The Models

Organization of the Prefabs

Props

The models are organized into the following folders:

Furniture: Prefabs in this folder are typically static and have only a collider with no rigid body, and they are not usually expected to move – though some have non-static moving parts as children of the main model (for example, lids or drawers).

Gear: Tools and weapons are here. These come with kinematic rigid bodies. Weapons come in various material types, with prefixed denoting the originally intended interpretation:

- **P:** primitive (wood, stone, bone)
- **B** or **Br:** bronze
- **I:** iron
- **St:** steel
- **Bs:** blue steel (more advanced fantasy steel)
- **E:** Elven make (material called "mythrite" as an ingot)
- **A:** adamantine

By material here, I mean material in the every day sense, "in world," not Unity material for rendering them. Of course, you are free to re-interpret what these mean, as they are mostly color variations.

Items: These are typically smaller things that are likely to move around, and so come kinematic rigid bodies attached. Of course, they can be made non-kinematic, though unless you are using expensive third party physics solutions (like Havok) doing so may not get the best results as placing some items onto other (e.g., food on plates) will involve either one item floating or else exploding off of or falling through the other. This is because of the limitations of Unity's default physics with non-static mesh colliders that are built in for optimizations reasons; this is not good for placing something on a concave object such as a plate.

The items category is where you can find food, bottles / bottled liquids, books, dishes / cookware, music instruments, and so on.

Nature: Things like rocks can be found here.

Traps: Self-explanatory; traps and parts of traps are here.

Treasure: This is for items intended primarily as treasure for the player to find as loot, though some of it works for other purposes such as general decoration or crafting. Paintings are included here. Of course, some items in the items category also work as treasure; gold and silver cups and candelabras make great loot, but to keep like items together are found in the same category as less valuable versions

of the same thing. Ingots are also here, as many are precious metals, with less valuable ingot also here to keep items of the same type in the same place.

Architecture

The architectural components are designed to make it easy to assemble a dungeons structure. They are based on a base scale of 3m x 3m, or 10 x 10 feet (10 feet being about 1 ¾ inches or 5 cm less than 3m, close enough to treat as the same in a game) – partially in imitation of table top games and partly because the results looked ideal for a first person game. They are designed to snap together easily and seamlessly.

These are the categories for architectural components:

Arches: Mostly for doors, but a few others are also found here. Post and lintel door frames are also here.

Ceiling Vaults: Models to make vaulted and domed ceilings. These are designed as ceilings, not roofs, and may not look good from the outside (more for dungeon interiors).

Doorways: This includes both complete doorways and also walls with hole as components of those; the plain holes were not designed to be used on their own and may not look good that way.

Doors: This category included both raw (static) doors and doors that can be opened with scripts, as well and complete composites of openable doors in doorways. Castle gates and portcullises are also here.

Fences: This includes modular fences, gates, parts of fences, short walls, and cage walls (including cage doors). The complete contents of *Simple Modular Fences* cam be found here.

Flats: All the floors and ceilings are here. Most of these prefabs are designed to be used as both a ceiling and a floor for the level above. Included here are some with flats with holes.

Pillars: Pillars / columns are here. This includes, caps / capitals for the pillars, main column pieces, composites forming complete pillars, complete pillars modeled as a single mesh, as well as some large wooden posts that could act as pillars. Most of *Simple Modular Pillars* is here, though some of it is also under Trim (end pillars), and some bonus pillars are not included.

Platforms: This includes two kinds of platforms:

- Blocks: Simple blocks to put under other prefabs
- Daises: Raised platforms with steps for your thrones and altars

Rooves: If you need a sloped roof for a house or cottage (or castle), it will be here. For flat rooves, use flats.

Stairs: Includes stairs (up 3m or 10 feet), stair railings, and shorter, less steep steps (up 1.5 m or 5 feet).

Trapdoors: Trapdoors / hatches, both static / decorative and openable.

Trim: This category include end pillars to decorate your walls (or hide hairline cracks), molding to decorate the tops and bottoms of walls, and some loose bricks to give a dungeon wall that look of being in disrepair.

Walls: This includes both interior walls, and outer walls for a castle or fortress. The interior walls, come in variation of thin versus thick, full (3 m) versus half-length (1.5 m), and one material versus two, as well as having diagonal variations. The half-length was added mostly to be able to center doors in walls that would otherwise use an even number of walls segments. The two material walls is to allow adjacent rooms to have different wall materials. Included here are also some broken walls for Included with the castle walls are some related features, such as parapets, battlements, walls with holes for gates, and fake / shell towers. (Castle gates and portcullises are food under doors, however.)

Windows: Includes window panes, window seals, walls with holes for windows, and composites consisting of walls with complete windows in them. Extra pane are included that can be added or swapped out – just add a window pane as a child of the window seen it will be in place.

Some notes on fences

The models are basic modular place, to place in a scene. If you know how to build in the editor, you know how to use these. They should work with any third party tools that can place prefabs, of course.

A few naming conventions:

- **Raw** gates and doors are non-moving, and intended for use as decorations or non-openable barriers. These come flagged as static.

- **Openable** gates and doors are just what they say; they are built as a nested hierarchy of game objects and have a script attached that will open them when activated.

- The suffix **SA** for stand-alone is for fence section with a post on both sides, good for placing as on there own, such as to block a passage in a dungeon, or to close a loop of "trad" section.

- Sections suffixed with **Trad** are for building fences in the traditional way, by inserting the the end with no post into the post at the beginning of the next. Thus, they only have a post at one end and the other end is slightly longer to provide for insertion.

- Section labeled with **W** (for *"whole"*) have a half-post (split exactly in half) at each end, to be "snapped" next to each other like walls. These were originally intended for use with a specific tool asset that sadly has become broken abandon-ware, but they could still be used as normal assets if you prefer to build in this way.

Materials

Most prefabs in this pack use a single material, with some (notable gems) using special variations of that material. This material is based on a color pallette designed for its aesthetic appeal and stylistic consideration, with some other more brighter, more saturated colors for special purposes where a brighter or otherwise different color is needed (hot coals, gem stones, etc.), and uses three textures too allow for metallic and emissive variations of those base colors.

A few items use special textures to allow for better rendition of images, while a few tiling textures and materials are included for some special cases where colors mapped to a pallette would not be effective.

Particles and Materials

A few particle systems are included for flames, explosions, and even fountains. These should not be hard to use, or even modify if you like.

The Scripts

There a variety of scripts included, in two main packages. The first are the KFUnityUtils, a family of utilities and pre-made systems; this is included mainly for the door opening scripts which have been repurposed for opening chests and drawer, though it has other parts and has been included in full as a bonus. The second are the Simulation Cameras, which are included mostly for use with the demos to create fly cams. All these scripts (and only the scripts) are available both under the standard Unity license as part of this pack and under the MIT license (choose which one you prefer).

KFUnityUtils

https://github.com/BlackJar72/KFUnityUtils

Door Opening Scripts

Included are a handful of fairly small, simple scripts for opening doors.

- ***IDoorOpener.cs*** is the base interface defining the core methods for opening door; only useful if you are a programmer want to extend the system with more way to move doors.

- ***SimpleOpener.cs*** is am abstract base class that applies the IDoorOpener interface to a MonoBehaviour, needed because variables assigned to interfaces don't show in the inspector; also really on important for programmers want to extend the system.

- ***HingeOpener.cs*** is the script for opening doors, gates, and other things, by swinging around pivot (the hinge).

- ***MovingOpener.cs*** is a script for opening a door by moving / translating a game object between an open and close position, such as to raise a portcullis.

- ***MultiOpener.cs*** is a script for opening multiple other SimpleOpeners at once. This is intend for things like double doors.

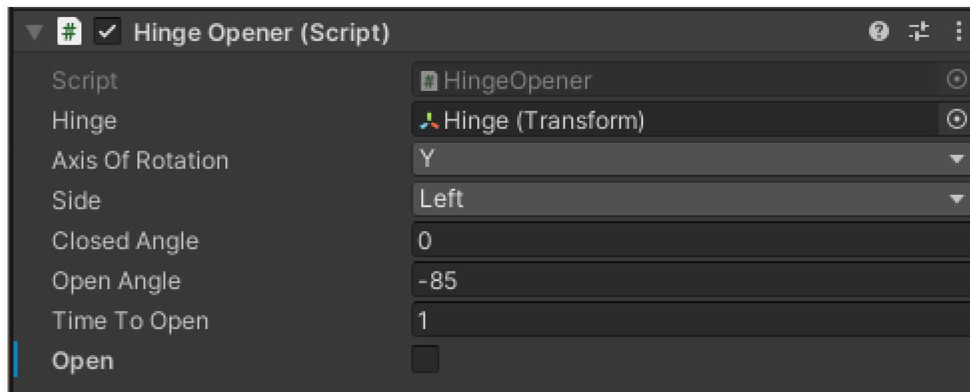
- ***StagedOpener.cs*** is a script that allows for moving a door (or other object) through several distinct stages. This was originally intend for things like secret doors that may first move back before then sliding downward or off to the side.
-

All scripts are triggered by calling their ***Activate()*** method, though ***Open()*** and ***Close()*** methods can also be called if you only want the door to move one way.

Using the Scripts

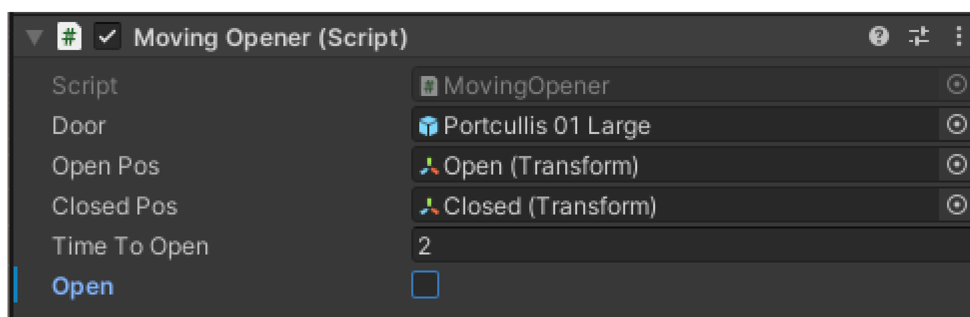
To use the scripts you will need a hierarchy of game objects in the prefab being opened.

To create a door that swings on hinge or otherwise rotates open, you will need to a hinge transform (usually an empty), which acts as the pivot around which the door will swing. This should usually be on or just past the edge of the door and inline with any visible hinges in the model. The door must be a child of the hinge, and the script may be placed on the hinge or on a containing game object or empty.



The values in the script should be fairly self-explanatory: ***Hinge*** is the hinge transform. ***Axis Of Rotation*** is the axis it will rotate through, usually Y for a door or gate, but might be X for a chest, while a trapdoor might have X or Z depending on its orientation. ***Side*** is the side the door would be on if is (or were) a double door, looking from the front; in effect, it is the side hinge is on relative to the door. ***Close Angle*** is simply the angle the door should be at when closed, while ***Open Angle*** is the angle when fully open. ***Time To Open*** is simple the amount of time it will take to move from open to close position or vice-versa. Finally, ***Open*** is simply if the door is currently open, and can be used to configure that starting position of the door from the inspector.

To create doors or gates that move to open, such as a portcullis that can be raised or lowered at the entrance to a castle, use the Moving Opener script.



The moving game object, the door or gate, needs to be placed in a container with two transforms (usually as empties) positioned where where the the game objects should be when open and when closed. The container may be an empty or any object they are part of or attached to. The variable you need to set in the inspector are those, plus ***Time To Open*** and ***Open***, which work exactly the same way as with the hinged opener.

The **MultiOpener** script simply allows you to create list of other simple opens (including hinge openers, moving openers, staged openers, or even other multi-openers) that it will all be activated when it is; it simply forwards the calls to Activate(), Open, and Closed() to any of the door openers in its list.

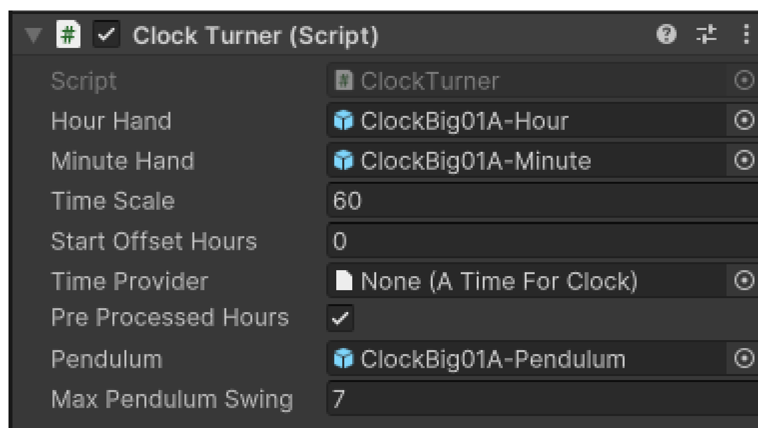
Calling Activate, Open, or Close from another script or with unity events wired in the inspector should be all it takes to make it work. Of course, it could be used for more than just doors and gates – lids on chests, or ever traps that swing a blade out or jab with something could also be animated with these scripts.

Staged Openers simply provides a series of transforms to representing the positions it will move through, along with a list of times to reach these positions.

Other Classes (directly under util)

ClockTurner.cs

This class is used to animate clocks and have them keep time.



Hour Hand and **Minute Hand** are simply the hands of the clock (game objects) which will be rotated around the Z axis to keep time. You must provide a game object for each (not doing so will cause a crash); if you don't want both hands you will need to provide and empty for the absent hand.

Time Scale is the factor time is multiplied by. The default of 60 will give a 24 minute day where each minute of real time will be an hour in the game world world. Similarly, 24 would give one hour days. To imitate Mojang you would give 72 for a 20 minute day, to imitate Bethesda you would use 20 for a 72 minute day, while 1 would make it completely real time.

Start Offset Hours tell the time the clock should start at (i.e., the time at the start of the game).

Time Provider is Scriptable Object derived from the public inner class *ClockTurner.ATimeForClock*; you must create this this yourself if you are going to use one. This is to hook the clock into a world time system if you have one. If this is left as "none," the clock will be based on Unity's *Time.time*.

Pre Processed Hours tells if time from a Time Provide is already providing the hour of the day. If so, Time Scale and Start Offset Hours will do nothing. If this is false it is assumed the Time Provider is giving raw seconds from the start of the game and will apply the time scale and offset.

Pendulum is an optional game object representing a pendulum which will swing continuously. If you the clock does not have a pendulum leave this as "none."

Max Pendulum Swing is the number of degrees the pendulum will swing in each direction. If there is no pendulum this will do nothing.

KFMath.cs

Some math functions for various usages.

`public static int ModRight(int a, int b)`: This will produce a modulo (remainder) but with negatives starting from the lower number as most naively would expect and as is more useful in many situations.

```
public static float Asymptote(float n, float start, float rate)
public static double Asymptote(double n, double start, double rate)
```

These will will take a number ***n*** and limit it asymptotically beyond a certain point. If ***n*** is less that or equal to ***start***, this will return ***n***. If ***n*** is greater that ***start*** the it will return a value approaching ***start***+***rate*** such that it would reach ***start***+***rate*** at ***n*** = infinity, and each time ***n*** is double it the difference from ***start***+***rate*** will be halved.

```
public static long GetLongSeed(this string str)
public static ulong GetULongSeed(this string str)
public static int GetIntSeed(this string str)
```

These will convert a string to number. If the string is a valid representation of number of the given type it will return that number. Otherwise it will return a hash of the string. This was created to primarily to generate seeds random number generators.

Shuffler.cs

This class contains convenience methods added to shuffle lists implementing `IList<T>`.

`public static void Shuffle<T>(this IList<T> list)` : This will shuffle a list using Unity's built in Random class.

`public static void Shuffle<T>(this IList<T> list, Xorshift random)` : This will shuffle a list using the Xorshift class from this library.

SpatialHash.cs

This class is for generating random numbers based on position in space. Methods are similar to typical random number generators, though lacking ranged forms and all taking either three or for coordinates. Technically, it bases its results on points in four dimensional space with integer coordinates (x, y, z, and t). The x, z, and optional y are for determining the location, while the t (for time) allows each point to have a full sequence of numbers. Holding all but one variable constant also allows for a random number sequence in which any number in the sequence can be found at any time, as many times as needed, without having to generate proceeding numbers.

This was originally created for procedural generation, but has also been useful in assigning hashes based on location for use in hash table such as the *Dictionary<K,V>* class, and may have other uses.

TransformData.cs

A struct to store transform data outside of a real transform.

Xorshift.cs

An alternative random number generator. This was created originally for creating a pool of random number from an assignable seed with would not be effected by the use of Unity's Random class. This was originally for procedural level generation, so that levels could be generated using a player supplied seed and not be effected by the use of random numbers generated for game play.

Simulation Cameras

https://github.com/BlackJar72/Simulation-Game-Cameras

This is a system of camera controls for use in simulation and strategy games, such as city builders, life simulators, turn-based / real-time strategy, and similar.

They are included mostly to provide fly-cams for the demo scenes. The scripts are with the demo, but are documented here none the less.

Features include the ability to swap camera types and a built in system for picking a creating events to process the results.

Camera Types

Classic Control (`ClassicControl.cs`): Control typical of city builders, in which the camera is moved by pressing against the edges of the screen. The mouse wheel zooms in and out, while middle mouse click will return the zoom level to default. WASD rotates the camera, while hold [shift] will change it to move based on WASD. If vertical flight is enabled, **Q** and **E** will raise and lower the camera.

Classic Control with Discrete Y (`ClassicDiscreteYControl.cs`): Similar to Classic Control, but with discrete Y levels that are moved between with **Q** (down) and **E** (up). This was designed for use in life simulation games where you might be moving between floors of a house or building.

First Person Controls (`FirstPersonControl.cs`): This is a first person camera controlled with WASD for movement and mouse for look direction, **[space]** to fly up and **[shift]** to fly down – similar to what you might find in a creative or spectator mode in a voxel based building game. Look direction only effects movement in the horizontal (x, z) plane, so you can look up or down without changing height. Perhaps some would rather build cities in this way as well?

First Person Controls with Discrete Y (`FirstPersonDiscreteYControl.cs`): Similar First Person Control, but with discrete Y level similar to Classic Control with Discrete Y.

Free Camera Control (`FreeCamControl.cs`): A first person fly cam in which you simply move in whatever directions you face. Flying "up" and "down" with **[space]** and **[shift]** also is available. Apparently, some people like this kind of camera, for some reason.

All cameras handle left click and right click in the same way, detecting when something is either clicked or held with either the left or right mouse button and sending an event that can be detected and used elsewhere.

Other Classes

ACameraControl.cs: The abstract base for all camera controls, implementing common functionality and allowing other camera controls to be created and used with the mode switcher.

EventConverter.cs: This handles events from the camera controllers and converts them into appropriate Unity Events.
ModeSwitcher.cs: This the basis of the ability to switch camera controls in game. To set up the camera with mode switcher, the camera should be be the child of an empty with this script attached. All desired camera controllers (from the above list) should them be added in the order they will be shifted through, and each need to be set to inactive. The camera should be added to the Player Eye field of each controller, and other variables set up as desired for each. The demo scenes contains an example that can be followed in the game object “Camera Holder.”

By default the **F** key is used to cycle through the camera controllers.