Some modding docs for Ground Control II

by Johannes Norneby and lots of other people

About .fruit, .juice, .ice, and .loc

.fruit files are analogous to C++ header files, defining structure that the .juice files use. .juice files are analogous to C++ source files, and define actual data values. .juice files use the INCLUDE keyword to reference .fruit files, and can then use the structures defined in the .fruit files. .juice files allow us to define arbitrary hierarchical data structures that can be automatically loaded by programs and accessed freely in a key-value manner.

We have tried, as much as possible, to use .juice as the single data definition and scripting file format in GCII, in order to make development and modding more accessible.

You should generally not modify any of the .fruit files, as the game code is dependent on the data structures defined in these files. .fruit files exist to make sure that .juice files conform to the data structures that the game code requires to function properly. In the few cases where .fruit file modification is allowed, this will be explicitly stated.

While it is possible to directly modify .juice files in any text editor, the recommended method is to use the JuiceMaker tool, as it performs compilation and validation of the .juice files when they are loaded. Any compilation errors are a sign that someone has done some manual hacking, so please avoid doing so, as you will have to fix these problem manually in a text editor in order to get JuiceMaker to compile them properly.

GCII.exe does not read .juice and .fruit files directly, but rather reads of type .ice. .ice files are binary exports of .juice files (including all possible .fruit inclusions), and can be seen as "frozen" versions of the corresponding .juice file. You can think of the .juice files as the "sources", and the .ice files as the "exports". JuiceMaker will, when you save a .juice file, automatically export an .ice version of that file.

Additionally, in order to support localization, JuiceMaker will automatically generate .loc files for each .ice file that includes instances of the type LOCTEXT (localized text). If a .loc file is generated for any .ice file, then the .ice file is inherently dependent on the .loc file when loaded by GCII.exe.

Note that JuiceMaker itself does not support any type of UNICODE character input, but will correctly export any LOCTEXT instances to the UTF-8 .loc format. If you want to localize your mod, you will have to manually localize your .loc files in a text editor that can save files to UTF-8 format, taking care only to modify strings on the right side of the delimiting tab character on each line.

Note that the .juice file is ALWAYS the "original source", and any changes done to a .loc file will be overwritten the next time a .juice file is saved by JuiceMaker (and a new .ice / .loc pair thereby generated).

Generally, GCII.exe reads .ice & .loc files, XEd reads .juice & .fruit, and JuiceMaker reads .juice & .fruit files.

JuiceMaker

JuiceMaker is our general-purpose tool for editing .juice files. To open a .juice file, use File/Open (you can only open files of type .juice). Note that you may only open files in the immediate directory where JuiceMaker resides, as well as subdirectories. This is to ensure that all paths stay relative to the working directory of the game.

Basic Operation

The main view in JuiceMaker consists of an log / error list (a drop down list at the top), the left hand instance tree, which displays the hierarchy of instances in a given .juice file, and the right hand view, which displays custom integrated editors (more on these below).

The left hand view (the JuiceTree) shows the instances in the current "namespace" (each namespace corresponds to a single .juice file). You can explore the JuiceTree by expanding and collapsing the instance nodes in the tree. Certain nodes are editable, and you can edit them by right clicking on them (or pressing ENTER). If the node clicked on is indeed editable, an editor dialog will appear. The type of dialog that appears depends on the type of instance node you clicked on.

Instance Types

The instance types are as follows:

TEXT / NUMBER / DECIMAL / LOCTEXT ("Free" types)

These are fields that can be freely edited. Any text string is valid, but beware that in many cases a number is what the game engine expects. In the case of LOCTEXT, the contents of this field will appear in the exported .loc text in order to be localizable. Note that JuiceMaker does NOT support any form of UNICODE character input, but will export .loc files in UTF-8 format. All localization must be done manually in a text editor that supports UTF-8 format (see above).

For certain "free" instances in certain contexts, we have implemented "constrained" editing (an example of this is when editing projectile sounds for units). A dialog will appear that resembles the TYPE of dialog (see below) that will constrain the valid values to the contents of the list. The contents of the list that appears are often constructed by program code that parses external files (for example, for unit projectile files, **sound/agentsounds.juice** is parsed so that you can only reference valid sound ids). All of this behaviour is hard-coded into JuiceMaker.

FILE

FILE fields are constrained to actual file names in the working directory. You are not permitted to close this dialog if the text string references a file that does not exist. Pressing the Browse button will allow you to browse the working directory for files.

TYPE (User defined)

TYPE is analogous to the enum keyword in C++, and allows us via .fruit to create constrained value ranges (mainly to avoid typos when entering data). You can choose a valid value from the presented list.

SCRIPTCLASS (list)

A SCRIPTCLASS instance is a list, but can include many different types of objects (sort of like a template based linked list in C++, except that the template parameter can be one of many different specified types).

Editing a SCRIPTCLASS instance will display a list dialog where you can add new instances, and change the ordering (move up / move down) of the instances in the list. Pressing the Add button will display a list of all the valid content types for the SCRIPTCLASS in question. Individual instance names for the contents of a SCRIPTCLASS are supported, but are generally not (but sometimes) used by game code, and exist primarily for readability.

(CLASS)

A CLASS instance is analogous to a C++ class, but these are fixed structures and cannot be edited using JuiceMaker (nor would you want to, as the game code is dependent on these structures being fixed). CLASS structures (like much else) are defined in .fruit files. All aggregate structures you see in JuiceMaker that are not SCRIPTCLASSes are CLASSes.

Other input commands

You can generally rename instances in the JuiceTree (if permitted) by selecting and pressing CTRL + R.

You can generally delete instances in the JuiceTree (if permitted) by selecting and pressing DELETE. Note that it is generally a very bad idea to delete any instance that is not a member of a SCRIPTCLASS, as the game code relies on the existence of instances with specific types and names.

You can recursively copy instances of any kind by selecting in the JuiceTree and pressing CTRL + C. To paste, select a target instance and press CTRL + V. Note that you can only copy instances to a target of the same type. If the copy succeeds, the JuiceTree will be updated to reflect the changes, and if the copy fails (due to pasting to an illegal target type), nothing will change.

For SCRIPTCLASSes, you can copy single content instances from the list by selecting the object you with to copy, press CTRL + C, and then selecting the target SCRIPTCLASS where you want to add a copy of the instance and pressing CTRL + V. In this case you will be prompted to enter a new name for the copied instance.

Integrated Editors in JuiceMaker (plug-ins)

During the course of development, a number of "plug-in" editors have been developed to ease the editing of the various Juice structures that evolved. These editors are automatically invoked when an instance of the appropriate type is selected in the JuiceTree. These editors are displayed in the right hand view of JuiceMaker when active, and accept both keyboard and mouse input.

Spreadsheet Editor

For all SCRIPTCLASSes that allows a single type of content, a spreadsheet style editor exists to provide an overview of SCRIPTCLASS instances. When selecting a SCRIPTCLASS in the JuiceTree, the spreadsheet editor will be invoked if the SCRIPTCLASS allows only a single type of content (for example, lists of files, unit type lists, etc).

Input commands:

Numpad + / -: Increase / decrease horizontal cell size
Numpad * / /: Increase / decrease vertical cell size
Arrow keys: navigate the spreadsheet
Spacebar: expand / collapse CLASS structures within the spreadsheet
Left mouse button: select instance in cell (this will synchronize and select in the JuiceTree as well)
Right mouse button: edit instance in cell (this is the same as right clicking in the JuiceTree)

Gui Editor

For instances of type Gui (can be found in for example **guis/gui_frontend.juice**) a custom editor exists.

Input commands:

Right mouse button: select a widget
Left mouse button inside selected widget: click and drag to move the widget
Left mouse button outside selected widget: click and drag to resize the widget
Spacebar: toggle grid
Numpad +: increase grid size
Numpad -: decrease grid size
Numpad *: auto size selected widget to the size of the prefab (so that it appears one texel per pixel on screen. This is 800x600 resolution dependent I believe)

Gui Skin Editor

For instances of type EXP_Skin (can be found in **guis/guiskins.juice**) a custom editor exists. Every widget in the gui system uses prefabs to control the way the widget appears on screen. This custom editor is mainly used to edit these prefabs.

Since we wanted to support multiple resolutions easily, each widget is defined in a virtual coordinate space (16000 x 12000 in the gui editor) which is converted to 0.0 - 1.0 range horizontally and vertically when loaded by GCII.exe. For this reason, we needed stretchable prefabs.

Each prefab consists of 9 parts: 4 corners, top and bottom edges, left and right edges, and a center part. Inner and outer minimum and maximum points define these parts. The corner parts are always drawn one pixel per texel on the screen, while the top and bottom edges are stretched horizontally to the size of the widget, the left and right edges are stretched vertically to the size of the widget, and the center part is stretched in both dimensions.

By manipulating how big the different parts of the prefab are, the user can control which parts are stretched and which parts are not when the gui is rendered by GCII.exe.

Input commands:

This editor is a little non intuitive. It has three modes of operation: Define, Drag Outer, and Drag Inner

In Define mode, the left mouse button can be used to define the size of the currently selected prefab. Prefabs must be created and selected via the JuiceTree interface. Clicking and dragging will define the outer limits of the prefab.

In Drag Outer mode, the left mouse button can be used to drag the outer minimum and maximum points of the prefab. The nearest handle point will be selected regardless of where you click on the screen.

In Drag Inner mode, the left mouse button can be used to drag the inner minimum and maximum points of the prefab. The nearest handle point will be selected regardless of where you click on the screen.

Note that the outer minimum and maximum must always be small and larger than the inner minimum and maximum, respectively. The editor attempts to enforce this.

Arrow keys: pans the view (useful when zoomed)
Numpad +: zoom in
Numpad -: zoom out
Numpad *: increase the "edges" of the current prefab (adjusts the inner min and max points)
Numpad /: decrease the "edges" of the current prefab (adjusts the inner min and max points)
Spacebar: toggle the mode of the editor

Sound Editor

For instances of type MS_Sound (can be found for example in **sound/agentsounds.juice**) a custom editor exists. The editor displays a graph of sorts, displaying the various .wav file components of the sound, with a falloff envelope for each. Input commands are documented on-screen.

Fruit Files Overview

Below is a list of the .fruit files that the .juice files reference.

directory.fruit
gametext.fruit
campaign/campaign.fruit
guis/buyableagents.fruit
juice/agentclassid.fruit
juice/agenttype.fruit
juice/aimer.fruit
juice/aimessage.fruit
juice/aistate.fruit
juice/animationtype.fruit
juice/caidata.fruit
juice/camera.fruit
juice/decisiontreenode.fruit
juice/default.fruit
juice/excl_rep.fruit
juice/messageresponse.fruit
juice/ordertype.fruit
juice/state.fruit
juice/turrethelperclass.fruit
juice/weaponconfig.fruit
juice/vector2.fruit
juice/vector3.fruit
juice/vitalstats.fruit
massgatetext.fruit
messagebox_characters/character.fruit
guis/exp_skin.fruit
guis/guied.fruit
juice/agent.fruit
sound/ms_sound.fruit
ui/default_settings.fruit
units/damagemodel1.fruit
units/dropship.fruit
units/unittype.fruit
xed.fruit
xed/autoprop.fruit
xed/campos.fruit
xed/customjuiceeditors.fruit
xed/exodus_database.fruit
xed/exodus_map_export.fruit
xed/fhmgpresets.fruit
xed/icevolcano.fruit
xed/meldtable.fruit
xed/missionstats.fruit
xed/script.fruit

xed/scriptlist.fruit
xed/supportthings.fruit
xed/weathers.fruit
xed/xinstancecontainer.fruit
xed/xinstances.fruit
xed/xlandscape.fruit
xed/xmap.fruit
xed/xprop.fruit
juicemaker.fruit

Juice Files Overview

Below is a case-by-case description of the various .juice files that the game uses. We have tried to use descriptive data member names throughout, so in most cases it should be easy to figure out what things are.

directory.juice

This is the main "directory" that GCII.exe loads on startup, and any and all other data references that the game needs can be traced from this file. This is, aside from the .sdf files, the only hardcoded asset reference in GCII.exe.

Directory.juice contains two data chunks at the root level, ExodusData and GetFilesIncludes. GetFilesIncludes is a data chunk used by internal asset tracking tools, and is not of interest to modmakers.

ExodusData is the main data chunk for the game. Here you will find top-level application data configurations and references to other asset files. In all cases where you see an .ice file being referenced, you can view the corresponding .juice file to further explore the data structures used by the game.

Under ExodusData you will find, among other things, the following data (not all data is explained below):

myGuiMouse - this controls mouse cursor settings for the entire application (the mouse cursor is a global resource)
myCampaignFile - specifies all campaign contents
myMultiplayerMissionsFile - specifies all multiplayer missions
myGuiFiles - this is a list of the various user interface scripts
    Mos - the MASSGATE screens (Massive Online Services)
    Frontend - the frontend screens (not ingame)
    Multiplayer - multiplayer specific screens, like the lobby and server configuration screens
    Ingame - ingame screens
    IngameSlim - the "slimmed down" ingame screens
myRenderFiles - assorted global rendering resources
myPlayState - configurations for the various ingame subsystems
myUpkeepLevels - controls how upkeep works in the game

errors.juice

Includes localized error strings for all critical application errors.

gametext.juice

Miscellaneous localized strings.

campaign/campaign.juice

This file defines all campaigns in the game. A campaign can contain both missions and cutscenes (in myItems).

Note about profiles:
When profiles are created, GCII.exe will read the campaign file and copy this information into the profile, and progression is recorded against this data in the profile. This means that once a profile is created, any changes to the campaign file are not visible to the user unless a new profile is created, which in turn will read the modified campaign file. Therefore, when playing a mod, a user must create a new profile for this mod, otherwise the campaign information will be that of the campaign file in use when the profile was created.

commanderai/

commanderai/constants.juice

Global constants for the Commander level AI.

commanderai/*.juice

These are "brain" files for the Commander level AI, and conceptually represent the different AI "personalities".

guis/

guis/buyableagents.juice

This file is used by the ingame reinforcements gui screen, and controls how the different unit categories are displayed on the screen.

Note about **AgentClassID**:
The structures in this file includes lists of the user defined **TYPE AgentClassID**. This **TYPE** is defined in **juice/agentclassid.fruit**. The ids in this **TYPE** correspond directly to the names of the unit types in **units/unittypes.juice**. If you add units to **units/unittypes.juice**, you must add new ids to **AgentClassID** that exactly match the name of your new unit types in order to make these units selectable in the game.

guis/gui_frontend.juice

The "frontend" gui screens.

Note about gui scripts:
The various gui scripts share a common file structure, and can be edited in an integrated gui editor built into JuiceMaker. When editing a gui script (for example **guis/gui_frontend.juice**), expand the structure until you see the contents of myGuis. By clicking on any of the guiscreens listed here, the gui editor will launch in the right hand view.

When editing guis, note that a great many of the widgets that exist are directly accessed by code (via name and type). Some widgets are however only eye candy. Unfortunately, a complete list of exactly which widgets are required by the game code would be too exhaustive to list here. We recommend testing often if you feel you need to delete widgets so that the game does not break (which it will do when the corresponding gui screen is activated by code).

guis/gui_ingame.juice

The "ingame" gui screens.

guis/gui_ingame_slim.juice

The slim "ingame" gui screens.

guis/gui_mos.juice

The MassGate (Massive Online Services) gui screens.

guis/gui_multiplayer.juice

The multiplayer specific gui screens.

guis/guiskins.juice

This file defines the graphical "skin" used by the user interface system.

Originally, we had planned to support multiple skins, but in the end only used a single one. Hence the structure of the file. The skin is best edited using the integrated editor in JuiceMaker (invoked by clicking on mySkinFile/mySkins/skinOne).

Under skinOne you will find:

myFileName - this is the texture file used by the skin. Note that this is a .tga file, which is used by the JuiceMaker editor, but GCII.exe replaces the .tga extension with .dds if such a file exists (.dds is the preferred compressed format for textures).

myPrefabs - these are the actual "widget appearances" that can be viewed in the game, for example the different button styles you see. You will often see references to these prefabs in other .juice files (my***Prefab).

myFonts - different high level font sizes and colors used in the game

myFontData - localization data for fonts, you probably don't want to mess with this.
myFontRanges - localization data for fonts, you probably don't want to mess with this.

maps//*.juice*

These directories include source and export data used by Xed. We strongly recommend using Xed directly to edit these files.

The root file is read by GCII.exe for each map is **maps/*/export.ice**, which in turn references a number of other files. Check out **maps/*/export.juice** to see these references.

massgatetext.juice

Miscellaneous localized strings used by the MassGate subsystem.

messagebox_characters/characters.juice

This file defines the 3d characters seen in messageboxes in the game.

*movies/*_subtitle.juice*

Subtitle scripts for the various cutscenes in the game.

Sound/

agentsounds.juice

This file contains hierarchies of sound groups and sounds. These files are identified in other files (for example **units/unittypes.juice**) by name.

The easiest way to edit these sounds is to use the integrated JuiceMaker editor. Expand the hierarchies and click on the names until you find a sound instance (the sound editor will launch in the right view).

Each sound consists of a number of layered .wav files with individual falloff envelopes (volume / distance), defined in mySpecs. It is possible to use any number of .wav file layers, but we have only used two (one for "near" sounds, one for "far" sounds). This approach was initially designed to simulate the attenuation of high frequencies over distance without requiring hardware support for filtering, but since there are two unique .wav files involved, more creative things can be done as well.

myNumChannels is to my knowledge not used anymore. I am unsure if myBaseFrequency is used, but if it is, it allows for frequency shifting of the .wav files. 0 means to use the default frequency of the .wav file (we used 22 khz).

sound/ambientsounds.juice

This file contains the same kinds of sound hierarchies as **sound/agentsounds.juice**, but is referenced by Xed scripts instead of by unit types.

ui/default_settings.juice

This file defines default options settings for new profiles.

units/

units/dropship.juice

This file defines dropship configurations for the various factions. Note that we are at this time, due to implementation details concerning the dropships, limited to the three factions that are found in Ground Control II, namely NSA, Viron, and Terran Empire.

units/unittypes.juice

This file defines most everything that has to do with the individual units in the game.

There are two main data chunks in this file, DamageModel and UnitTypes.

Damage Model:

DamageModel controls how units interact with each other (damage each other)

myHighGroundAngle - the angle at which high ground bonus is active. For example, if unit A is firing at unit B from an upward angle of at least 30 degrees, unit B will have high ground bonus.

myHighGroundAvoidDamageChance - if a unit has high ground bonus for a projectile collision, this is the chance (in percent) that he will avoid all damage from this collision

myBuildingDamageMultiplier - if a unit is inside a building, all damage inflicted upon this unit is multiplied by this factor (for example a factor of 0.6 will reduce damage to units inside buildings)

myForestDamageMultiplier - if a unit is in a forest, all damage inflicted upon this unit is multiplied by this factor (for example a factor of 0.75 will reduce damage to units inside forests)

myTable - this is the main DamageType vs. ArmorType table, which defines multipliers for damage inflicted by a projectile of DamageType vs. a unit that has ArmorType. The actual hit point damage that the projectile has will be multiplied by this factor.

UnitTypes is the list of all unit types in the game. These unit types are referenced from many places by their name (for example F1_Marine), and these names must therefore be globally unique. Each individual unit has the following data fields:

myFactionId - the only valid values here are NSA, Viron, or Terran Empire. We are currently limited to these values.

myType -

myRoleType -

myRadius - the metric radius of the unit, used for spherical collision detection

myHealth - hit points

myDisableWhenKilledFlag - 0 or 1, used to specify that a unit should be disable when hit points <= 0 instead of dying. We only used this for turrets that are intended to be repairable.

myUIName - localized "user interface" name of the unit type

myInfoText - localized "info" text

myCost - AP cost of the unit type

my*Feedback - generalized audio feedback for positive, negative, and damage events

myDeathEffects - for each damage type, a script of the animated death effects for the unit type. DAMAGE_MELD is special, and is only used for units that are melded / unmelded, as well as the actual meld / unmeld eggs.

myDropshipOccupation - the number of "slots" the unit type will take in the dropship

myMaintenance - the maintenance cost of the unit type

myPresence -

myUnitCategory -

myPrimaryMode / mySecondaryMode

myUIIconFile - the image displayed in the gui (for example when the unit is selected)

mySwitchButtonIconFile - the image displayed in the order palette slot for toggle modes (primary / secondary)

myRep - script for the visual appearance of the unit in each mode.

myModelName - reference to an .mrb file that is the visual representation of the unit (the unit model)

myShadowFile - reference to an .sdw shadow file that is used to define how the unit shadows appear

myTracksTexture - the texture file used to represent unit tracks on the ground

myTracksWidth - metric width of the tracks

myLightType - controls how the unit is lit in the game

myColoredParticleEmitters - a list of the particle emitters in the unit .mrb representation that should take their color from the minimap

myParasites - see below (Unit Parasites)

Unit Parasites:

These lists are the core of unit behaviour and functionality for the respective modes. Unit behaviour is implemented using a system of "parasites", meaning that each functionality itself is a parasite that is attached to the unit, whereby the unit gains that particular functionality. This system allows for very flexible unit construction with a minimum of programmer involvement.

Units always have a primary mode, but the existence of a secondary mode is controlled by the existence of parasites in the mySecondaryMode list. If this list is empty, the unit type will not have a secondary mode. Note that all order palette icons are controlled directly by the presence of certain parasites.

Here is a list that shows the available orders and the Parasites that generate them.

Orders: **Move**, **Stop**, **Follow**
Generated by:
TankMover, TankNonMover, DropshipMover, HoverMover, HoverNonMover, VehicleMover, VehicleNonMover, WalkerMover, InfantryMover, InfantryNonMover, CopterMover

Orders: **Attack**, **Stop**
Generated by:
NonShooter, StraightShooter, BallisticShooter, HomingShooter

Orders: **GoToBase**, **GoToLandingZone**, **ToggleReturnToBase**, **SelectLandingZone**
Generated by:
Dropship

Orders: **Load**, **UnloadAll**
Generated by:
Container

Orders: **Meld**
Generated by:
Meldable

Orders: **Unmeld**
Generated by:
Unmeldable

Orders: **Blow**
Generated by:
Blower

Orders: **Repair**
Generated by:
Repairer

Orders: **EnterBuilding**
Generated by:
Resident

Orders: Empty order slot
Generated by:
EMPTY_ORDER_SLOT

Available parasite types are:

AntiMissile

Allows a unit to shoot down incoming enemy Homing and Ballistic projectiles.

Armor

Armor specification for the unit type. Units always have armor, and if no armor parasite exists, the unit will have armor type ARMOR_NONE and zero directional reduction on all sides.

AutoCloudSpawner

Causes the unit to automatically spawn Clouds at certain intervals.

BallisticShooter

Causes the unit to fire ballistic projectiles (i.e. artillery). Only a single shooter per mode is allowed. If a shooter is not included, the unit cannot fire.

Blower

Adds an order which the user can destroy the unit manually (self-destruct). Was not used in Ground Control II.

BlowerTrigger

Causes the unit to detonate other units, which have a Blower parasite within a certain range. Was not used in Ground Control II.

Containable

Allows the unit to be contained within another unit (i.e. APC passenger, infantry, barrels). The numeric type must match that of the Container in question for the Container to be able to container the Containable.

Container

Allows the unit to contain (carry) other units that have a Containable parasite of the corresponding type (i.e. APC).

CopterMover

Makes the unit move like a copter / helidyne. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

Dropship

Makes the unit a dropship. Must be accompanied by a DropshipSpawner.

DropshipMover

Makes the unit move like a dropship. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

DropshipSpawner

Makes the unit able to spawn units (i.e. the dropship).

EMPTY_ORDER_SLOT

This is a little cheat used to allow for designer manipulation of how the orders appear in a unit type's order palette. Since each parasite automatically generates orders in the order they appear in the list, this "fake parasite" allows for empty slots in the palette. This is useful for example in the case when a unit type has no shooter in secondary mode, but we don't want the order of the other order icons to change (we would thus make the first slot empty). This does not add any unit functionality at all.

EngineSoundSource

Makes the unit emit "engine" sounds.

HomingShooter

Causes the unit to fire homing projectiles. Only a single shooter per mode is allowed. If a shooter is not included, the unit cannot fire.

HoverMover

Makes the unit move like a hoverdyne. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

HoverNonMover

This is special animation functionality that makes the unit hover properly when immobile. This should be used for units that have no normal mover (i.e. should not be able to move) in order to make them animate correctly (animation is normally completely controlled by a mover).

InfantryMover

Makes the unit move like infantry. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

InfantryNonMover

This is special animation functionality that makes the unit stand properly when immobile. This should be used for units that have no normal mover (i.e. should not be able to move) in order to make them animate correctly (animation is normally completely controlled by a mover).

Meldable

Allows the unit to meld with another unit. Use this in conjunction with the meldtable to make units meldable.

Melder

Allows the unit to die and produce two new units after a certain time (i.e. the meld egg). Use this for units that are defined in the meldtable as myMeldEggUnit.

NonShooter

This is special animation functionality that makes the unit aim properly (i.e. aim forward) when not shooting. This should be used for units that have no normal shooter (i.e. should not be able to shoot) in order to make them animate correctly (animation is normally completely controlled by a shooter).

Regenerator

Allows the unit to automatically regain health over time.

Repairer

Allows the unit to heal / repair other units.

Resident

Allows the unit to enter buildings. Note that this is not limited to infantry.

Seer

Allows the unit to "see" enemy units and contribute to Team Line Of Sight (LOS).

Note about Team LOS:
All units that "see" (have Seer parasites) contribute what they see (via range and line of sight vs. terrain calculations) to a "pool" that is shared by all units on a given team. This means that a player will see every unit that any of his own or allied units can see.

Stealth

Stealth effectively reduces the view range of an enemy unit that is looking at this unit.

StraightShooter

Causes the unit to fire "straight" projectiles. Only a single shooter per mode is allowed. If a shooter is not included, the unit cannot fire.

TankMover

Makes the unit move like a tank. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

TankNonMover

This is special animation functionality that makes the unit stand properly when immobile. This should be used for units that have no normal mover (i.e. should not be able to move) in order to make them animate correctly (animation is normally completely controlled by a mover).

TargetAcquirer

Causes the unit to automatically acquire targets when not idle (i.e. no current target). A shooter must accompany this.

Unmeldable

Allows the unit to unmeld into source units. Use this in conjunction with the meldtable to make units unmeldable.

Unmelder

Allows the unit to die and produce one new unit after a certain time (i.e. the unmeld_egg). Use this for units that are defined in the meldtable as myUnmeldEggUnit.

WalkerMover

Makes the unit move like a walker. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

VehicleMover

Makes the unit move like a wheeled vehicle. Only one mover per mode is allowed. If a mover is not included, the unit cannot move.

VehicleNonMover

This is special animation functionality that makes the unit stand properly when immobile. This should be used for units that have no normal mover (i.e. should not be able to move) in order to make them animate correctly (animation is normally completely controlled by a mover).

ZoneCapturer

Allows the unit to capture zones. For example, only ground units had a ZoneCapturer in Ground Control II.

units/unittypes_advanced.juice

xed.juice

This is the XEd configuration script. Some things that you may find interesting are:

myXXXXTool.myTool.myCameraMoveSpeed
Defines the default speed of the camera for the tool.

myXXXXTool.myTool.myCameraHeightSpeed
Defines the speed of which the camera changes height.

myXXXXTool.myTool.myCameraRotateSpeed
Defines the speed of which the camera changes rotates.

myXXXXTool.myTool.mySlowDownRate
Multiplier of the speed value when the ALT key is pressed.

myWorldToolsState.myBrushes
Any new world tool brushes needs to be defined here.

myWorldToolsState.myDefaultBrush
Defines the index in the list above that will be used as default.

myTextureToolsState.myBrushes
Any new texture tool brushes needs to be defined here.

myTextureToolsState.myDefaultBrush
Defines the index in the list above that will be used as default.

myScriptState.myCommands.XXXXX_xxxxx
Defines default values for all command parameters available in the script dialog.

myDefaultWorld.myMissionStats
Defines default values for the missionstats.

myLandscapes
Any new landscape needs to be added to this list to be selectable in XEd.

Note about landscape script files:
A landscape script defines everything associated to what we call "a landscape" type in GCII. For example, the desert setting is a landscape, and the ruined city setting (battleground) is a landscape.

A landscape script includes the following data chunks:
myTextures - this is the "splatting" script, which controls how textures are automatically applied to the mission terrain.
myEnvironments – This chunk defines the different light and sky presets that are available for a certain landscape.
myPropTypes – Defines all available props for the landscape.

xed/battleground.juice

A landscape script file.

xed/desert.juice

A landscape script file.

xed/farmland.juice

A landscape script file.

xed/islands.juice

A landscape script file.

xed/orange.juice

A landscape script file.

xed/exodus_database.juice

This file defines the database that XEd uses. This is only a compilation of other data files.

myDatabase.myLandscapes
Any new landscape needs to be added to this list for XEd to find their data.

xed/fhmgpresets.juice

This is where the presets for the FractalHeightMapGenerator are stored. You would normally not edit this file outside XEd.

xed/globalproptypes.juice

All the “multi-landscape” or global proptypes should be placed here.

xed/meldtable.juice

This file globally defines how units meld / unmeld. This information is exported in each mission export.

The names of the relations in this list are of no consequence, and exist simple for designer reference.

myFirstUnit - the first unit type of a relation
mySecondUnit - the second unit type of the relation
myMeldEggUnit - the unit type used to represent the melding process
myUnmeldEggUnit - the unit type used to represent the unmelding process
myResultingUnit - the unit type that is the result of a meld between myFirstUnit and mySecondUnit
myPregnancyLength - the time it takes to meld myFirstUnit and mySecondUnit into myResultingUnit
myUnmeldLength - the time it takes to unmeld myResultingUnit into myFirstUnit and mySecondUnit

Note that unmelding is the reverse of melding, so myResultingUnit will become myFirstUnit and mySecondUnit when unmelding. Note that even though we have only allowed the melding of a certain unit type with itself, the system will allow any combination of source and output units, including meld and unmeld eggs.

xed/supportthings.juice

This file defines all support weapons for all factions. This information is exported in each mission export.

myGuiName
The localized name that will be presented to the user.

myFaction
The faction ID that this supportweapon belongs to.

myAPCost
The cost in AP to use this weapon.

myRechargeTime
The time in seconds before the weapon can be used.

myTimeBeforeActivation
The time in seconds from the issue of the command until anything happens (traveltime).

myDeviationRadius
Radius in meters of the scatter area.

myRateOfFire
The number of seconds between the individual shells in a barrage.

myNumberOfProjectiles
The number of individual shells in a barrage.

myProjectileType
STRAIGHT or BALLISTIC

myMarkerModel
Not used.

myProjectile

mySpeed
The speed of the projectiles

myStartHeight
The starting height of the barrage.

myGuiIconFileUp/Inside/Down
Gui icons for the weapon

xed/templates.juice

This is where the templates for the scripts are stored. You would normally not edit this file outside XEd.

xed/weathers.juice

This is where weathers are defined.

xedrenderfiles.juice

This file defines the global renderer resources used by XEd. .

juicemaker.juice

This file includes configuration information for JuiceMaker.