The Models

Organization of the Prefabs

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 kinomatic rigid bodies. Weapons come in various material types, with prefixed denoting the originally intended interpretation:

- P: primitive (wood, stone, bone)
- Br: bronze
- I: iron
- St: steel
- Bs: blue steel
- E: Elven
- 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 kinomatic rigid bodies attached. Of course, they can be made non-kinomatic, 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.

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 are available both under the standard Unity license as part of this pack and under the MIT license though (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.

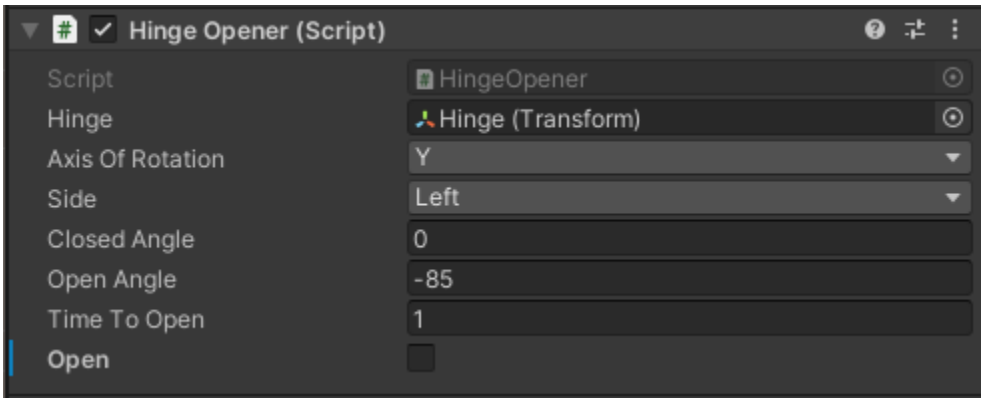
- ***StagedOpener.cs*** is a script that allows for moving a door (or other object) through several distinct stages. This was originally intend for things like secret doors that may first move back before then sliding downward or off to the side.
-

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

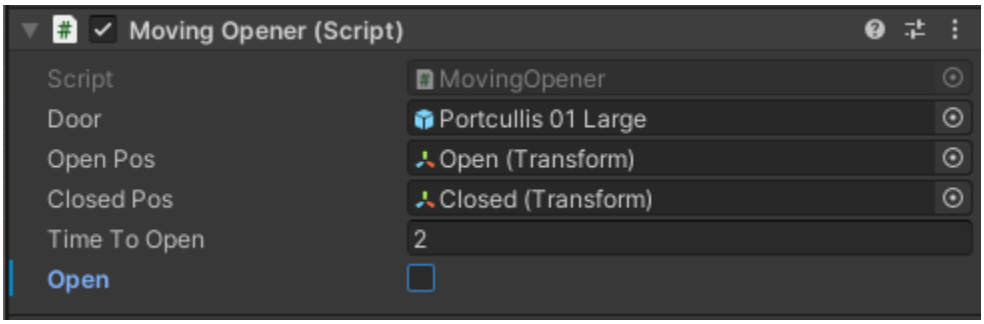
To use the scripts you will need a hierarchy of game objects in the prefab being opened.

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.

Damage System

This is a complete advanced health, damage, armor, and resistances system. It is based on the use of a dual-health system involving both wound damage and shock damage (as in, going into shock), loosely inspired by the table-top RPGs *TORG: Roleplaying the Possibility Wars* and *TORG: Eternity*. It is an improved version of the health system used in my game *Caverns of Evil* (available on Steam).

Integrating it with other game systems will require some programming. It was originally designed around 100 health as typical for a player character, perhaps as low as 50 at low level scaling to a few hundred at high level, and weapons typically having two digit base damages (e.g., a steel arming sword might do 30 base damage). See the file *ExampleForDamageSystem.pdf* if you want an example of kind of scaling this was designed around.

In this system, shock regens over time, while wounds require healing of some form, though code exists for healing wounds when resting for the night.

Note that “entity” in this system simply refers to any game object that moves through the world (such as a creature or projectile) with living entities such as NPCs or monsters being the relevant kind. It has *nothing* to do with Unity’s entity component system.

Damage from weapons (or other damage sources) is stated as a base damage, representing the averages damage. Actual damage done is calculate using RNG, total damage is 50% + (0% to 50%) + (0% to 50%) of the base damage, for a total of 50% to 150% of base damage. The use of two random numbers, both 0.0 to 0.5 is important as it makes the average damage 100% while also weighting the damage toward the center. This creates a triangular probability curve with some central tendency but not too much.

Attacks also have one or more of several damage types: physical, fire, electric, acid, poison, magic, cold, spiritual. Magic will not be a pure damage type, but is used to tag an attack with some other damage as magical – e.g., a fireball spell would be tagged as both fire and magic, while normal fire would be fire only. Some enemies may be effected differently based on the damage types applied. The magic tag is primarily for creatures immune to normal attacks (e.g., ghosts), though some creatures could exist specifically resistant to magical attacks. Status effects will also exist that change how much damage certain attack types do.

The order is as follows:

Apply stealth and power attack modifiers to base damage → roll damage → apply armor → apply natural modifiers → apply status effect modifiers → calculate wounds → apply damage to health

Armor effects are represented by the following pseudo-code:

```
damage = ((rolledDamage - (armor / 2))) * (1 - Asymptote(armor / 100, 0.5, 0.4));
```

Thus both subtracting half the armor and then reducing the result by one percent per point of armor up to 50. The `asymptote` function provides a soft cap of 50% and a hypothetical hard cap of 90% (for an armor value of infinity) for the percentage based reduction. This allows armor to be especially effective against attacks which it outclasses, but become increasingly less effective as the power of the attack increase while still always being useful (as long as the attack doesn't have a very high armor penetration). The asymptote function is defined as follows in C#, Java, or C++:

```
public static float Asymptote(float n, float start, float rate) {
    if(n > start) {
        float output = (n - start) / rate;
        output = 1 - (1 / (output + 1));
        output = (output * rate) + start;
        return output;
    }
    return n;
}
```

While armor penetration interpolates linearly between damage with and without armor. This is applied especially to blunt weapons and some types of spells, but also things like rondel daggers that are designed to stab between gaps. For

Wounds are calculated from shock with the following C# function (valid as Java or C++):

```
public static int CalcWounds(int shock) {
    if (shock > 12)
    {
        return (shock - 10) / 2;
    }
    else if (shock > 5)
    {
        return 1;
    }
    return 0;
}
```

Damage modifiers represent status effects, such as those from potions. These simply remove (or add) a percentage of total damage, and subtract the strongest relevant vulnerability from the strongest relevant resistance to find the total effect when more than one modifier is in play.

Damage adjusters represent natural resistances, weakness, and peculiarities of certain creature types, and should be created as needed for creatures. These allow for all sorts of adjustments, from simple resistances, to requirements, to more special transformations of damage. For example, and animated creature such as a golem or zombie, experience not shock and having no vital organs, might take only

wound damage (having shock set to 0), while an incorporeal creatures might only be harmed be attacks that have a magical and/or spiritual aspect.

EntityHealth.cs

Given to an creature to represent its health, damage can be applied by getting this as a component for a game object that has been hit and applying the damage if the result is not null (making it a good way t detect if something with health has been hit). To get the actual damage, use *DamageUtils.cs*.

DamageUtils.cs

This is the backbone of the damage system. The file contains the Damages struct used to represent damage, the DamageType enum used to define types of damage, and the IDamageAdjuster interface, as well as the DamageUtils class.

The DamageType enum is meant to be edited to customize for use in specific games. For example, "silver" could be added as a damage type for werewolves; this would be done in conjunction with creating a new damage adjuster. It is coded as [Flags] so damage can have multiple types, such as an attack that does physical damage with a fiery aspect (additional fire only damage could added and treated as a separate attack).

The Damages struct is meant to be used in place of the int or float that might be used to represent damage in a simpler system. It the damage type for determining damage changes by damage adjusters and damage modifiers.

DamageUtils is static and includes static methods for creating Damages from a damage source such as an attack. This is how damages are supposed to be created (not directly by constructor). To create Damages, call this method:

```
Damages damages = DamageUtils.CalcDamage(baseDamage, armor, damageType, armorPenetration);
```

This will generate Damages using the RNG and armor mechanics described above. The generated damages would then be passed through any damage adjusters or damage modifiers using *`DamageAdjustList.Adjust(damage, damageAdjustType)`* and `DamageModifiers.Apply(damages)`, then passed to *`EntityHealth.TakeDamage(damage)`*. This will cause damage to be taken, after which *`EntityHealth.ShouldDie`* can be tested to see if the entity is dead.

DamageAdjusters.cs

These are intended primarily to represent special resistances or weakness, or other adjustments innate to certain creature. It is intended to be edited with new custom adjusters created for creatures who may handle damage in a special way.

For example, after adding a “silver” type to the damage types (see above) a the following could be added and given as an adjuster to werewolves:

```
public static Damages RequireSilver(Damages damages) {
    if(((damages.type & (DamageType.silver | DamageType.spiritual)) == 0)) {
        return new Damages(0, 0, damages.type);
    }
    return damages;
}
```

Which would then be added to the DamageAdujstList and assigned to werewolves.

It includes a list of example, most of which are fairly generic; feel free to delete any you aren’t using. The *GHOSTLY* and *ANIMATED* adjusters are probably the best examples of why this exists, with the option to code any adjustment a developer might want, as these are both special conditions, though things like generic fire resistance is perfectly valid.

DamageModifiers.cs

This class if for damage resistances and weakness caused by status effects, including on player characters. It uses a system in which the highest relevant resistance and highest relevant vulnerability are found and added together (the vulnerability is basically a negative resistance, so this acts as subtracting the vulnerability from the resistance). This cannot reduce damage to less than 0, i.e., it will never become negative damage / healing, at most all damage will be avoided.

These modifiers do not stack, but instead use the strongest available modifier in each direction. They can, however, coexist, so taking a long lasting but weak fire resistance potion and then a stronger one with a shorter duration would give the higher resistance, but the weaker resistance would still apply when the stronger wears off.

Technically, it is actually the resistance that is a negative number, representing a fraction of damage to remove, while the vulnerabilities are fractions to add. The formula for damage after damage modifiers is: *damage * max(1.0 + weakness - resist, 0)*.

Note that physical damage is handled differently from other types, and will always be used in place of other if the damage type contains physical. If some other damage type is added it is expected to be treated as a separate effect, while the non physical aspect of the main effect is used for other purposes.

It is expected that each entity (or all those to which modifiers can be applied) will have it own instance of DamageModifiers to store such effects and process them.

Noise

These are special utility class for procedural world generation, and probably not the useful to most, but is included as they are part of the same utility library. These provide alternate, seed influenced noise fields. These produce noise in defined areas, that can be cached and used to create terrain. (Unlike other scripts in the KFUnityUtils library, these were ported from Java, where they were originally used to re-write the Minecraft world generator as part of the Climatic Biomes mod, which itself was a proof of concept for generating believable maps for a game.)

Other Classes (directly under util)

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.

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.