that give out those items and send Purchase Walkers to get them. The Sale Walkers are filled up with items whenever they leave and pass out the items to the IItemRecipient they pass.

> *markets*

Retrieval

Retrieval Components periodically send out walkers that get items from the closest dispenser. They then spawn delivery walkers that try to move them to an items receiver.

> *hunting lodge, wood cutter, ...*

Collection

Collection Components spawn walkers that collect items from generation components the pass while roaming. These also use delivery walkers to move the collected items on.

> *tax collector*

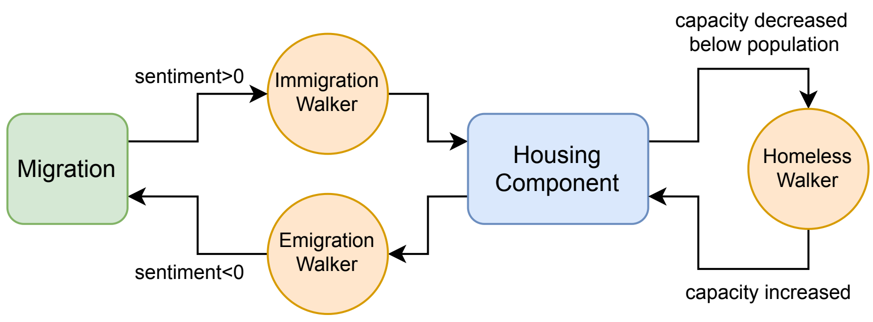
People

CCBK supports multiple classes of citizens, create a Population ScriptableObject for each one. Different populations may be able to fill different job openings or pay different tax rates.

> *plebs, scholars, aristocrats, ...*

Housing

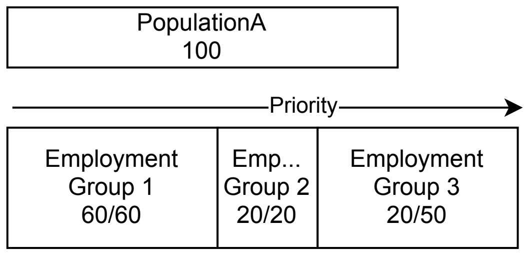
Housing components provide space for a specified number of a defined population. Migrations control how fast new citizens arrive or leave based on their sentiment value.



One migration script per population is needed which allows different rates of immi-/emigration per population as well as different entry points on the map.

Employment

Directly sustained by the population is the employment system. Individual Buildings either always have full access to the employment pool or they send out an EmploymentWalker that checks if any buildings housing the needed population are nearby.
Employment is mostly used to drive building efficiency.



It is possible to define different groups of employment to specify which buildings get employees first.

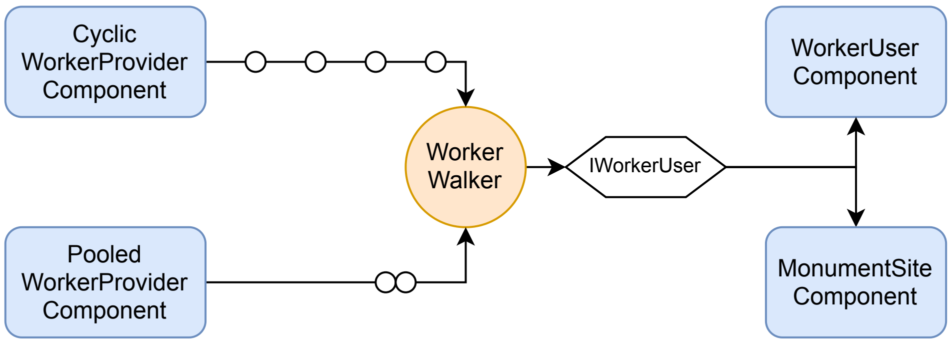
> *essentials like water should not cease operation because ore mining is using up its employees*

CCBK comes with a dialog that allows the user to change employment priorities at runtime.

Workers

The worker system is not directly liked to populations but will most likely be effected by building employment/efficiency.
Some buildings need trained workers to function effectively which creates a slightly different kind of production chain.

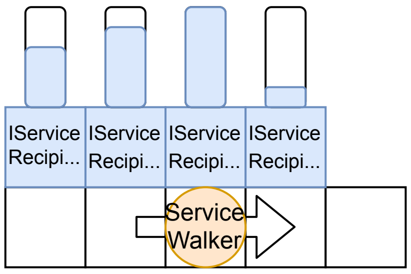
> *farms, monuments, ...*



Services

Services are values on Buildings that decrease over time and influence building efficiency and evolution. ServiceWalkers roam around filling the Service values of ServiceRecipients they pass.

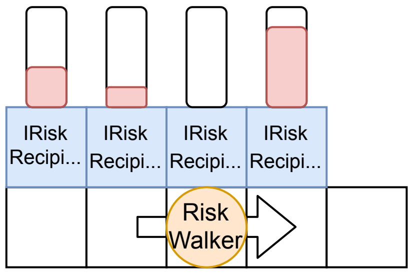
> *water access, religion, education, ...*



Risks

Risks are values on Buildings that increase over time and cause some event when full.
RiskWalkers roam around reducing the Risk values of RiskRecipients they pass.

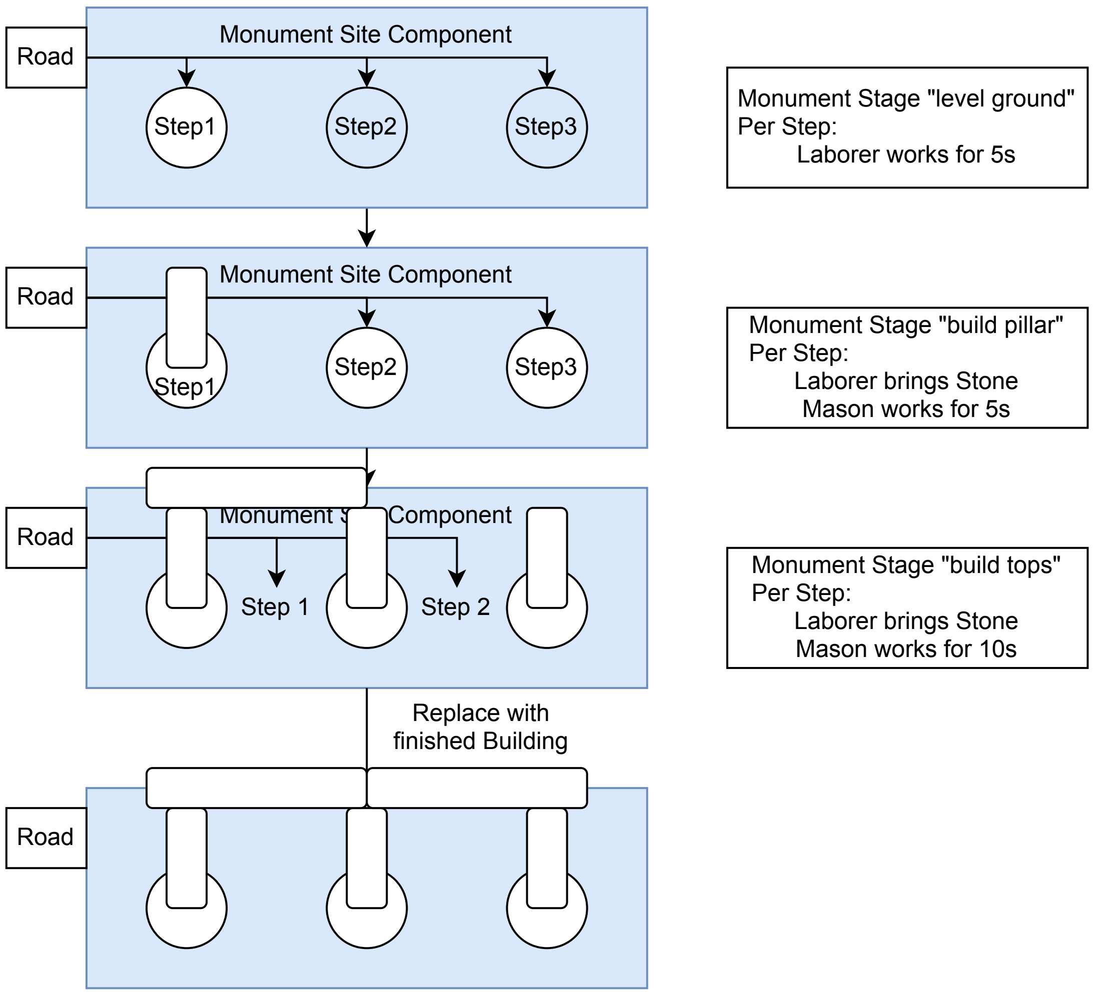
> *fire, collapse, disease*



CCBK comes with Risks that add Addons to Buildings or replace the Building with other structures or Buildings.
To add Risks that execute other actions simply inherit from Risk and implement the Execute and Resolve methods.

Monuments

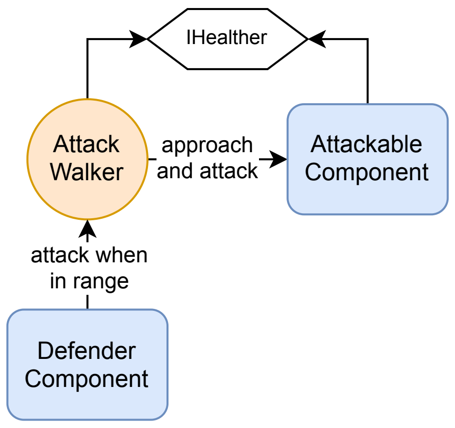
Monuments are buildings that take time, resources and workers depending on their current stage of construction. They are supplied by the Worker System and, unlike other worker users, provide the workers with a path to their respective work places inside the site when they arrive.



The created building does not necessarily have to be a monument, monument sites can be used for any building that needs time and resources to be built instead of being instantly plopped down.

Attacks

The attack system coming with CCBK provides the features needed for basic tower defense functionality. While it is prominently featured in the Defense Demo.



The IHealther interface is used to display the health bars.

> *i'd love to see a city builder with tower defense mechanics*

Structures

A structure is basically anything that is placed on the map.
The default structure manager stores basic structures in a dictionary based on the position, so checking and retrieving structures when the position is known is fast. This has the downside that only one regular structure can be stored per position. Since only Buildings are stored this way in Core CCBK this is not a problem.
StructureCollections and Tiles are always stored separately and Roads register themselves as underlying structures which also stored them in a separate list.

Structures are defined by the points they occupy on the map as well as three bit flags:

- IsDestructible
  Specifies whether the structure can be removed by the player
- IsDecorator
  Decorator structures are automatically removed when something is built on top of them
- IsWalkable
  Whether the MapGrid Pathing Type includes the points of the structure

Roads

The DefaultRoadManager is a special structure that includes all the points that are filled out in the TileMap on the GameObject and does pathfinding between them. It is possible to define multiple Roads with different Stages that work similar to Building Evolutions.

> *simple paths turn into prettier streets when the desirability of the area improves*

StructureCollections

A collection of GameObjects that interact with the map somehow.
Only one kind of GameObject is allowed per collection and the prefab has to be specified. This is done so the Structures can be restored after the game is loaded.

> *3D - trees, rocks, rubble, walls...*

StructureTiles

Similar to StructureCollections but using Tiles instead of GameObjects.

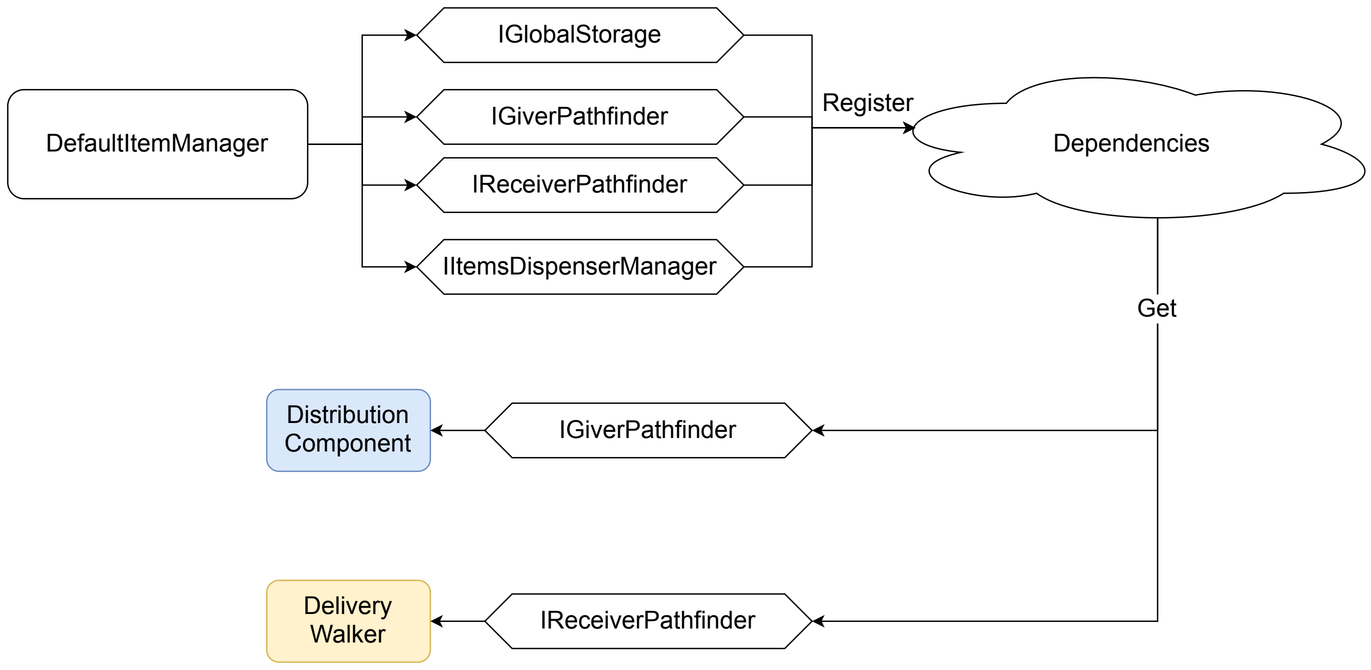
> *2D - trees, rocks, rubble, walls...*

Buildings

see Buildings

Dependencies

Dependencies in CCBK are used similarly to IOC Containers in other areas of development. The implementation however is stripped down to a minimum to accommodate the performance needs of games.
Basically Dependencies allow the registration and retrieval of interfaces. This allows the implementation to be changed and moved without having to rewire all entities that depend on it.



Most systems in CCBK have some central manager component that all the components need access to. Theses managers are not wired up in the editor or implemented as singletons.
Instead the system includes an interface for that manager. The Behaviour that actually implements the interface calls Register on Dependencies when it awakens in the scene. Any components that need the manager call Get on Dependencies.

When possible keep changes to CCBK to a minimum. Instead remove the default implementation from the scene and add your own.

Let's say you wanted to change how the production components are prioritized for delivered raw materials.

- copy DefaultItemManager to the project folder and it to something like MyProjectItemManager
- change the code to fit your project
- replace the item manager in the scene
    - pro tip: switch to debug mode in the inspector and replace the script to keep the field values

Many of the default managers register multiple interfaces for convenience, these can be easily split up thanks to the Dependency system. To replace just one of those interfaces with a custom implementation comment out the line in Awake where that interface is registered. That change will be lost if the Asset is upgraded but should be easy enough to restore.

Dependencies are automatically cleared between scenes.

All interfaces registered with Dependencies are effectively singletons. A possible extension of the system would be passing a tag when registering and getting to partition different areas of dependencies(eg different road managers for underground, ground and sky). CCBK does not include such an extension to keep performance optimal for those that do not need it.

Scores

Scores represent all kinds of numerically quantifiable values for a city. They are used for displaying statistics to the player and checking win conditions.
The different Score Scripts each implement a way of calculating a score value. They are available as ScriptableObjects in the CityBuilder/Scores Context Menu.

You can easily create your own scores by inheriting from Score and implementing the Calculate Method.

Some of the Scores included in CCBK are:

Average Building Score

Calculates the average value from different values assigned to Buildings

> *Tent:0 Hut:50 Palace:100 > 1xTent 2xHut 2xPalace = 60 Housing Quality*

Average Service/Risk Score

Averages out the risk/service value of a defined building category

> *how well are housings supplied with water, whats the general risk of collapse across my city, ...*

Building, Item, Population Score

Just counts the number of a building that exist.

> *how many pyramids have been built, how many diamonds stored, how many plebs live in town, ...*

Coverage Score

Calculates the number of people buildings would cover against the current population count

> *1 temple covers 1x100 , 2 shrines cover 2x50 > population=400 => 50% Coverage*

Multiplied, Summed, Threshold Score

can be used to transform other scores

culture = entertainment + monuments, different threshold of employment, ...

Misc

Saving

CCBK includes a complete save and load system that serializes save data using unity’s json serializer. The save data containers and save/load logic can generally be found in ‘#region Saving’ at the end of the respective script.
Save-/LoadData are perpetuated through the different systems starting at the central manager scripts. This means core components of CCBK like building components and walkers already have overridable Save/Load Methods that will be called without any extra work.
The easiest way to hook data that is outside of CCBK into the save system is to inherit from ExtraDataBehaviour. The default game manager finds all those in its awake Method.

Mission Parameters

Mission, Difficulty and MissionParameters are a proposal for managing different scenarios in your game.
Mission and Difficulty are Scriptable Objects that can be defined in the Editor.
MissionParameters includes all parameters needed to start a game and are passed to the IMissionManager when the scene is loaded.