What is the SRPGCK?

The Strategy RPG Construction Kit (SRPGCK) is a set of Unity libraries and editor scripts capable of expressing a broad swath of the genre. The project has three main goals:

- To provide an “SRPG Maker”, especially for games in conventional styles, and even for networked multiplayer games;
- To support modular recombination of SRPG mechanics with arbitrary Unity scripts and frameworks; and
- To develop a pragmatic and experimental approach to game genre studies grounded in a genre’s works.

Many SRPGs should be implementable in the editor alone, without modifying or extending any of the library’s scripts. This should include turn- and tile-based games as well as realtime and continuous-movement ones, taking place on grid-based maps, conventional Unity terrains, and arbitrary sets of Unity colliders. Characters could be billboarded 2D sprites or animated 3D models.

What are the pieces?

The core SRPG Construction Kit is comprised of a set of about a dozen types of MonoBehavior. Some of these—Map, or the Arbiter/Scheduler/Character trio for example—can just as well be used in isolation of the other types.

Map

A Map is a 3D grid of tiles. Each tile can be painted with a different texture on each face. Tiles don’t necessarily have to be rectangular prisms—any upper or lower edge can be lifted to form ramps or taller tiles, and their edges can be inset to form thin bridges. Maps can also contain arbitrary Props, which can be outlined with invisible tiles to describe their collision areas. Similarly, a Unity Terrain can be placed as a Prop on a Map, and its collision can be outlined with tiles. A future version of the SRPGCK may automatically downsample the terrain into a grid of tiles. The visual collision area of a Prop should remain static.

Overlay

An Overlay is a specially-shaded signifier of a selected area, mainly used by Skills. Currently, there are Grid and Radial overlays that highlight the tops of grid tiles or a cylindrical or spherical surface respectively.

Arbiter Components

A Map may have a single child object responsible for administering the top-level rules of the game. This object is called an Arbiter, and contains a variety of important behaviors. Having more than one Arbiter per scene (and therefore having more than one Map per scene) is currently unsupported.

Arbiter

An Arbiter tracks the teams involved in the game. Each team has a unique integer identifier, and an Arbiter tracks which team identifiers are for local players, which are under local AI control, and which are the responsibility of networked players.

Schedulers

A Scheduler tracks the characters involved in the game and decides which ones are active at any given time. This can be done automatically, through player input, or using some other means. In other words, Scheduler behaviors determine the turn order. At the moment, Schedulers beside the CT Scheduler are not yet upgraded to take the Skill model into account.

CT

The Charge Time (CT) Scheduler is modeled after the turn ordering in Final Fantasy Tactics, with a few extensions. This is a turn-based scheduler with a global timer. Each tick, every character's CT is increased by that character's speed statistic. The first character to reach their CT limit becomes active. A character may move, act, or do neither on their turn, but they cannot move or act more than once on a single turn.
Every Character action may cost CT. There can be a base cost associated with moving, a cost per tile moved (currently not implemented), a cost per action performed, and a cost for merely activating and receiving a turn. CT costs may be applied immediately or deferred until the end of the turn.
This scheduler adds an additional behavior (CTCharacter) to its controlled Characters, which tracks whether that character has acted or moved on this turn. The CT Scheduler observes what Skills the tracked Characters use, sets the "Has Moved" and "Has Acted" flags and adjusts CT when Move Skills or Action Skills are applied, and looks for the use of a Wait Skill before ending the Character's turn, at which point it resets all flags.

Team Rounds—Pick Any Once

The Team Rounds—Pick Any Once Scheduler is based on the round/turn structure used in earlier SRPGs such as Shining Force. Teams take turns, and on each turn the player may select individual characters to perform their actions. Each character may only act once per round. The player may end their round at any time.

Team Rounds—Points

The Team Rounds—Pick Any Once Scheduler is based on the round/turn structure used in the modern hybrid SRPG Valkyria Chronicles. Teams take turns, and on each turn the player has a limited number of points. Activating a character requires spending a point, and while the character is active they will use up energy with movement and with the passage of time. Characters may be activated multiple times in a round, but each activation provides a lesser amount of energy. The player may end their round at any time.

AI—Currently under-implemented

Currently an underdeveloped part of the framework—and with the assumption that individual games will want to implement their own AI—the AI component acts as a virtual player for the characters of the AI-controlled teams (according to the Arbiter).

Grid AI

The Grid AI, for now, just moves characters around.

Network—Currently unimplemented

Currently an underdeveloped part of the framework—and with the assumption that individual games will want to implement their own networking code—Network coordinates all of the communication between players' computers.

Discrete Move Network

The Discrete Move Networking behavior assumes that all player actions can be represented as remote procedure calls. This is most appropriate for turn-based games. Well, it currently isn't appropriate for any game, since it doesn't do anything.

Formulae
The Formulae behavior is a named registry of mathematical formulae used in statistic, parameter, and effect calculations, which will be explained later.
Debug GUI
The Debug GUI is unsuitable for games, but can be useful for testing. It's currently implemented for the CT Scheduler.
Character
Each character to be used in a Scheduler must have a Character behavior. Characters have a name, an associated team identifier (and an "effective team identifier" to account for traitor, charm, or confusion effects), a list of allowed equipment slots, and a list of statistics. Characters also send animation trigger messages so that other attached behaviors can animate the character's movement and actions. Characters also store a transform offset to describe how high their visual component rests above the Map's floor.
Characters have a list of named attributes called statistics. Each statistic is backed by a Formula, which can either be a constant or a mathematical expression referring to other statistics on the same character or to formulas stored in the Formulae registry. For example, `2` or `health*4` or `f.calculatedMaxHP`, where the `f.` indicates a named formula lookup and the default lookup scope is the owning Character, which would explicitly be `c.`. These statistics are further modulated by the passive Effects of the Character's Skills, Equipment, or Status Effects.
Skill
A Character's available skills are aggregated from all of the Skills on the object itself and any child GameObjects, including Equipment or attached skill sets (GameObjects that simply contain groups of Skills). Skills each have a name, a group (separated by two slashes, e.g. "Act//Attack" or "Act//Black Magic//Fire", and potentially null), and a sorting priority within that group (any integer, with ties being broken by alphabetical ordering; actually using this priority is up to the specific game and its UI code).
Skills may replace other specific skills, nullifying their effects while they are active. Each skill may provide a single full replacement path and a replacement priority. For example, a pair of boots that override the default Move skill with a completely different skill called "Run" can provide "Move" as the replaced skill with a priority greater than 0. Conflicts between multiple replacement skills are settled based on a replacement priority.
Each Skill can be defined to have any number of Parameters. Parameters are the Skill's version of statistics: named Formulae which may be constant values. In the context of Skill Parameters, the default lookup scope is `skill.`, which refers to the owning Skill. To access the Skill's owning Character, use the `c.` scope. These Parameters may be used for any application-specific purpose.
A Skill may have any number of passive Effects which can influence the owning Character's stats. These are active while the Skill is available to the owning Character, but not if the Skill is being replaced.
Effect
Effects are modulations to a specific statistic name, applied in a chain with all other effects on the same stat at query time. Effects can augment (add or subtract), multiply, or replace their attached stat outright according to an attached Formula. Skill effects are

applied second in this chain, after Equipment effects and before Status Effects. Like Skill Parameters, the default scope is the owning skill.

Action

The Action Skill is a specialization of Skill (that may cease to exist in the future, depending on how the design evolves) that augments the base behavior with target selection and targeted effects. Currently, these all use a tile-picking UI affordance, but in the future, the SRPGCK ought to support skills that work in just as well in tile-based or continuous selection (or, indeed, quick-time events), in turns or in real-time.

Every Action Skill has a Strategy which is used to describe what tiles are valid to act upon, and what tiles the resulting action will effect. Strategy can currently select tiles based on minimum and maximum distances in the XY plane, the upward Z direction, and the downward Z direction, optionally blocked by walls or enemies (in the future, allies as well). These distances have default values, and can be overridden by Parameters of the same name on the skill. Strategy should be extended to support conical, line-based, pattern-based, and predicate Formula-based targeting and effects as well.

Range Parameter	Default
`range.xy.min`	1
`range.xy.max`	1
`range.z.up.min`	0
`range.z.up.max`	1
`range.z.down.min`	0
`range.z.down.max`	2

The Strategy is also responsible for describing, given a selected tile, which tiles should be influenced by the Skill's effect. Skills currently must have an inner effect radius of 0, but this will be changed soon and the Parameters will more closely match those for `range`.

Radius Parameter	Default
`radius.xy`	0
`radius.z.up`	0
`radius.z.down`	0

Action Skills must also interpret player input to select tiles. This is performed by a targeting IO. The provided Action IO generates an Overlay on the targetable tiles provided by the Skill's Strategy. Tiles on this overlay can be optionally selected by the keyboard and mouse, and the IO can also request confirmation of the selected tile. If confirmation is requested, the IO will also display which tiles will be targeted by the strategy's effect.

Due to limitations in Unity's serialization, if you wish to use custom subclasses of Action Strategy or Action IO, you must make your own subclass of Skill. This will be simplified in the future.

Action Effects

An Action Skill can apply Effects either to each target Character (the "Applied") or to the acting Character (the "Applier"). Currently, the Effect on the acting Character is applied for each target Character. This design will change to support once-per-use actor effects as well as per-target actor effects. Unlike normal Effects, these are applied at once and modify the underlying base statistic rather than adding to the Effect chain.

The Action Skill may decide between several groups of Effects to apply based on the `hitType` parameter, which is expected to return an index into the list of groups. For example, an Attack skill may dispatch between group 0 (miss), group 1 (dodge), group 2 (block), group 3 (parry), group 4 (hit), and group 5 (crit) based on a probability distribution function via `random{ (1-(c.accuracy+accuracy)):0; t.dodgeChance:1; t.blockChance:2; t.parryChance:3; c.critChance+critChance:5; default:4; }`. (In this context, the `t.` scope refers to the targeted Character, the `c.` scope refers to the owning Character, and unscoped lookups default to the `skill.` scope for Skill Parameters.) Each of these hit types can support completely different effects.

Targeted Effects used by an Action Skill (along with Reaction Effects described later) gain access to several additional, implicit Skill Parameters in the `arg.` scope:

Parameter	Meaning
`arg.distance`	Distance in XYZ from target.
`arg.mdistance`	Manhattan distance in XYZ from target.
`arg.mdistance.xy`	Manhattan distance in XY from target.
`arg.dx`	Distance in X from target.
`arg.dy`	Distance in Y from target.
`arg.dz`	Distance in Z from target.
`arg.angle.xy`	Angle in XY plane from actor to target.

To modulate an Effect based on the relative facing of the actor to the target, the `targeted-side{}` or `targeter-side{}` branches may be used in the Formula (for example: `targeted-side{front:0.5*accuracy; sides:1.0*accuracy; back: 1.5*accuracy}`).

Actions support a further extension to Effect: the Reactable Types. Reaction abilities can trigger off of these Reactable Types, which will often be categories such as "attack", "physical", or "ranged". Each Effect may have its own list of Reactable Types.

In the future, Action and Reaction Effects will be able to apply Status Effects to their targets.

Reactions
Any Skill or Action Skill may also support Reactions. Reactions may trigger on Reactable Types (as described above) or on changes to stats. If any trigger is met, the Skill will react. Reactions are like implicitly targeted Action Skills, so they also have an Action Strategy to determine which tiles should be targeted by the reaction. Currently, the attacker is targeted, but in the future other target selection options should be provided. If the attacker cannot be targeted, no reaction takes place.
Reactions require a similar set of Parameters to Action Skills (and have access to the same `arg.` scope, with the target being the initial attacker), but are prefaced by `reaction.`:

Range Parameter	Default
`reaction.range.xy.min`	1
`reaction.range.xy.max`	1
`reaction.range.z.up.min`	0
`reaction.range.z.up.max`	1
`reaction.range.z.down.min`	0
`reaction.range.z.down.max`	2

Radius Parameter	Default
`reaction.radius.xy`	0
`reaction.radius.z.up`	0
`reaction.radius.z.down`	0

Like Action Skills, Reaction Effects may also be dispatched by a `reaction.hitType` parameter, which indexes into Reaction Effects. In the future, Reactions and even Action Skills will be able to provide the name of another skill which should be used for the relevant Parameters. For example, a basic counterattack should work on exactly the same Parameters as the active Attack skill without requiring you to duplicate the Formulae or look them up in the Formula registry. Another future capability of Reactions will concern whether the Character turns to face their target during a Reaction.
Reaction Effects have access to additional lookup state that is not available to conventional skills. The `effect` variable stands in for the Targeted Effect to which this Reaction Effect is reacting; in particular, it represents the associated value (the amount of change in the underlying statistic). `effect()` can also be used with arguments to search for Targeted Effects on the reacted-against Action Skill, and to find their values (e.g. `effect(increase in health by magic get max)` will return the biggest magical heal performed by the reacted-against Action Skill. Currently, the `effect()` lookup cannot disambiguate between effects meant for the targeter and effects on the targeted

character. This may be done by the addition of an optional “`on {targeter|targeted}`” parameter to the `effect()` lookup or by the use of a character scope {`c.`|`t.`}before the `effect()` lookup.

Move

Character movement is performed via Move Skills. Like Action Skills, Move Skills each possess their own Strategy for picking where to move and their own IO for presenting these moves. Also like Action Skills, replacing these with custom implementations requires a custom subclass of Move Skill. Move Skills may also grant passive Effects and Reactions, just like Skills.

Every Move Skill has a Move Executor, a script that defines how a move through the Map should be performed by the Character. It understands three types of moves: Temporary, Incremental, and Normal moves. A Temporary move indicates that the Character’s final position is undecided, and that some visual proxy for the Character should be repositioned for feedback purposes. An Incremental move also indicates that the final position is undecided, but the Character’s actual position has changed. A Normal move indicates a final destination for the Character and, usually, the end of the Move Skill’s application.

Another responsibility of the Move Executor is to trigger animation hooks on the Character by calling TriggerAnimation() on the involved Character. This in turn dispatches a UseAnimation message from the Character to its attached components. Along with triggering animations, the Move Executor also adjusts the Character’s facing in accordance with the animations, which sends a UseFacing message from the Character to its attached components.

The default Move Executor will probably only work with grid-based movement. It generates a path between the centers of the tiles from the source to the destination and gradually moves the Character along that path, triggering facing changes and animations as it goes. It can also be configured to gradually move the Character on Temporary moves as well, although it does not distinguish between Incremental and Normal moves.

Like other Skills, Move Skills have a Move IO to interpret player input. The Move IO differs slightly from Action IO in that it dispatches requests to the Move Executor rather than immediately applying the attached Skill. Currently, only the Pick Tile Move IO is supported. Continuous movement via the keyboard or mouse and radial movement on the grid should also be supported in the future.

Many grid- and turn-based SRPGs in the style of Final Fantasy Tactics or Shining Force will be able to use the Standard Pick Tile Move Skill. This Move Skill employs a Pick Tile Move IO and a standard Action Strategy along with a default Move Executor to present an interface much like Final Fantasy Tactics. It requires two Parameters, which default to stat lookups on the Character, or 3 if those stats are absent (this also illustrates the use of the `if:;` and `exists()` syntax):

Move Parameter	Default
`range.xy`	`if exists(c.move): c.move; 3`
`range.z`	`if exists(c.jump): c.jump; 3`

Wait
The final type of built-in Skill is the Wait Skill. Wait Skills can carry no passive Effects, Reactions, or Actions. They implicitly have a Wait IO that lets the player select between one of four arrows, which must be provided as a prefab GameObject to the Wait Skill (this object must have four children named XP, XN, YP, and YN to indicate the axis and the direction).

Equipment
Equipment is simply any Game Object with the Equipment behavior. Each piece of Equipment has a name, a list of required slots (two-handed weapons, for example, may use "hand, hand"), and a list of equipment categories (e.g. "weapon, ranged, bow"). Equipment may also provide passive Effects to their possessor to modulate statistics. Characters will automatically wield any child Equipment objects at game start, and Equipment can be added and removed during play by calling methods on the Character behavior.

Equipment can also provide Skills to their wielder by attaching Skill behaviors to the Equipment, and can provide a list of built-in Status Effect prefabs which will be applied when the Equipment is added and removed when it is removed.

Passive Effects on Equipment use the Equipment's Parameters as a default lookup scope (`equip.`). Note that this is distinct from the `equip()` lookup which searches through equipment to find those matching certain criteria.

Status Effect
A Status Effect is a GameObject with the Status Effect behavior, applied to whichever Character has it as a descendant object. Status Effects can therefore carry particle systems or other indicators to show their existence.

Status Effects have an Effect Type, an optional time limit in ticks (either global ticks or Character speed-modulated ticks [right now, the time limit only works with the CT Scheduler!]), an optional "Unremovable" status that overrides this time limit, and a flag determining whether this Status Effect takes precedence over other Status Effects of the same type, or whether its effect is cumulative.

Status Effects carry a list of passive Effects. Within a Status Effect's passive Effects, the implicit scope is `c.`. Other Formulae can test for the existence of a status effect via e.g. `exists(t.status.poison)` to determine if the target is poisoned. The meaning of the `status.` scope outside of an `exists()` query is currently undefined.

In the future, Status Effects will also support Parameters and be able to perform per-global-tick, per-Character-tick, per-Character-activation, and per-Character-Skill-use action Effects targeted on the owning Character. Special hooks for Character death should also be provided, and they should be able to look up stats and Parameters on the responsible Character, Skill, and Equipment, if any.

Movable Camera
The Movable Camera is a basic isometric-view camera that focuses on an individual Character at a time. It can be repositioned, rotated, tilted, or shaken by script calls, and it will smoothly animate from place to place.

How do you use its behaviors from the editor?

Prefabs

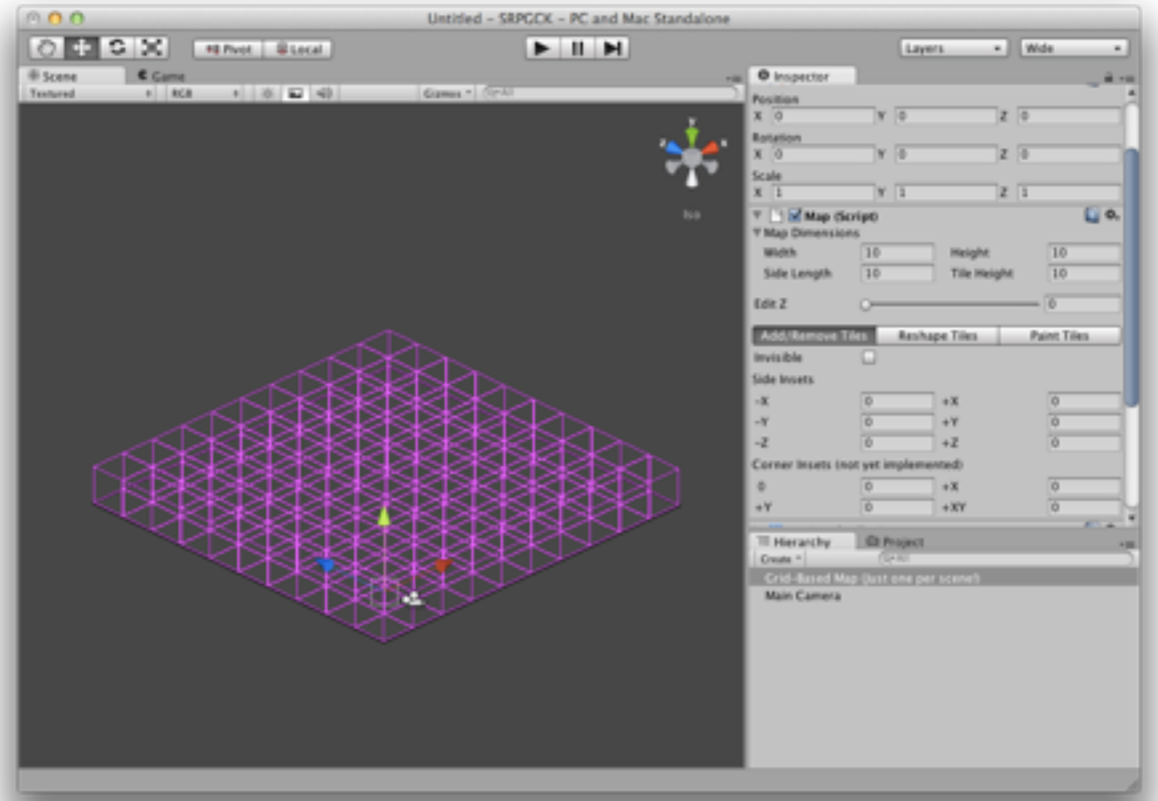
The SRPGCK comes with a handful of useful prefabs. The Grid-Based Map supplies an empty map with an Arbiter running a CT Scheduler, a Prop, and a Character wielding an Equipment that provides a Status Effect. The Movable Camera is a drop-in scriptable camera assembly. The Slow Effect illustrates an example Status Effect, and the Wait Arrows provide a default set of arrows for the Wait Skill.

Editors

The SRPGCK defines nine custom editors for its most important behaviors. It is highly recommended that any game-specific subclasses of these behaviors also get their own custom editors—see “How do you extend it?” for more.

Map

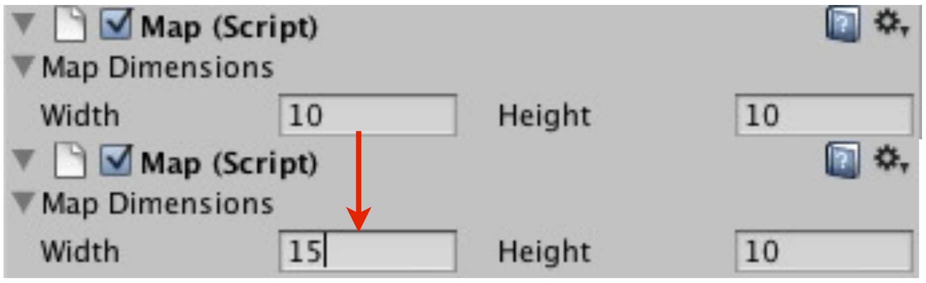
The Map editor is the most complex of the SRPGCK editor scripts. On selecting a Map, new UI is drawn in the Editor’s Scene view:



Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Map (Script)
Map Dimensions
Width 10 Height 10
Side Length 10 Tile Height 10
Edit Z 0
Add/Remove Tiles | Reshape Tiles | Paint Tiles
Invisible
Side Insets
-X 0 +X 0
-Y 0 +Y 0
-Z 0 +Z 0
Corner Insets (not yet implemented)
0 0 +X 0
+Y 0 +XY 0

It may be easiest to use the Map editor in iso mode. Each pink box represents a tile on the Map.

A 10x10 Map may not be appropriate for all SRPGs. To change the size of the Map in tiles, modify the “Width” and “Height” of the Map Dimensions.

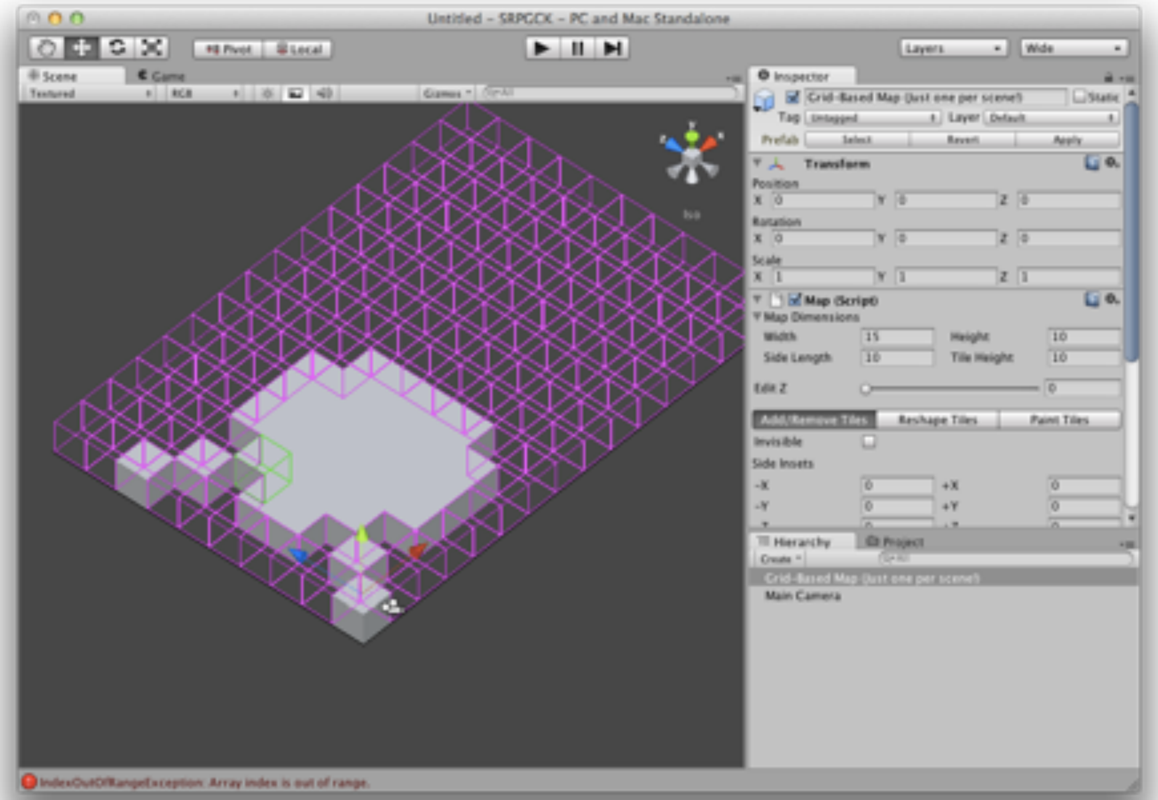


Map (Script)
Map Dimensions
Width 10 Height 10

Map (Script)
Map Dimensions
Width 15 Height 10

Currently, resizing the Map will scramble the placed tiles. Be sure to set your map size first!

To place tiles, click or drag among the pink boxes, ensuring that the Map is in "Add/Remove Tiles" mode.



Untitled - SRPGCK - PC and Mac Standalone

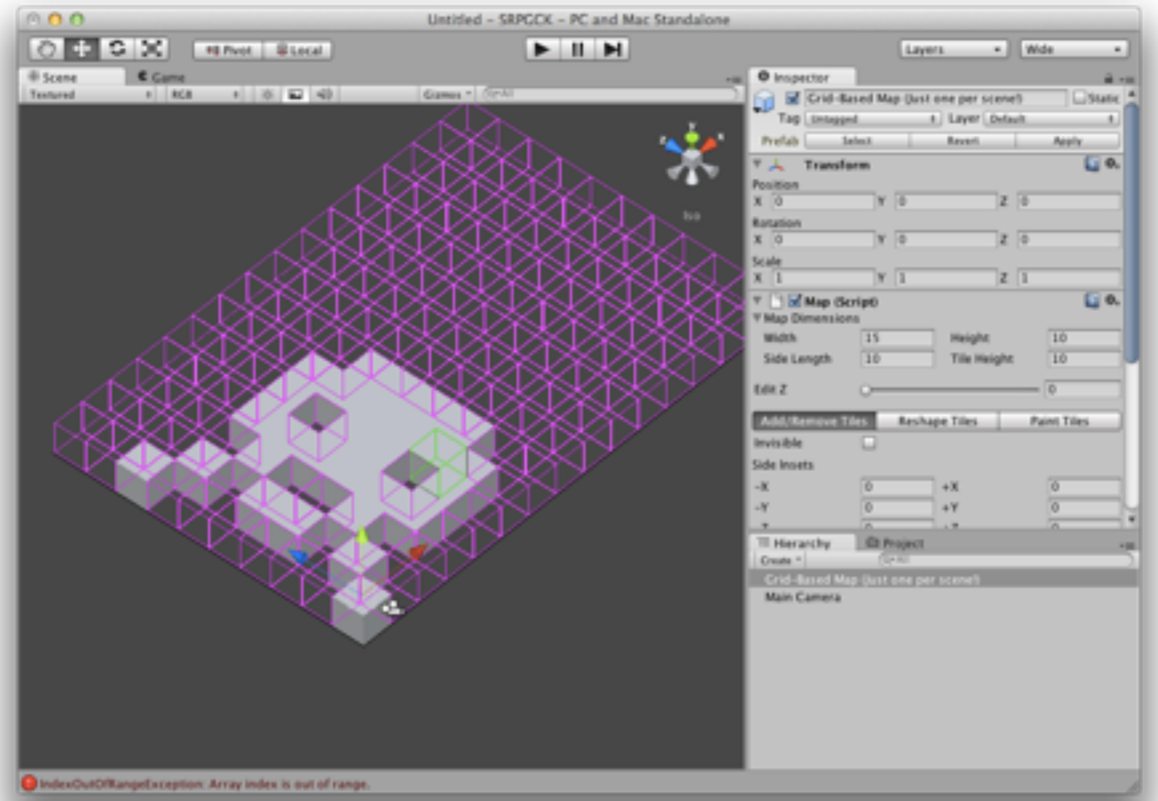
Inspector
Grid-Based Map (just one per scene) Static
Tag Untagged Layer Default
Prefab Select Revert Apply
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Map (Script)
Map Dimensions
Width 15 Height 10
Side Length 10 Tile Height 10
Edit Z 0
Add/Remove Tiles | Reshape Tiles | Paint Tiles
Invisible
Side Insets
-X 0 +X 0
-Y 0 +Y 0

Hierarchy
Grid-Based Map (just one per scene)
Main Camera

IndexOutOfRangeException: Array index is out of range.

Note that the pink boxes have been replaced by solid cubes.

To remove a tile, hold Alt/Option and click or drag.


Untitled - SRPGCK - PC and Mac Standalone

Inspector: Crid-Based Map (Just one per scene)

Tag Untagged · Layer Default

Transform

Position	X 0	Y 0	Z 0		
Rotation	X 0	Y 0	Z 0		
Scale	X 1	Y 1	Z 1		

Map (Script)

Map Dimensions

Width	15	Height	10
Side Length	10	Tile Height	10

Edit Z 0

Add/Remove Tiles · Reshape Tiles · Paint Tiles

Invisible

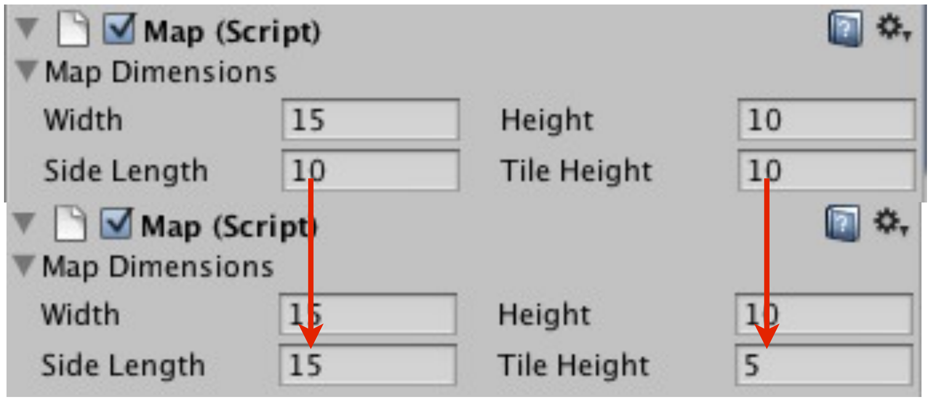
Side Insets

-X	0	+X	0
-Y	0	+Y	0

Hierarchy: Crid-Based Map (Just one per scene), Main Camera

IndexOutOfRangeException: Array index is out of range.

Perfectly cubic tiles may not be appropriate for all games. If your game is one of them, change the Side Length or Tile Height of your tiles. Currently, tiles must be the same width in X and Y.


Map (Script)

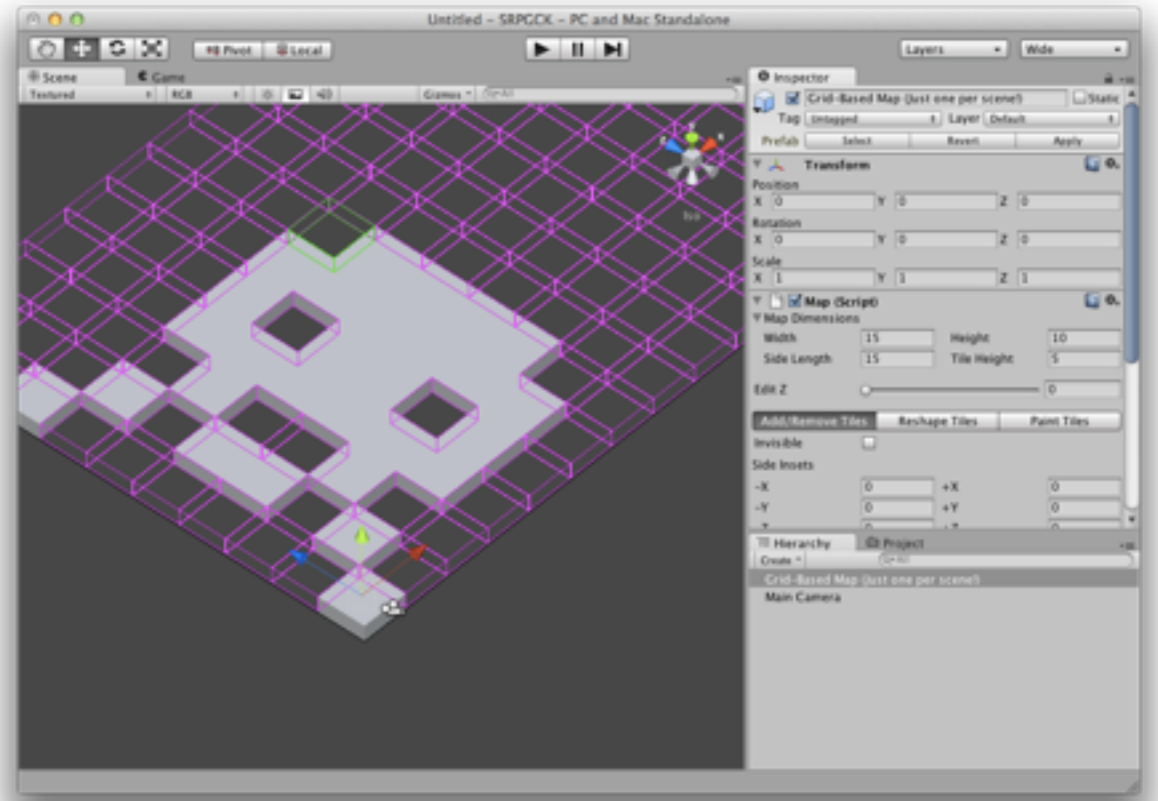
Map Dimensions

Width	15	Height	10
Side Length	10	Tile Height	10

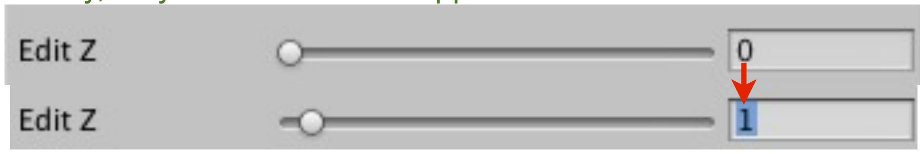
Map (Script)

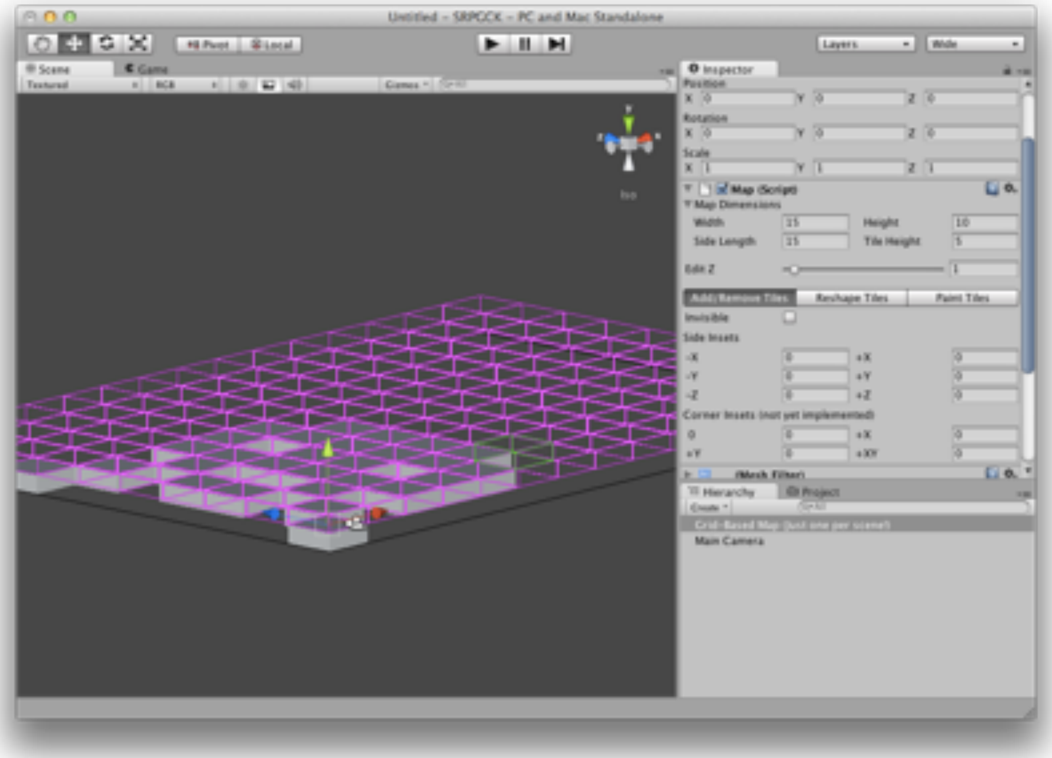
Map Dimensions

Width	15	Height	10
Side Length	15	Tile Height	5

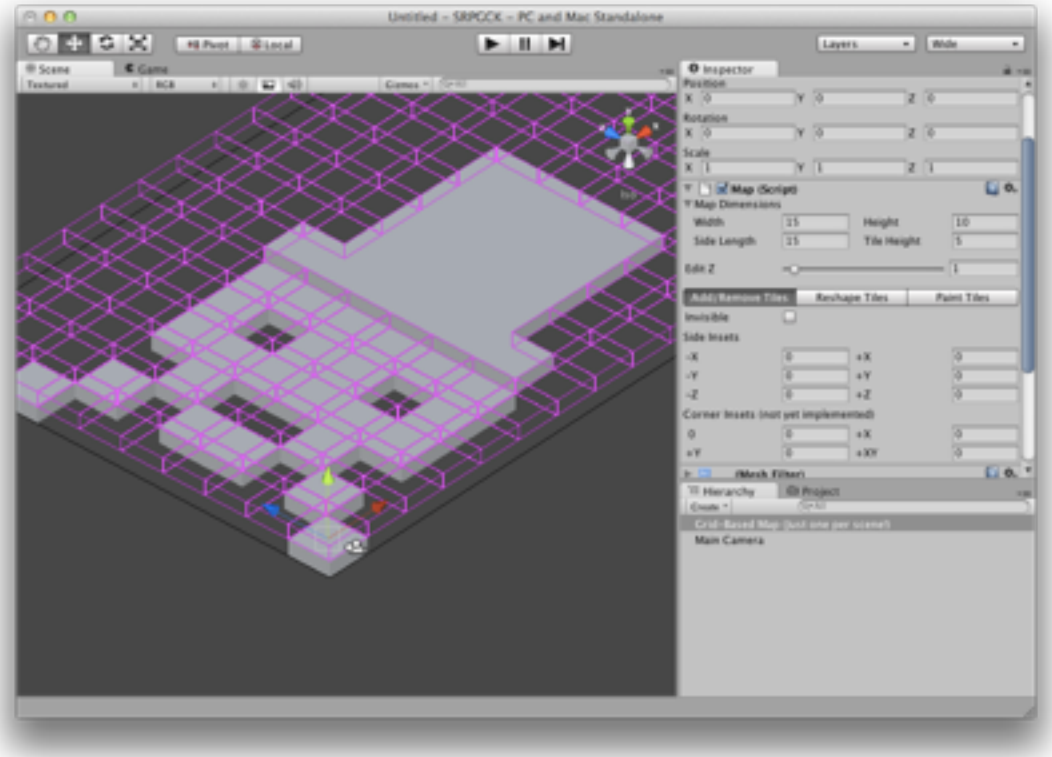


SRPGCK Maps are three-dimensional, but so far we've only modified tiles on the base level. The Map editor only edits one level at a time to simplify the interface. To change levels, adjust the Edit Z or use the up or down arrow keys while the Scene Editor is focused. Currently, only 20 Z levels are supported.





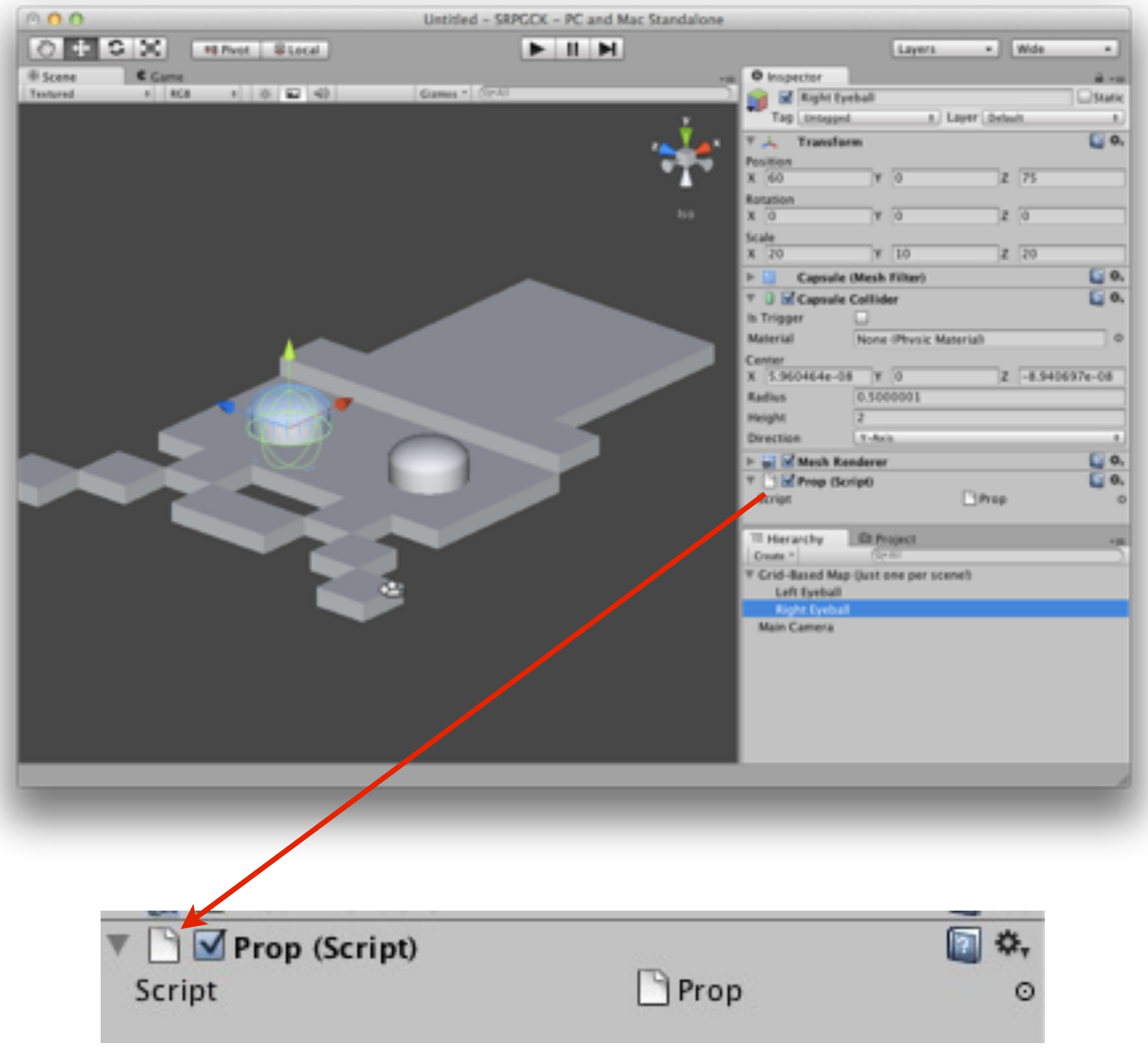
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Map (Script)
Map Dimensions
Width 15 Height 10
Side Length 15 Tile Height 5
Edit Z 1
Invisible
Side Insets
-X 0 +X 0
-Y 0 +Y 0
-Z 0 +Z 0
Corner Insets (not yet implemented)
0 0 +X 0
+Y 0 +XY 0
Grid-Based Map (just one per scene)
Main Camera



Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Map (Script)
Map Dimensions
Width 15 Height 10
Side Length 15 Tile Height 5
Edit Z 1
Invisible
Side Insets
-X 0 +X 0
-Y 0 +Y 0
-Z 0 +Z 0
Corner Insets (not yet implemented)
0 0 +X 0
+Y 0 +XY 0
Grid-Based Map (just one per scene)
Main Camera

Tiles may be placed anywhere on the 3D grid.

Maps may also contain arbitrary GameObject Props, so long as they have the Prop script.



Right Eyeball   Static
Tag Untagged   Layer Default

Transform
Position X 60 Y 0 Z 75
Rotation X 0 Y 0 Z 0
Scale X 20 Y 10 Z 20

Capsule (Mesh Filter)

Capsule Collider
Is Trigger
Material None (Physic Material)
Center X 5.960464e-08 Y 0 Z -8.940697e-08
Radius 0.5000001
Height 2
Direction Y-Axis

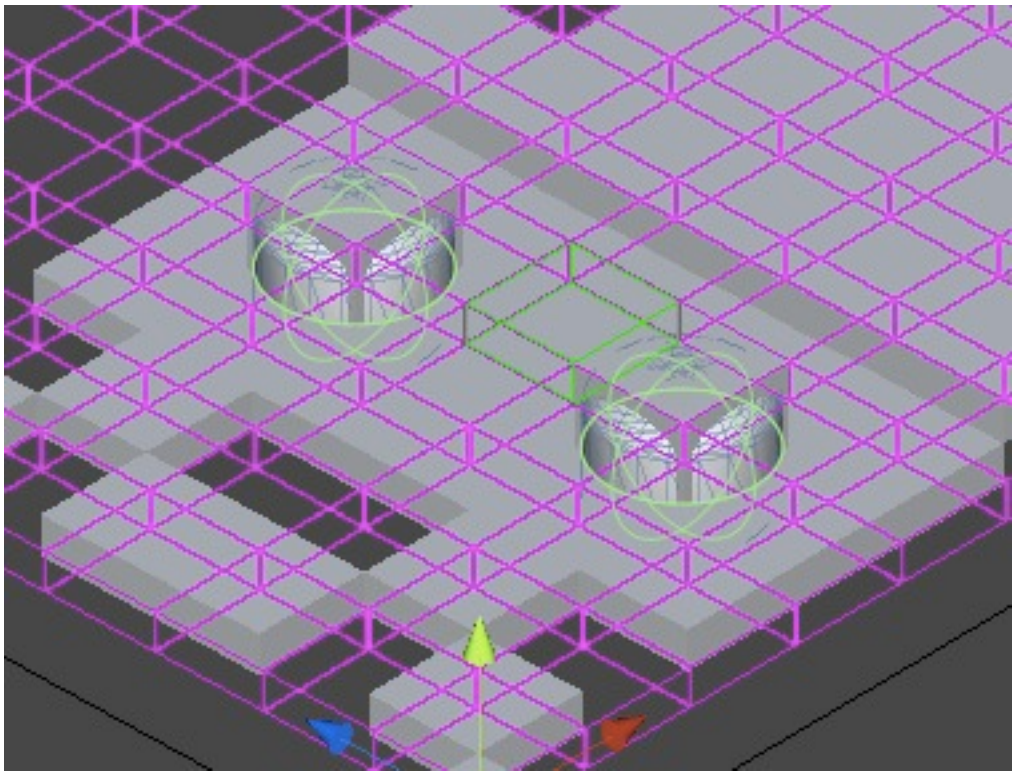
Mesh Renderer

Prop (Script)
Script Prop

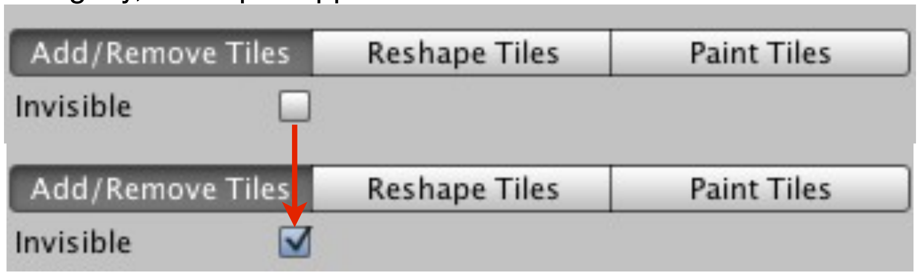
Grid-Based Map (Just one per scene!)
Left Eyeball
Right Eyeball
Main Camera

Prop (Script)
Script   Prop

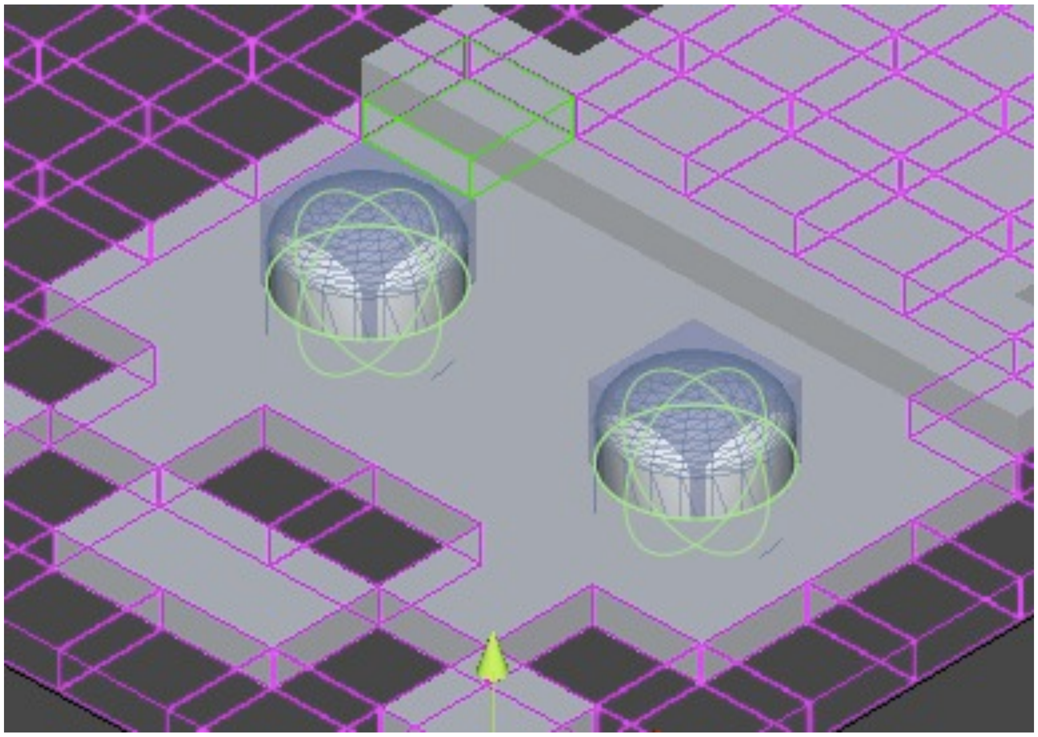
If you choose to use Props, you must also tell the Map what their collision should be. The best way to do this is by adding tiles to outline the Props.



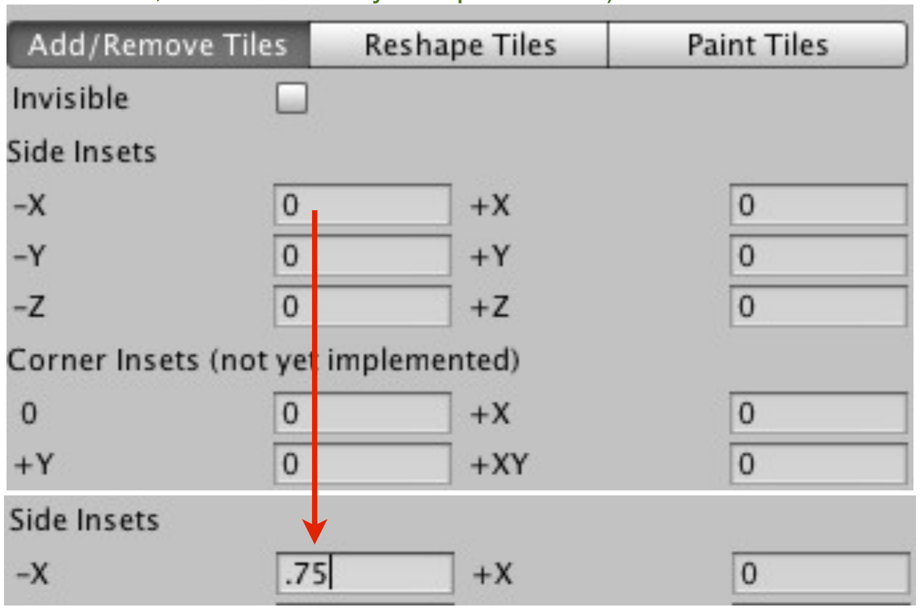
This can be unsightly, so Maps support invisible tiles.



You can draw new invisible tiles or draw over old tiles, turning them invisible.


Invisible tiles are drawn in a translucent blue in the Editor and are invisible at runtime.

Not all unusual shapes require custom Props, however! Any shape that can be made by insetting the X, Y, or Z dimension of a Map tile can be made in the Map editor (except for corner indentations, which are not yet implemented).



Invisible ☐

Side Insets

-X	0	+X	0
-Y	0	+Y	0
-Z	0	+Z	0

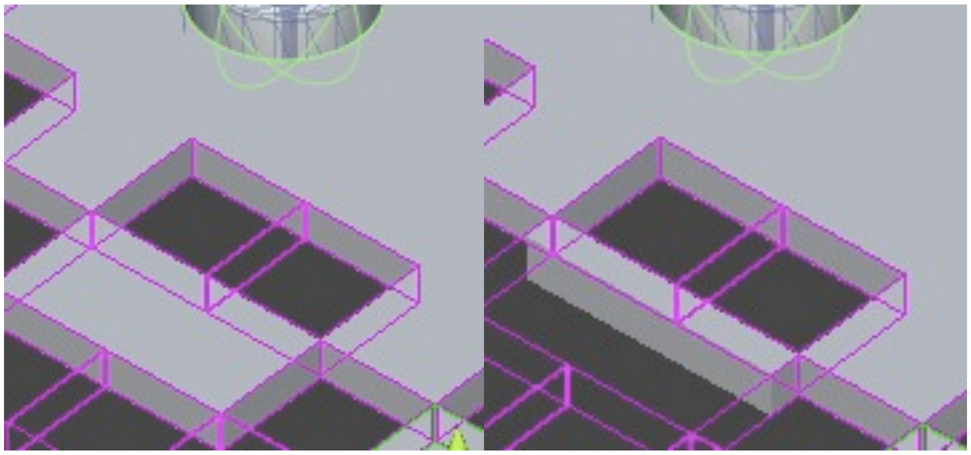
Corner Insets (not yet implemented)

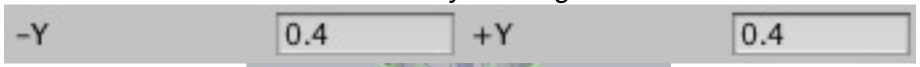
0	0	+X	0
+Y	0	+XY	0

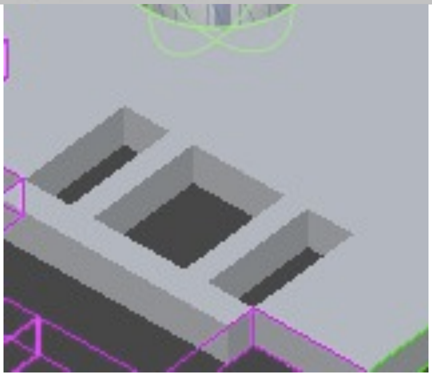
Side Insets

-X	.75	+X	0

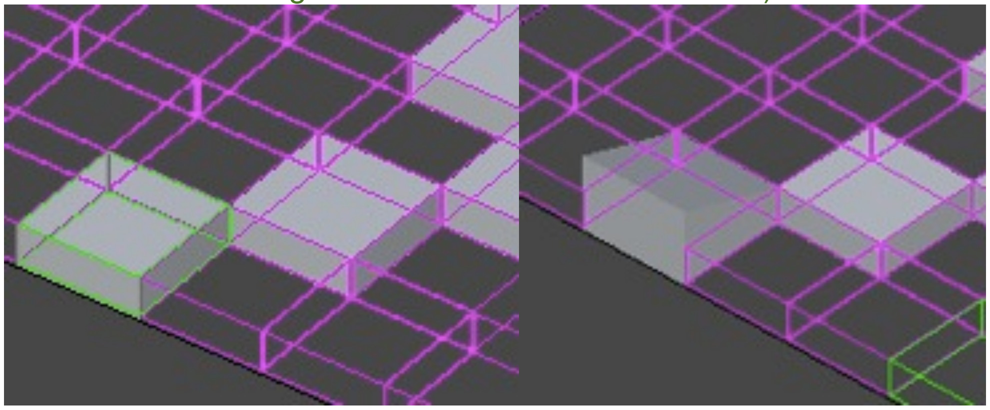
This will indent the -X side of the tile by 3/4. The - and + indents should add up to less than one.

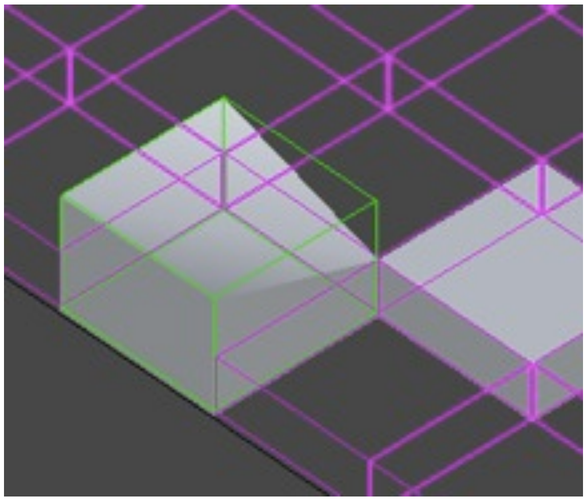

You can modify existing tiles...


-Y 0.4 +Y 0.4

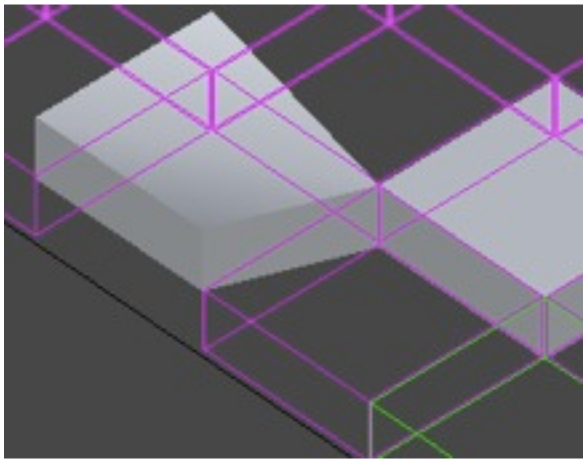

...or draw new ones (if you use the Reshape Tiles mode, it won't create any new tiles).

There are additional manipulations that can be performed in the Scene Editor. Holding Shift while clicking on a tile's edge raises it (the edge selection is currently quite buggy, so I don't recommend messing with this too much until it's fixed):

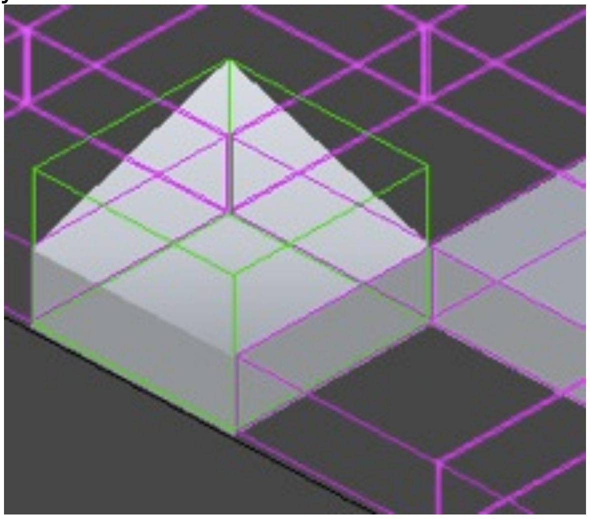

And you can raise multiple edges to form more complex ramps:



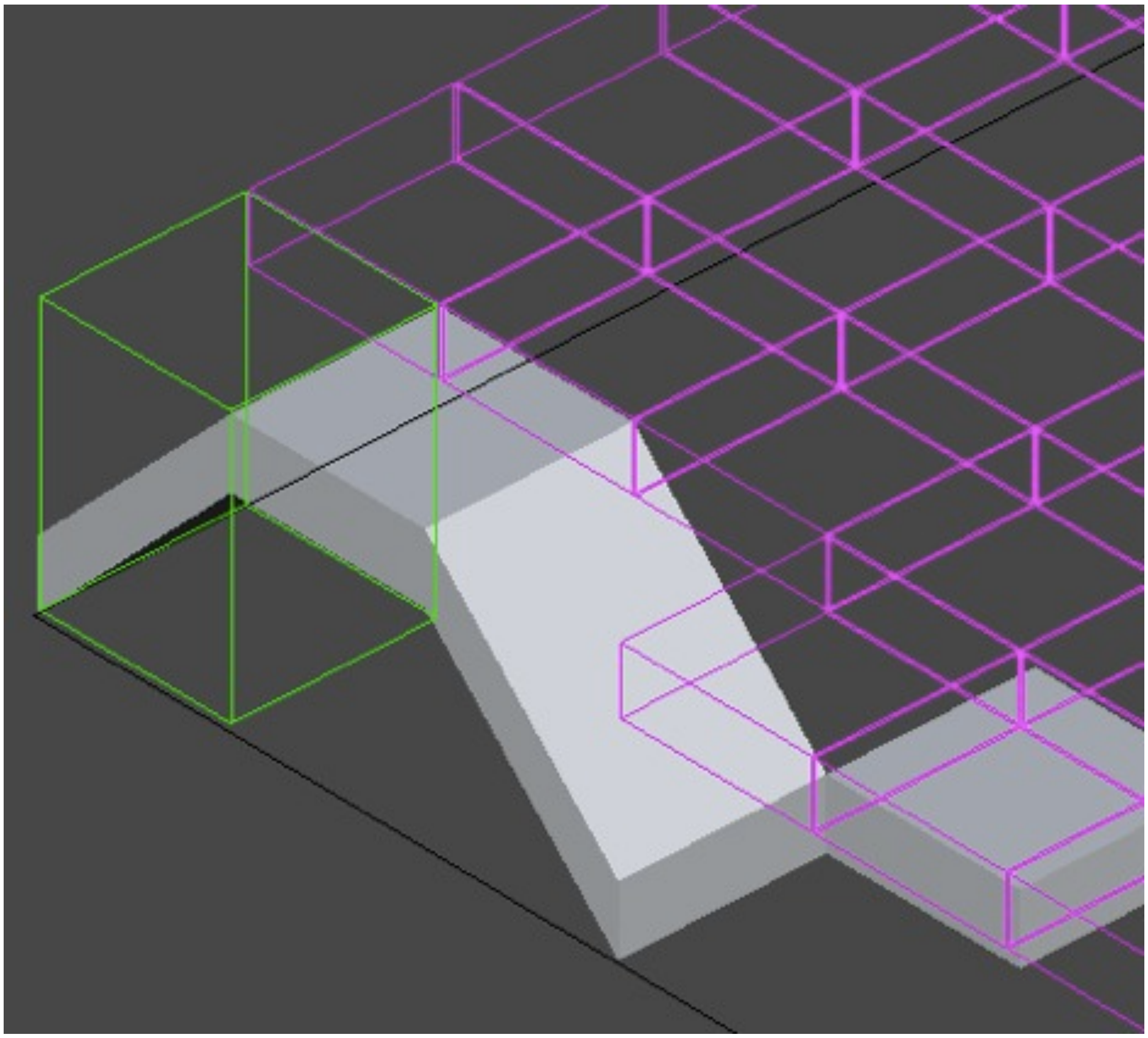
You can also Shift-click from the lower side of the tile to raise its baseline:



Option-Shift-clicking any side will lower it:

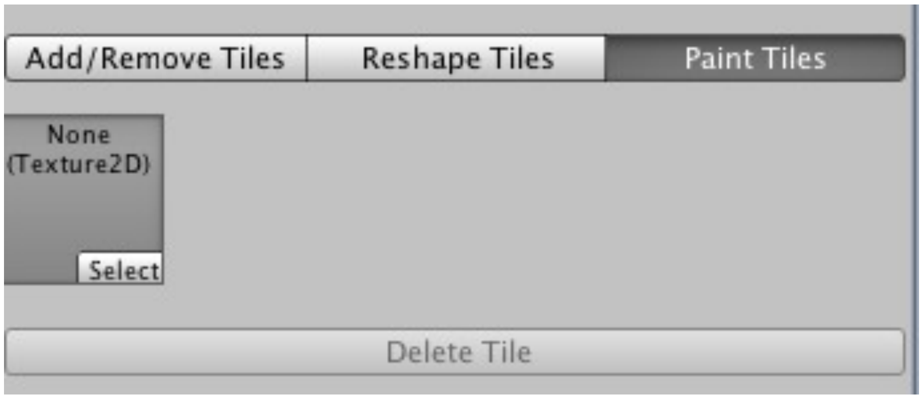


Tiles can be raised multiple levels, too:

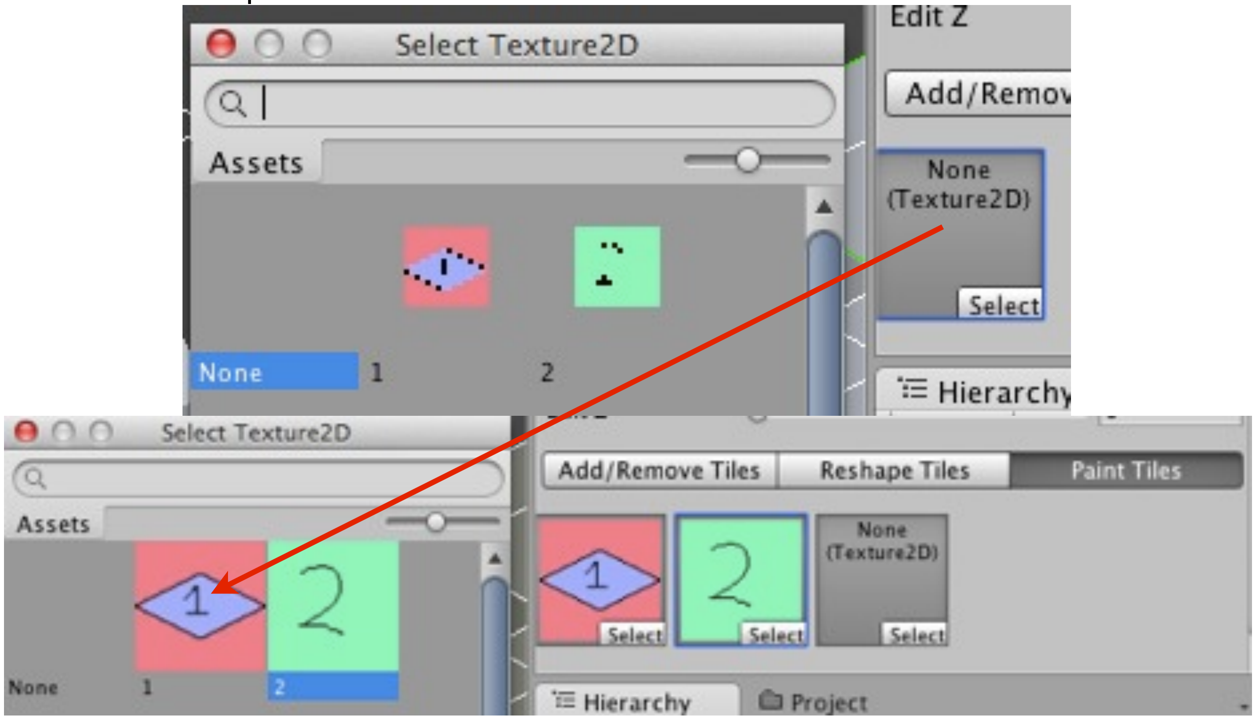


You can combine normal tiles with ramps to make fancy bridges!

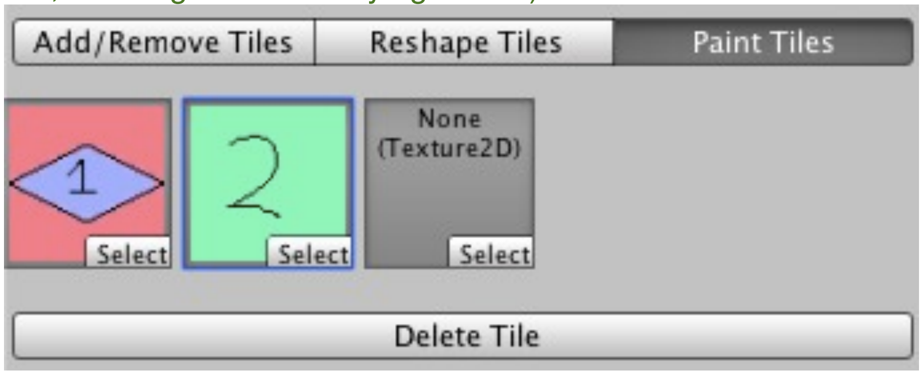
Finally, the faces of tiles can be painted with textures. To paint tiles, go into the Paint Tiles mode:



And add some square textures:

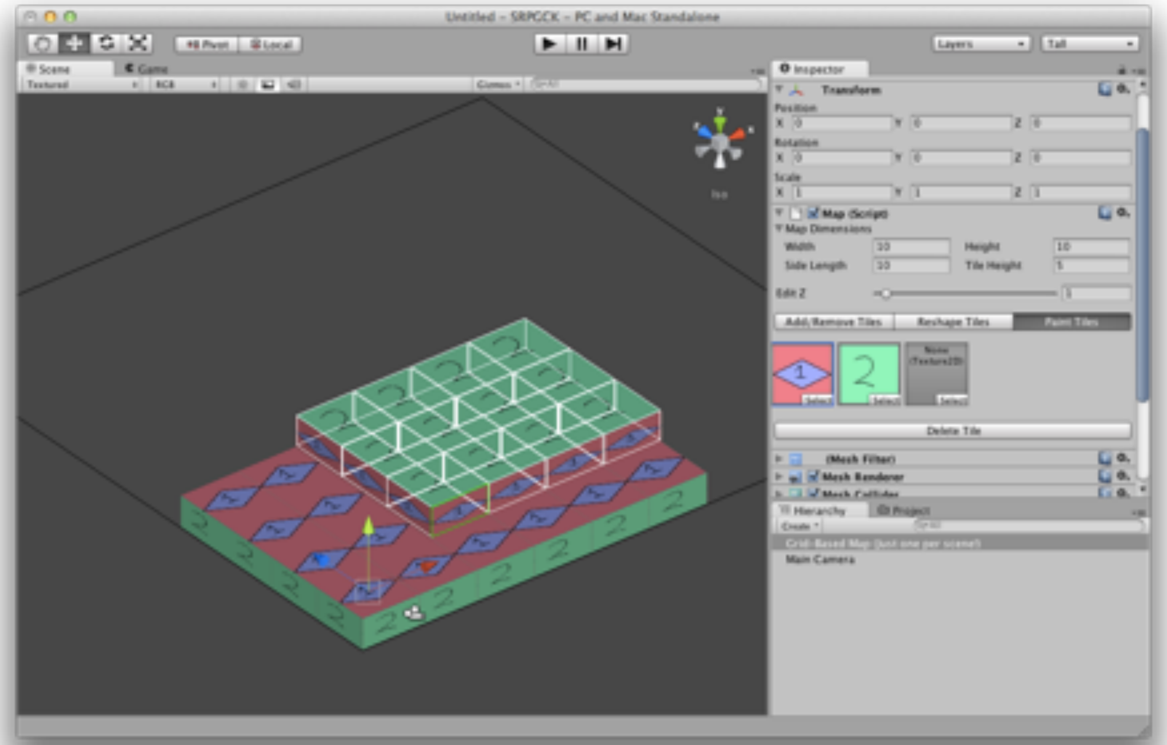


If you added a texture by accident, select it and choose "Delete Tile" (WARNING: Don't actually do this, as it might crash Unity right now.)



Note the selected highlight around texture 2.

To draw onto the faces of tiles, click and drag in the Scene Editor.



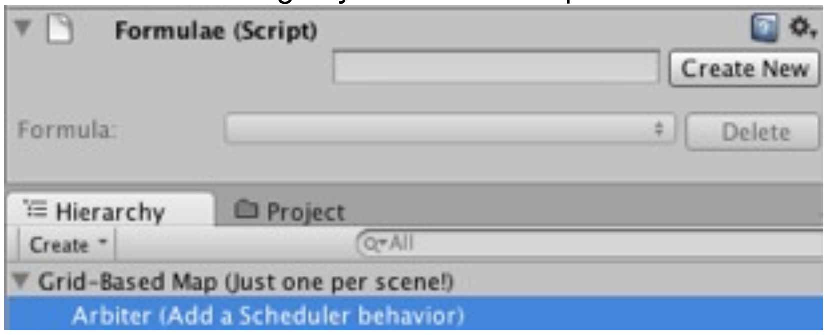
As in placing tiles, painting occurs on specific Z levels.

The Map editor is in a very early stage, and will see more improvements and less use of obscure modifier-key-clicks in the future. Currently, all six sides of a tile must be painted separately. This will change, as tiles will be given a single stamp that controls the appearance of all its sides. This stamp will also support arbitrary Parameters which will be queryable by other Formulae for things like terrain effects. Tiles should also support a “staircase” mode where they generate a stepped slope rather than a smooth one.

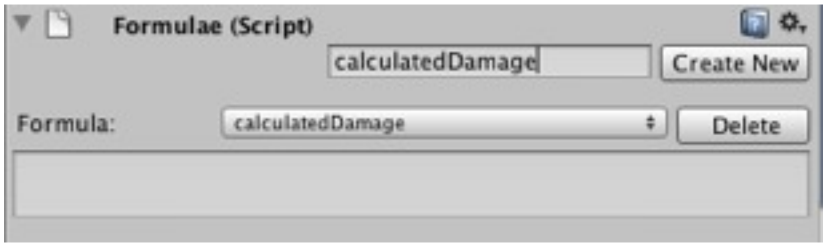
Formulae

Much of the work in designing an SRPG will consist of devising and tweaking the Formulae that drive its simulation. The Formulae editor lets you write and name Formulae to be referenced from other Formulae using the `f.` scope. These Formulae will use the default scope of their calling context.

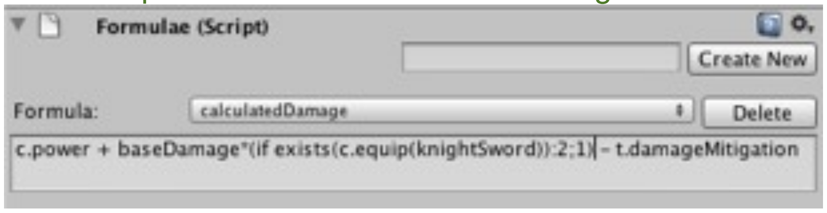
The initial state of the Formulae registry is somewhat sparse:


Formulae (Script)
Create New
Formula:
Delete
Hierarchy
Project
Create
All
Grid-Based Map (Just one per scene!)
Arbiter (Add a Scheduler behavior)

To add a new formula, start by typing in its name (which may contain dots or any letter) and press “Create New”.

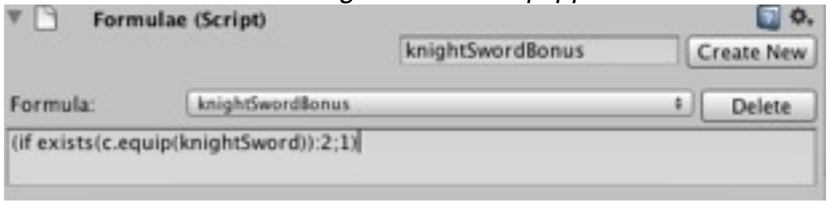


This will produce a new formula that you can edit. Any changes you make are applied immediately. Currently, the user interface will let you create multiple Formulae with the same name, but an exception will be thrown. This is a bug.

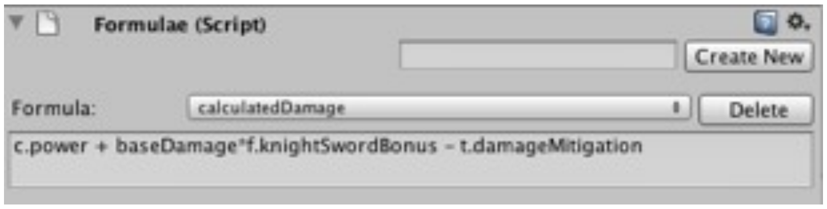


```
c.power + baseDamage*(if exists(c.equip(knightSword)):2;1) - t.damageMitigation
```

This Formula will work in any Action Skill Effect or Reaction Effect. It does double damage if you have a Knight’s Sword equipped!



```
(if exists(c.equip(knightSword)):2;1)
```



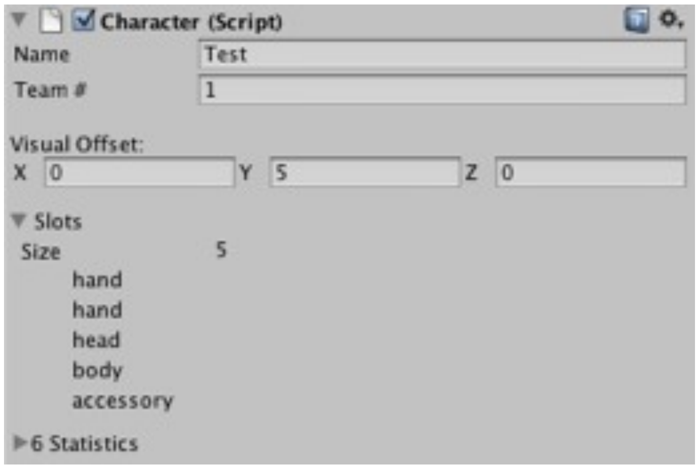
```
c.power + baseDamage*f.knightSwordBonus - t.damageMitigation
```

You can also factor out complex sub-Formulae into new Formulae.

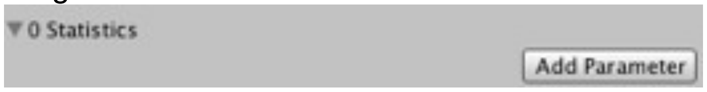
Currently, there is a bug in the Formulae editor: if you the Formula text field has keyboard focus and change the Formula popup, the previous Formula will replace the new selection’s Formula. This is undoable, but be forewarned!

Character

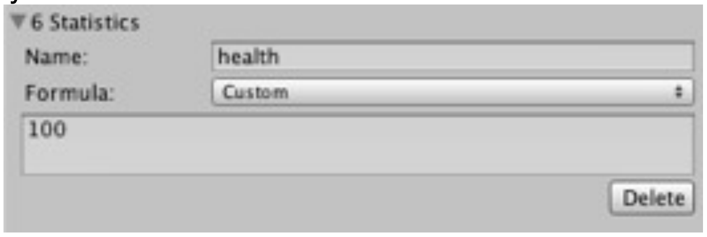
The Character editor provides simple access to the Character’s name, team ID, and visual offset, as well as its available equipment slots.


Character (Script)
Name: Test
Team #: 1
Visual Offset:
X 0 Y 5 Z 0
Slots
Size 5
hand
hand
head
body
accessory
6 Statistics

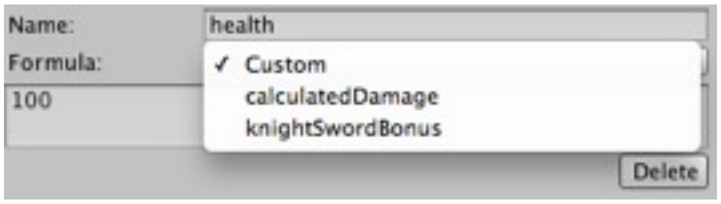
The Character editor also lets you define Statistics using an interface that will appear in many places throughout the SRPGCK. If you have made this Character from scratch, you may see something like this:


0 Statistics

To add a new Statistic, press “Add Parameter”. The built-in Character prefab comes with several stats already defined.


6 Statistics
Name: health
Formula: Custom
100

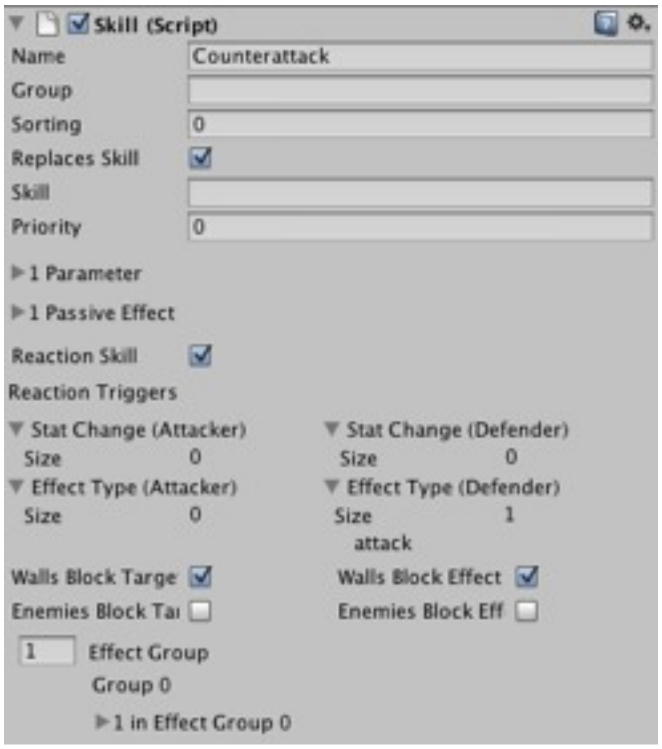
These Statistics may be implemented as custom Formulae (often returning constant initial values), as more complex calculations, or as references to entries in the Formulae registry.


Name: health
Formula:
100

Statistics can be removed with the attached Delete button, and new Statistics may be added by the “Add Parameter” button at the bottom of the list.
Characters will often have other behaviors, such as Skills, attached, but these use their own editors.

Skill

The Skill Editor is used for passive Skills, including Reaction Skills with no attached Action. It provides access to a Skill’s name, group, and sorting order, as well as whether it replaces any other Skills.



Skill (Script)

Name	Counterattack
Group	
Sorting	0
Replaces Skill	✓
Skill	
Priority	0

▶ 1 Parameter

▶ 1 Passive Effect

Reaction Skill ✓

Reaction Triggers

▼ Stat Change (Attacker)		▼ Stat Change (Defender)	
Size	0	Size	0
▼ Effect Type (Attacker)		▼ Effect Type (Defender)	
Size	0	Size	1
		attack	
Walls Block Targe	✓	Walls Block Effect	✓
Enemies Block Tai	☐	Enemies Block Eff	☐

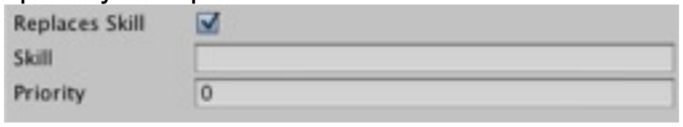
1 Effect Group

Group 0

▶ 1 in Effect Group 0

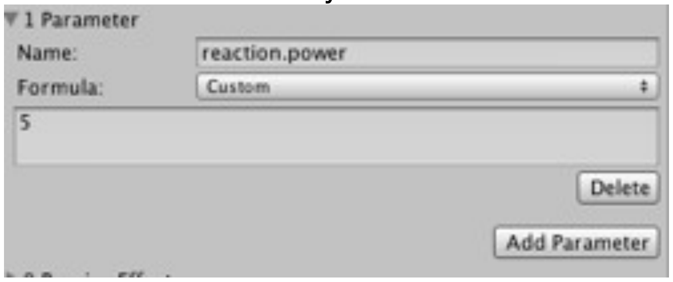
If the Skill is not a Reaction Skill, the UI below that toggle will not be shown.

If the Skill is configured to replace other skills, additional UI to declare the replaced Skill and the replacement priority is exposed:



Replaces Skill	✓
Skill	
Priority	0

Skill Parameters are added in the same way as Character Statistics:

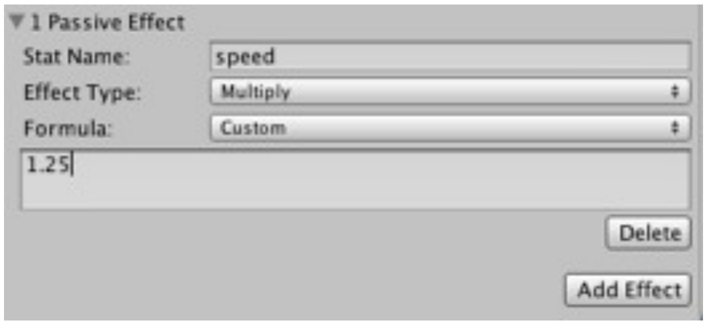


▼ 1 Parameter

Name:	reaction.power
Formula:	Custom

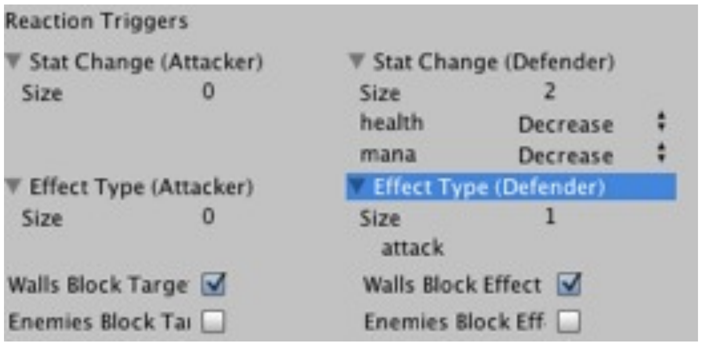
```
5
```

Skills, like Equipment, may also possess Passive Effects, such as this one which increases the owning character's speed by 25%:



▼ 1 Passive Effect
Stat Name: speed
Effect Type: Multiply
Formula: Custom
1.25

Reactions are configured in the Reaction Triggers section. They can key off of any type of stat change in the attacker or in the defender, or off of any particular type of effect triggered on the attacker or defender.



Reaction Triggers

▼ Stat Change (Attacker)
Size 0

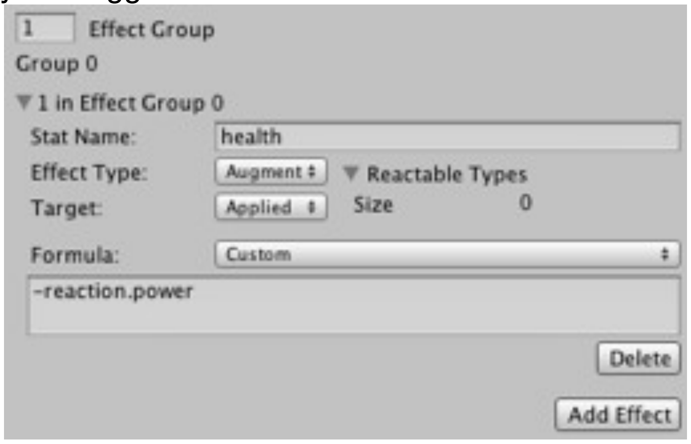
▼ Stat Change (Defender)
Size 2
health Decrease
mana Decrease

▼ Effect Type (Attacker)
Size 0

▼ Effect Type (Defender)
Size 1
attack

Walls Block Targe ✓
Enemies Block Tai ☐
Walls Block Effect ✓
Enemies Block Eff ☐

The “Walls Block Target/Effect” and “Enemies Block Target/Effect” fields have the same meaning as in an Action Skill’s Strategy section.
Reactions, like Action Skills, dispatch one of potentially several Effect Groups to perform their Reaction if they are triggered.

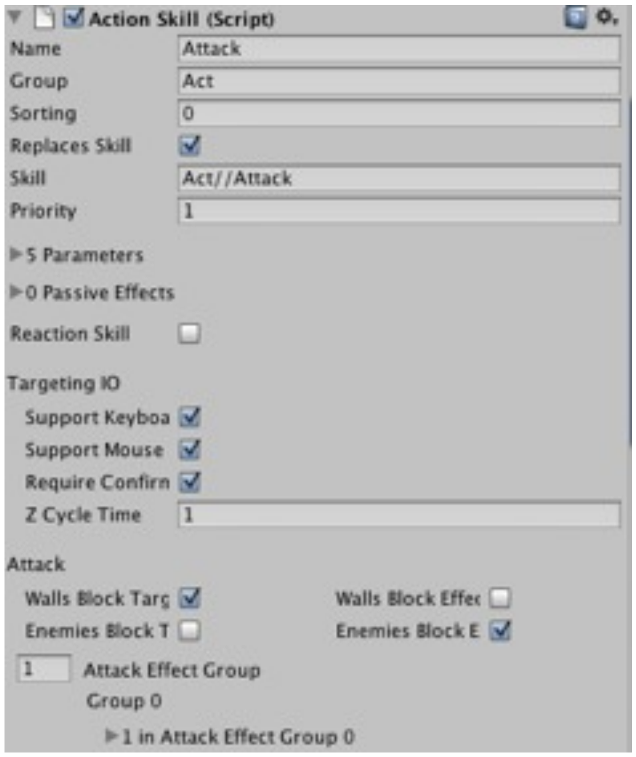


1 Effect Group
Group 0
▼ 1 in Effect Group 0
Stat Name: health
Effect Type: Augment
Target: Applied
▼ Reactable Types
Size 0
Formula: Custom
-reaction.power

Note that Reaction Effects, unlike passive Effects, may choose whether to apply to the Applied Target (in this case, the original attacker) or to the Applier Target (the counter-attacker). Reaction Effects can also define Reactable Types, so games that want to support counter-counter-counter-attacks may theoretically do so (although that would currently require a custom subclass of the Skill behavior).

Action Skill

The Action Skill editor is used for any Action Skill. It provides the same basic interface as the Skill editor, with some additions:



```
Action Skill (Script)
Name                Attack
Group               Act
Sorting             0
Replaces Skill      ✓
Skill               Act//Attack
Priority            1
▶5 Parameters
▶0 Passive Effects
Reaction Skill      ☐
Targeting IO
  Support Keyboa    ✓
  Support Mouse     ✓
  Require Confirn   ✓
  Z Cycle Time      1
Attack
  Walls Block Targ  ✓          Walls Block Effec ☐
  Enemies Block T   ☐          Enemies Block E   ✓
  1    Attack Effect Group
         Group 0
           ▶1 in Attack Effect Group 0
```

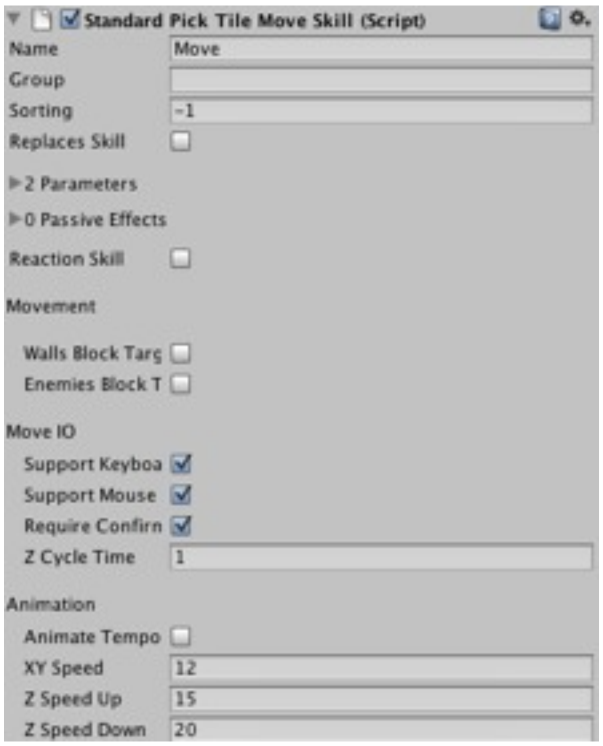
Note that this skill replaces Act//Attack. Skill groups are delimited by double-slashes. This is on a weapon that replaces the conventional attack skill with a new skill by the same name, though it could just as well be named something else.

The Targeting IO section of the Attack Skill editor determines certain features of the targeting interface, as described in “What are the pieces?::Skill::Action”. The Z Cycle Time, not described above, determines how long the IO waits before cycling the selected tile indicator between Z levels if there are multiple selectable tiles at the same XY position.

The Attack section works identically to the Reaction Effect section from the standard Skill editor.

Standard Pick Tile Move Skill

The Standard Pick Tile Move Skill editor extends the Skill editor with features relevant to the Standard Pick Tile Move Skill’s Strategy, IO, and Move Executor.

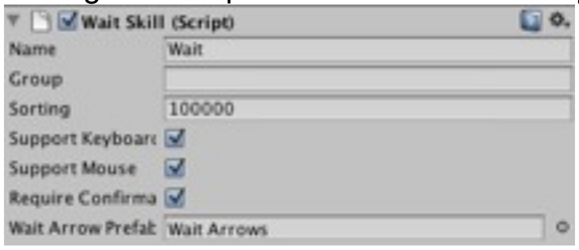

Standard Pick Tile Move Skill (Script)

Name	Move
Group	
Sorting	-1
Replaces Skill	☐
▸ 2 Parameters	
▸ 0 Passive Effects	
Reaction Skill	☐
Movement	
Walls Block Targ	☐
Enemies Block T	☐
Move IO	
Support Keyboa	✓
Support Mouse	✓
Require Confirn	✓
Z Cycle Time	1
Animation	
Animate Tempo	☐
XY Speed	12
Z Speed Up	15
Z Speed Down	20

Move Skills may also provide passive effects and reactions or replace other Skills. The notable features in the Standard Pick Tile Move Skill editor concern the Strategy (note that Move Skills may only affect their target tile, so there are no flags for the effect being blocked), the Move IO, and the Move Executor's parameters. "Animate Tempo" is the unfortunately-truncated "Animate Temporary Movement" flag. This is a bug.

Wait Skill

The Wait Skill offers few configuration options besides the basic properties of a Skill.

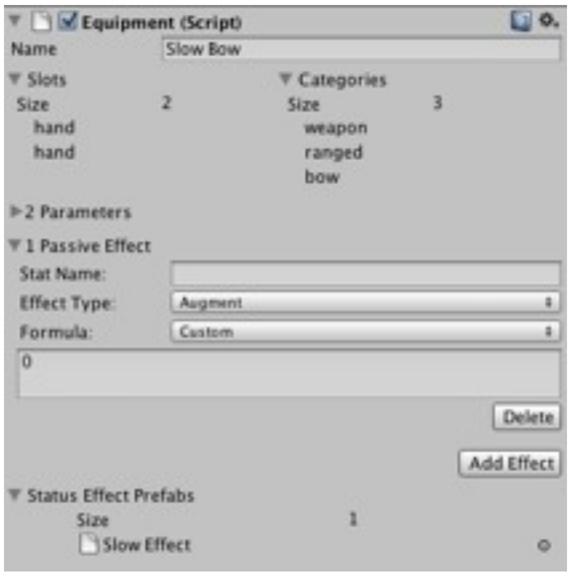

Wait Skill (Script)

Name	Wait
Group	
Sorting	100000
Support Keyboarc	✓
Support Mouse	✓
Require Confirma	✓
Wait Arrow Prefab	Wait Arrows

The Wait IO features, including the Wait Arrow Prefab, are the notable elements.

Equipment

The Equipment editor provides access to the Equipment's name, used slots, and categories, along with its Parameters, passive Effects, and attached built-in Status Effects.

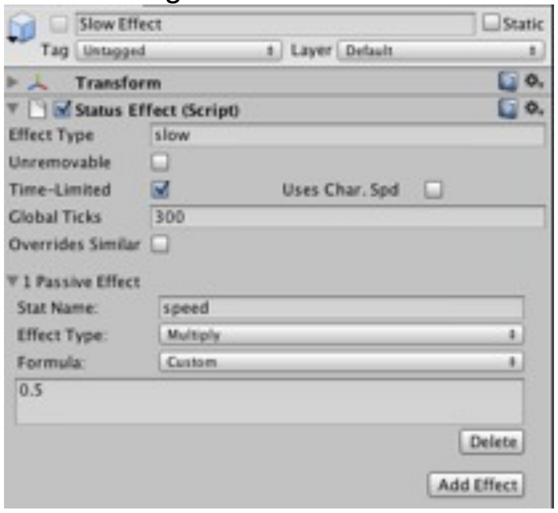


```
Equipment (Script)
Name            Slow Bow
Slots                        Categories
Size        2                Size        3
  hand                         weapon
  hand                         ranged
                               bow
2 Parameters
1 Passive Effect
  Stat Name:
  Effect Type:  Augment
  Formula:      Custom
  0
                                        Delete
                                        Add Effect
Status Effect Prefabs
    Size                      1
    Slow Effect
```

A piece of Equipment may take up any number of slots and be a member of any number of categories. It may have Parameters in the same way as Skills or Character Statistics, and passive Effects like Skills. The novel element of the Equipment editor is the Status Effect Prefabs section. These Status Effects are applied to the wielder when the item is equipped.

Status Effect

The Status Effect editor is currently quite simple, offering access to the effect's type, duration, permanence, and other flags.



```
Slow Effect                          Static
Tag  Untagged          Layer  Default
Transform
Status Effect (Script)
Effect Type        slow
Unremovable        ☐
Time-Limited       ✓          Uses Char. Spd  ☐
Global Ticks       300
Overrides Similar  ☐
1 Passive Effect
  Stat Name:    speed
  Effect Type:  Multiply
  Formula:      Custom
  0.5
                                        Delete
                                        Add Effect
```

Like Skills and Equipment, Status Effects can provide passive Effects.

How do you extend it?

Subclassing Scheduler

Subclassing Skill

Strategies and IOs

Custom Editors

Subclassing AI

Subclassing Network

Extending the FormulaCompiler

Appendix A: The Formula Language

The Formula Language is a small expression language with simple control structures used to define the rules of a game without requiring a bunch of subclasses and custom code. Formulae are compiled by the editor scripts from a textual representation into an abstract syntax tree, which is evaluated at runtime on every call. In the future, optimizations like constant folding and inlining may be in order, but for now that is very low on the project’s list of priorities.

Grouping

Formulae may be grouped for precedence by round parentheses `()`.

Constants

Any numerical constant, positive or negative, is permitted in decimal notation (no engineering notation).

Examples: `0`, `1`, `0.5`, `-20`, `1024`

Arithmetic

Arithmetic operations can be performed on any pair of Formulae.

Examples: `1+1`, `2^6`, `-1*-1`, `(2^(4/6)+9/8)*3^3^2`

Comparison

Formulae can be compared using standard expressions, resulting in either `0` or `1`.

Examples: `0 <= 1`, `10/2 < 9/3`, `4 == 2 + 2`, `0 == (1 != 1)`

Builtins

Other mathematical functions are provided as builtins:

`abs(F)`

The absolute value of F.

`root(F)`

The square root of F.

`root(F, G)`

The G th root of F.

`mean(F₀, ..., Fᵢ)`

The mean of $F_0 ... F_i$.

`min(F₀, ..., Fᵢ)`

The minimum value of $F_0 ... F_i$.

`max(F₀, ..., Fᵢ)`

The maximum value of $F_0 ... F_i$.

`random()`

A random value between 0 and 1.

`random(F)`
A random value between 0 and F.
`random(F, G)`
A random value between F and G.
`clamp(F)`
F clipped between 0 and 1.
`clamp(F, G, H)`
F clipped between G and H.
`floor(F)`
F rounded down to the nearest integer.
`ceil(F)`
F rounded up to the nearest integer.
`round(F)`
F rounded up or down to the nearest integer.
`any(F₀, ..., Fᵢ)`
A random value chosen uniformly from f_0...f_i.
`exists(Lookup)`
1 if the lookup would provide a valid result, 0 otherwise.
Control Statements
`if F: G; H`
If F evaluates to 1, the `if` evaluates to G; otherwise, it evaluates to H. Often seen wrapped in parentheses.
`targeted-side{`**branches**`}` **and** `targeter-side{`**branches**`}`
Dispatches to a given branch based on which side of the target the actor is facing or which side of the actor the target is facing, respectively. Branches have the syntax: `branch-type:F` and are separated by semicolons. Valid branches include the specific sides `front`, `left`, `right`, `left`, and `away`; the broader categories `sides` and `towards`; and the catchall `default`. `default` does not need to be provided, but if it is absent and no other cases match, the behavior is undefined.
`random{`**branches**`}`
Note the distinction from the builtin `random()`. Dispatches to a given branch based on a probability distribution function. Branches have the syntax `F:G` or `default:G` and are separated by semicolons. A random number between 0 and 1 is chosen, and each branch is expected to evaluate to its contribution to the probability function. For example, `random{0.1:0; 0.1:1; 0.2:2; 1/2:sqrt(3^2); default:4}` has a 10% chance of producing 0, a 10% chance of producing 1, a 20% chance of producing 2, a 50% chance of producing 3, and a 10% chance of producing 4. If the branches add up to more than 1.0, any branches after the 1.0 line will silently never be reached; if the branches add up to less than 1.0, there will be intermittent errors when high numbers are rolled.
Lookups
The real value of the Formula Language comes in lookups. Rather than having to go hunting through instance variables and component searches to find character statistics and equipment parameters, scoped lookups can grab predefined Formulae, dynamic

variables, or other values on an as-needed basis. There are eight defined lookup types, of which six serve as scopes for further lookups.

Lookup Type	Explanation	Required Scope	Implicit Scope
`f.`	Stored Formula.	None.	
`c.`	Character Statistic, Skill, Equipment, Effect, or Status Effect scope.	None.	
`t.`	Target Character Statistic, Skill, Equipment, Effect, or Status Effect scope.	None. Only valid in action and reaction effects.	
`skill.`	Skill Parameter or passive Effect scope.	`c.` or `t.`, except on Skill Parameters and passive Effects.	`c.`, if required.
`reacted-skill.`	The Skill that this Skill is reacting against.	None.	
`equip.`	Equipment parameter lookup.	`c.` Only valid in Equipment Parameters and passive Effects.	`c.`
`effect`	Reacted effect value lookup. Not a scope.	None.	
`Term`	Any other term not addressed above is a Stored Formula, Statistic, Parameter, or Status Effect name.	See Implicit Scope.	`c.`, `skill.`, `equip.`, or none, depending on context.

In addition to lookups, the Formula Language supports searches over more complex state. Searches (except `status.Type`) provide a number of filters, all of which are optional, along with an optional aggregation technique.

Search	Explanation	Required Scope	Implicit Scope
`equip().`	Finds `equip.` scopes and describes how to aggregate their gathered Parameters.	Character scope.	`c.`

Search	Explanation	Required Scope	Implicit Scope
`effect()`	Finds `effect` lookups and describes how to aggregate their gathered Parameters. Refers only to Action or Reaction Effects that have been applied in this action/reaction cycle.	`reacted-skill.`	`reacted-skill.`
`skill().`	Finds `skill.` scopes and describes how to aggregate their gathered Parameters. Not yet implemented.	Character scope.	`c.`
`status.Type`	Evaluates to 1 if the scoped Character is under the given Status Effect type `Type`. Currently only valid when used from `exists()` builtin.	Character scope.	`c.`

`equip().`
`Categories`
A comma-separated list of category names. To match, all of these categories must be included in the candidate equipment's categories.
`in Slots`
A comma-separated list of slot names. To match, candidate equipment must be in at least one of these slots.
`effect()`
`Changes`
A comma-separated list of Statistic changes (`any`, `nochange`, `change`, `increase`, or `decrease`). To match, at least one of these categories must match the candidate Effect's change.
`in Statistics`
A comma-separated list of Statistic names. To match, at least one of these Statistics must match the candidate Effect's changed Statistic.
`by ReactableTypes`
A comma-separated list of reactable type names. To match, all of these types must be included in the candidate effect's reactableTypes.
`skill().`—Not yet implemented
`SkillName`
A full path to a Skill or Skill Group, for example "Act//Action" or "Act//Black Magic//".

Aggregators

`get first`

Returns the first value in the search set.

`get last`

Returns the last value in the search set.

`get min`

Returns the minimum value in the search set.

`get max`

Returns the maximum value in the search set.

`get mean`

Returns the mean value of the search set.

`get sum`

Returns the sum of the values in the search set.

Appendix B: License

The SRPGCK is comprised of two components: the SRPGCK Editor Scripts and the SRPGCK Library. The Editor Scripts are licensed under the LGPL, due to a dependency on the LGPL'd Sasa.Parsing library (http://sourceforge.net/projects/sasa/). The Library is licensed under the 3-clause BSD license.

Since the source code of both the Editor Scripts and the Library are provided, Sasa.Parsing can be replaced with an interface-equivalent module, so I believe this confirms to the terms of the LGPL. Furthermore, since it is my understanding that editor scripts are linked into a separate assembly from game scripts, the LGPL's linking requirement is not violated.