Xconq

The Penultimate Strategy Game

Version 7.0

March 1995

Stanley T. Shebs

1 Xconq, the Penultimate Strategy Game

Welcome to *Xconq*!

Xconq is a powerful multi-player game system. With *Xconq* you can build empires, fight wars, relive history, and adventure across fantastic worlds. You can play computerized board games, or be Godzilla rampaging through Tokyo.

Xconq runs on many computer systems. It features several different user interfaces, including ones based on Unix(tm) terminal graphics ("curses"), the X Window System, and the Apple Macintosh(tm). Although all share the same basic design, the "look and feel" of each interface matches the system you're on. The details of a particular *Xconq* game depend heavily on its *game design*, which defines the pieces in the game and its rules of play. So, to understand everything about a game you're playing, you need three sources of information: this manual, which covers all general information; documentation for the user interface; and documentation for the game design in effect. Although this sounds like a lot, most games don't use every feature of *Xconq*, and the user interfaces will usually offer plenty of online guidance.

1.1 About This Manual

The remainder of this chapter discusses compatibility with older versions of *Xconq*, and additional information resources.

Chapter 2, "Playing Xconq" is the player's guide. It describes the general concepts shared by all the *Xconq* games.

Chapter 3, "Designing Games with Xconq" is a designer's guide for building and modifying *Xconq* game designs, using both Game Design Language (GDL) and online editing tools.

Chapter 4, "Reference Manual" is the complete definition of GDL. It includes the syntax and semantics of all parts of the language.

Chapter 5, "Hacking Xconq" describes the general architecture of *Xconq*, and how to extend the basic program. This chapter also includes a rationale for the major design decisions, and some ideas for future development.

1.2 Compatibility

Xconq version 7 is not directly compatible with previous versions, although most of the basic game concepts remain unchanged. If you've played *Xconq* before, you should have little trouble getting used to the differences.

If you've designed any games (periods, maps, or scenarios) for version 5, you will discover many changes. Version 5 of *Xconq* used a mix of crude fixed-format syntax and a simple postfix language for game designs (which were then called "periods"). This version has changed too radically to be able to read any of the old period, map, or scenario files. For instance, version 7 eliminates the distinctions between "period", "map", and "scenario". Therefore, if you have old *Xconq* files, you should invest the time to convert. You may even discover that some of the new features of *Xconq* provide a better solution to your design problems. (The shell scripts `per2game`, `map2game`, and `scn2game` will help you get started on conversion; you can find them in the `misc` directory.)

1.3 Where to Get Game Designs

Xconq is not useful without game designs. The library distributed with *Xconq* includes many designs, some inherited from previous releases and others that are entirely new.

You are encouraged both to modify any of the existing game designs and to develop your own. Chapter 3, "Designing Games with Xconq", includes a comprehensive tutorial on how to do this. The version 7 game design language (GDL) is better-designed and more robust than the machinery in version 5, so if you've been discouraged by mysterious problems before, you might want to try designing with version 7 instead.

1.4 For More Information

The ftp server `ftp.cygnus.com` usually has the latest version of *Xconq*, as well as other contributed material, in the directory `pub/xconq`. Other servers also have copies of *Xconq*. See your local Internet wizard, or buy one of the many excellent Internet guides if this isn't enough information for you.

1.5 Acknowledgments

Since the first release of *Xconq* in 1987, it has benefited from the work and ideas of literally hundreds of people, first at the University of Utah, then worldwide.

Special thanks must go to Eric Muehle, a tireless source of ideas, advice, and playtesting at Utah; Greg Fisher, who added many good things to make 5.4; and Robert Forsman, who did a great deal of work for 5.5. Eric Ziegast and Alan Clegg have been essential to maintaining the *Xconq* mailing list and archives.

Massimo Campostrini contributed the printing code, xshowimf, and a number of library modules, as well as many other fixes and enhancements.

Other contributors have been (in alphabetical order): Jim Anderson, Ed Boston, Mark Bradakis, Alain Brossard, Richard Buonanno, Germano Caronni, Harold Carr, Ben Chase, Chris Christensen, Kevin Deford, Dan Dickey, Fred Douglis, Miles Duke, Barry Eynon, David Harr, Scott Herod, Eiji("A.J.") Hirai, Kurt Hoyt, Jeff Kelley, Bob Kessler, Jed Krohnfeldt, Rick Ledoux, Brian Lewis, Sandra Loosemore, Michael Lounsbery, Steve McInerney, Eric Mehlhaff, Jimmy Miklavcic, Tim Moore, Scott Mueller, Julian Onions, Dave Pare, Stephen Peters, Chris Peterson, Mohammad Pourheidari, Dan Reading, Tom Richards, Joel Rives, Jay Scott, John Shovic, Josh Siegel, Leigh Stoller, Ravi Subrahmanyam, Cimarron Taylor, Spencer Thomas, John Tonry, Rich van Gaasbeeck, Henry Ware, Grant Weiler, Jeff Young, and many others.

Thanks also to the University of Utah, Apple Computer, Inc., and Cygnus Support, who have all contributed machine resources that helped in the development of *Xconq*.

2 Playing Xconq

This chapter is about how to play *Xconq*. Although *Xconq* supports a wide variety of games, and runs on many different computer systems, they all have much in common, and it is these common features that will be described here. This chapter, along with the document for your system and the document for the game you're playing, should provide all the information you need to play and enjoy *Xconq*.

The term *interface* refers to the particular graphical user interface in use. Examples include X11, curses, and Macintosh. Interfaces can vary radically from each other, since each is designed to be best suited for its environment. In practice though, interfaces will tend to share the same commands, so that you don't learn to learn a whole new set when switching computers.

When reading this chapter, you should be aware that the term *game* is more precisely *game design*, since it is the set of rules and definitions of the game you want to play. Since *Xconq* allows for many different kinds of games designs, much of the information in this chapter will be irrelevant to a particular game. This will be indicated by phrases like "some games" or by saying that a game "may" implement some concept or behavior. You should learn what the game you're playing actually does in these cases. The names of the variables or tables to look at will often be mentioned in `computer type`.

2.1 Setting Up A Game

To get started with *Xconq*, you have to select which game you want to play. The possibilities may be presented to you, or you may have to look in some sort of library to see what's available and then supply that name on a command line. If you don't do anything, then you will get a default game.

Some games require no additional setup; once loaded, you're ready to go. Others will require additional decisions, such as the size and shape of the playing area, whether exploration will be necessary, or whether the game is realtime. These will all be called *variants*. The exact set of variants is determined by the game design, and the interface will (usually) tell you about them.

In addition, most games also give you a choice of which player is to play which side in a game, as well how many players can join in. There are two kinds of players: humans, who have displays, and *artificial intelligences* or *AI*s for short, which are run by the computer. Some versions of *Xconq* may include more than one kind of AI; each type has a distinct name. The AI named `mplayer` is always available.

An example might be a simulation of Europe *ca* 1900, named `"la-belle-epoque"`, in which the sides might be "England", "France", "Germany", and "Austria-Hungary", and the players might be Joe on a Sun-4, Natalie on a Mac, and two of the `mplayer` AIs. You can set Natalie to play England, Joe to play Germany, and the two AIs to play France and Austria-Hungary.

Some game designs provide a way to even things up if the players are of vastly differing abilities. In these designs, each player has an *advantage* that affects how much he or she gets to start with. Weaker players should get a higher advantage, so for instance a game with two players, of advantages 1 and 4, might give the advantage=4 player 8 cities while the advantage=1 player gets only 2. This affects setup only; during the game all players are equal. The variability of advantage also depends on the game; some may allows differences of 10 to 1 or more, while others, especially historically accurate scenarios, will have a fixed advantage that the designer has set for each side.

Once a trial player setup has been made, *Xconq* runs "synthesis methods". These methods are specified by the game design, and randomly generate anything that was not explicitly spelled out; for instance, the initial location of countries, terrain features, and so forth. As a player, you don't have to concern yourself about synthesis methods, but you should be aware that you may sometimes run into situations were a synthesis method simply cannot cope, and your game setup will fail. A common case is where you ask for many players to be set up in a small world, and the set of constraints is too "tight" for an initial setup to succeed. In such cases, you just have to try different setups and maybe complain to the game designer. Synthesis methods may also take a long time to run; for large worlds and lots of players, be prepared to wait.

When initialization and setup succeeds, *Xconq* will try to open up displays for every player that wanted one. Exactly how this happens depends on the interface and networking capabilities of the version of *Xconq* you're using. Once this is done, *Xconq* will start the game for real.

You may also get a warning that "images were not found". This happens when the game design specifies the use of particular icons or patterns (collectively call *images* here), but they cannot be found anywhere by *Xconq*. This is not fatal, because *Xconq* will use generic default images instead, but the display may be hard to understand. There are several possible reasons for images not to be found: 1) the game designer might have specified the use of particular images, but never defined them, 2) the library of images was not updated to include the needed images, or 3) the image library is not located where Xconq is looking.

2.2 Starting Play

What you'll first see depends entirely on the interface you're using. Typically there will be a map and a list of sides, possibly other displays as well. Help is available in the "usual" ways, and the interface is robust, so you can always just try to find your way around by experimentation. (This is best done by yourself, rather than in a game with a lot of other people.)

The game proceeds as a sequence of *turns*. During each turn, you and the other players get to move your *units*, which can be anything from cities to submarines to insects, depending on the game. In addition, there may be *backdrop activities*, such as changing seasons and weather, that go on all by themselves. These typically happen at the beginning or end of a turn, not while players are moving their units.

Your exact goal in the game depends on the *scorekeepers*. Most games have at least one, some have several, and some have none. There are many kinds of scorekeepers, so be sure you know and understand what they are before getting too far into a game! If there are no scorekeepers at all, you can do whatever you like; any AIs playing in such a game will behave quite randomly.

A game may last anywhere from a few turns to many hundreds. Again, this may be limited by the game design, or perhaps by the nature of the game. For instance, a game of oil empires might be forced to end when the world's oil supplies are exhausted...

2.3 Worlds and Areas

> Gallia est omnis divisa in partes tres [All Gaul is divided into three parts] – JULIUS CAESAR

The *Xconq* "world" is always a sphere. However, you only play on a piece of its surface, which is called an *area*. Currently, there can only be one world and one area in a game; this may change in a future version of *Xconq*.

An area is divided into a grid pattern of *cells*. Although squares with four or eight neighbors could be used (and were, in the very first version of *Xconq*), currently only a hexagon grid is available. Each cell is therefore adjacent to six others, in the directions NW, NE, W, E, SW, and SE. Areas have a *width* and *height* that are the number of cells across and up/down. Although you can ask for areas down to 10x10 or less, or up to 1000x1000 or even more, the ideal default is typically around 60x30. Larger areas consume vast quantities of memory, plus they're slow and unwieldy to play on.

If the area's width matches the circumference of the world, it is a cylinder in shape. The cylinder can be circumnavigated in an east-west direction. This is what an 8x6 cylinder area might look like (periods are sea, `+` and `^` are land, `#` indicates edge cells):

```
 # # # # # # # #
  . . + + . . . .
 . . . + ^ . . .
  . . . . . . . .
 . . . . ^ . . .
  # # # # # # # #
```

Areas whose width is less than the world's circumference have a hexagonal shape. This is an 8x7 hexagon:

```
    # # # # #
   # . + + . #
  # . . + ^ . #
 # . . + ^ . . #
  # . . . . . #
   # . . ^ . #
    # # # # #
```

The top and bottom rows of the cylinder shape, and all the sides of the hexagon shape, although they are displayed, may not be entered (except when leaving the world entirely, which some games allow). These cells are called *edge* cells.

The types of terrain you'll find in the world depends on the game design; typically there will be sea, land, mountains, swamp, and so forth, but more exotic games have been known to feature junkheaps, lava, and black holes as "terrain".

Terrain can cover an entire cell, be linear features passing through or between cells, or be a coating overlaying other terrain. *Cell terrain* covers the entire cell uniformly, right out to its edges.

A *border* is the boundary between two adjacent cells; it has a distinct terrain type, such as "river" or "beach". A *connection* is a narrow ribbon of terrain that reaches from the middle of one cell to the middle of an adjacent cell. Like borders, connections are distinct types, for instance "road", "railway", or "canal". Connections take precedence over borders and underlying cell terrain; in other words, if cell or border terrain is impassable, but there is a passable connection type, then the connection allows passage. Thus a connection can be usable as a bridge. You may also find more than one type of connection or border, between two cells, such as both a road and a rail line.

A *coating* is like snow; it is a type that co-exists with cell terrain. Coatings can change from turn to turn, varying in depth.

Note that any single terrain type can only play one of these roles. This means you will never have river terrain that is both border and connection, nor will snow be both a coating and a cell type.

In some games, each cell has an *elevation*, which is basically elevation above sea level, but could be any range of values, as set by the game design. The game design also defines the effect of elevation on movement, visibility, weather, and so forth.

A world can have named *geographical features* or just *features*, such as a bay, mountain, desert, or valley. Geographical features never have any direct effect on your game, but some interfaces may label features when drawing a map, or use them to help describe locations verbally, in phrases like `"1 hex NW of Broken Hill"`.

A world can have *people* living in some or all of its cells. People belonging to a side report everything they see in their cell to their side. Some types of units will change the people's side to the unit's, if that unit is of the proper type, such as an occupying army.

2.4 Units

Units can be almost anything: adventurers, armies, balloons, bicycles, dragons, triremes, spiders, battleships, bridges, headquarters, cities. Units move around, manufacture things, fight with other units, and possibly die. They are the playing pieces of *Xconq*.

Units have a location, either in out in the open terrain of a cell, or inside some other unit. In games that define connections, a unit may be on the connection rather than on the predominant terrain of the cell. (Think of a truck on a bridge.) There may be more than one unit in the open in a given cell, up to a game-defined limit. The collection of units sharing a cell is called a *stack*. A unit inside another unit will be called an *occupant* in a *transport*, even if the "transport" is a type that can never move.

A unit's location may also include an *altitude*, expressed as its distance above the surface of the cell it is in.

A unit either belongs to a side, or else it is considered *independent*. Independent units do not do very much. In more complex games, the unit's side merely represents the current ownership, and the unit may have a range of feelings towards each side, including its current one. In those games, it is possible for one of your units to be a traitor!

Units can have a name, full name, a title, and a number, as appropriate to the situation. The name is an ordinary name like "Joe Schmoe" or "Cincinnati", while the full name might be something like "Joseph P. Schmoe". The title is a form of address such as "Lord". The unit number, if used, is an ordinal that is maintained for each side and each unit type, so you can have both a "1st national bank" and a "45th infantry division" on your side. Names and numbers are always optional, and can usually be changed at any time during the game.

Every unit starts out with a number of *hit points* or *hp* representing how much damage it can sustain before dying. Certain types of units, such as armies and fleet of ships, have multiple *parts*, which means that damage to them reduces their effective size. Multi-part units can merge with and detach from each other. Damaged units may recover their hp on their own, or else be repairable by explicit action, either by themselves or by another units (ships in port for example).

In addition to occupants, a unit can also carry *supplies* (food, fuel, treasure, etc), which are type of materials (see the next section). Supplies are used up by movement, combat, and by just existing, and are gotten either by producing them or by transferring them from some other unit. Some games start units out with lots of supplies, while in others you have to acquire them on your own.

What a unit can do at any one time depends on the *action points* or *acp* available to it. Each sort of action - movement, construction, repair, etc - uses up at least one action point, and possibly more. A unit with an acp of 0 can never do anything on its own, although other units can still manipulate it. Also, not every type of unit can do every type of action; this is also defined by the game design. Section xxx lists all the types of actions that are possible in *Xconq*.

[explain cxp, when it's implemented]

You can lose a unit in many different ways: in combat, by running out of essential supplies, by being captured, by revolt, by garrisoning a captured unit, by leaving the world, or in accidents.

2.5 Materials

In *Xconq*, *materials* are basically bulk inanimate stuffs, like food or fuel. They are kept in units or in cells, up to limits defined by the game. Materials may be provided as part of the initial game setup, or else produced by units and cells. They are consumed by construction, movement, or merely in order to survive. You can also move materials around from unit to unit. Some games define laws of supply and demand, which will move materials for you, though not necessarily in the directions you would prefer!

In a few games, possession of a material type may figure into your score (your gold in a medieval game, for instance). In other games, there are no types of materials at all.

2.6 Sides

Each player in *Xconq* runs a *side*. The concept of "side" is somewhat abstract in *Xconq*; units in a game belong to sides, but the sides themselves are not attached to any particular unit. Side often represent countries, but not invariably.

It is important to be clear about sides and players. A side is a part of the simulated world, while a player is the actual real-world person or program that is playing the side. You yourself are always the player, but in one game you may play the German side, and in another the Klingon side. During a game, there will always be a player for each side, and vice versa. The distinction is most important during setup, since you can swap players between sides.

Each side can have a name and associated parts of speech, such as a noun for individuals on the side and an adjective to describe anything belonging to the side. [example?] Sides can also have emblems and colors that are used in displays. Some game designs preset all this, while others let you personalize as desired. See the *Xconq* document for your system to learn how to do this.

2.6.1 Interaction Between Sides

In games with two players, your interaction is usually pretty simple, i.e. bash on each other. In games with many players, some human, some mechanical, it is possible to have a variety of relationships, ranging from complete trust to complete hatred.

One thing you can do is to make your side be controlled by another side. This is basically surrendering, because the controlling side can manipulate any of your units as if they were its own.

The controlling side also has the option of allowing or forbidding you to move your own units. The relationship is strictly one-sided, and only the controlling side can release the controlled side. (Note that this is a way to have several people play on a side; have one player run the controlling side and be helped by several other players running controlled sides, usually with agreed-upon responsibilities.)

A less extreme, but still very close, relationship is trust. This is like a close ally - you can enter each other's transports, you share view data, and so forth. Trust is a two-way relationship; both you and the other side each have to declare you want to trust the other. You can do this at any time. You can also, unilaterally, withdraw your trust in another side at any time.

2.6.2 Agreements

> Diplomacy is to do and say // The nastiest thing in the nicest way. – ISAAC GOLDBERG (1938)

If you don't want to declare a special relationship with another side, but still want to make some sort of adhoc arrangement, you can create an *agreement*. An agreement is a sort of generalized treaty; it consists of a number of *terms* agreed to by a number of *signers*, which are sides. Agreements may be public or secret, and you can declare them to be enforced by *Xconq* if the terms are in a form it understands. An agreement that just says "help each other out" cannot be evaluated by the computer!

To make an agreement, you tell the interface to create one, fill in its terms, possibly give it a name, make up a list of proposed signers, then either propose it directly or else send to *drafters*, which are the side you want to help with the composition of the agreement. The draft also includes the list of sides that will know about the agreement.

When the agreement is officially proposed, it will be displayed to all sides that are to sign, and represented as coming from the sides listed as *proposers*. *Xconq* will then ask each proposed signer to sign; if all do so, then the agreement goes into effect immediately. All sides that are to know about the agreement will be informed of its terms.

Some interfaces may allow players to copy and modify a proposed and circulate it along with the original. The proposing side may also withdraw a proposal, but cannot modify it without having it signed again by everybody involved.

Once in effect, an agreement cannot be modified, and it cannot be removed unless it includes a term that provides for this.

An agreement can have any number of terms. Each term can have one of several forms:

A text string. This is not interpreted in any way and could be a comment, preamble, or whatever.

A true/false expression. This must always be true for the agreement to be valid.

A statement of an action. This action will be performed at the instant that the agreement goes into effect.

An if-then statement. If the condition is true while the agreement is in effect, then the action will be performed.

[need some examples]

Note that the drafter/proposer/signer distinction has many uses; for instance, you can draft an agreement to be proposed by a coalition of sides, but the proposed signers are neutral sides that you want to keep quiet.

2.6.3 Trade

You can specify the nature of the trading relationship with other sides. The basic theory is that traders are businessfolk and don't care much about politics; they will do business with anybody. However, a player can define relationships with other sides via tariffs. A tariff is a per-side per-material percentage that will be taken from any transfer from/to units on one side to units on another side. You can define both import and export tariffs. A tariff of zero means free trade, and negative tariffs are allowed; in such cases your stock of material is used to add to the transfer.

2.6.4 Tech Levels

In some game designs, technology and research are important. These games give each side a set of *tech levels* (or just *tech* for short), one for each type of unit. The tech level represents the technological knowledge needed to see, operate and build a type of unit. Tech levels never decrease (they may in real life, but only over very long time intervals), and they can be increased by research and espionage.

There are several tech thresholds for a unit. First there is `tech-to-see`, below which you will not even be aware of the existence of a unit (consider barbarians unable to see spy satellites passing overhead). Then there is `tech-to-use`, which you must have in order to make the unit do any actions. The `tech-to-understand` and `tech-on-acquisition` are points at which your side can increase its tech level just by owning a unit, and finally the `tech-to-build` is what you must have to create new units of the given type.

See below to find out how you can do research and espionage to increase your tech level.

2.6.5 Side Classes

In some games, several sides may be very similar, while being very different from other sides in the same game. These similar sides can be given the same *side class*. Units may then be restricted to be usable only by the sides in a particular class. (Note that this is different from tech level, which allows units to be used by any side that has managed to acquire a sufficient tech level.)

2.6.6 Self-Units

A *self-unit* is one that represents your whole side in some way. For instance, in a dungeon exploration game, your "side" might consist of an adventurer (you), your possessions, your followers, and perhaps more. In such a case, if the adventurer dies or is captured, then the game should be over, at least for you.

Usually the self-unit will be set up by the game design, and all you have to do is to be aware that losing the self-unit ends your participation instantly. Some games might have "self-unit resurrection" which just means that if another unit is available when the self-unit dies, then that another unit becomes your new self-unit. This is like where admirals would leave their sinking flagship and board another ship, thus "transferring the flag". (Admirals presumably being more valuable than captains, who're supposed go down with their ships!) Some games may also allow you to change self-units manually.

2.7 Moving the Units

Once the first turn begins, you can begin looking at the display and moving your units. Depending on the game design and startup options, you may or may not be moving simultaneously with the other players. If not, then the players move one at a time, in the order that their sides are listed in any display. Usually, you can choose freely which units to move next; you can move one a bit, switch to another, move it, then come back to the first one later, and so forth. Some game designs may require that you move units in a specific order; perhaps all your aircraft must finish all their movement before any ships can move.

2.7.1 Turn Setup

First, *Xconq* computes the number of action points available to each unit. Each unit gets an increment of action points equal to its `acp-per-turn`. Actions during a turn reduce this down; when it reaches a value less than the cost of any action, the unit cannot do anything more until the next turn.

The range of action points for a unit is normally 0 up to the value of `acp-per-turn`, but the parameters `acp-min` and `acp-max` may allow for an extended range. You use this range by allowing a unit to accumulate extra action points by doing nothing for several turns, or to recover from an activity that used many action points all at once. Think of this as a sort of temporary action "debt". Units in debt at the beginning of a turn cannot act during that turn.

2.7.2 Types of Actions

Actions are the most basic kinds of things your units can do. During play, the interface will usually give you capabilities that are easy to use, such as the ability to point at a destination and have the unit figure out which path to take to get there, but all such input eventually breaks down into sequences of actions. You will therefore find it useful to understand all the types of actions available.

Movement Group:

- *Move to* a given location. The unit being moved may be in a transport or out in the open, the destination is any location in the open (this will usually, but not always, be an adjacent cell), and may be at any altitude allowed for the unit.
- *Enter* a given transport unit. The transport need not be on the same side as the entering unit.

Combat Group:

- *Attack* a given unit. A successful attack causes damage and destruction to the unit being attacked.
- *Overrun* a given location. The overrunning unit attempts to occupy the destination, capturing, ejecting, or eliminating any unfriendly unit present.
- *Fire at* a given unit, possibly using a given material as ammunition.
- *Fire into* a given location, possibly using a given material as ammunition.
- *Capture* a given unit.
- *Detonate* at a given location. Detonation causes damage to all unprotected units in the vicinity of the detonation.

Construction Group:

- *Research* a given unit type. This increases the tech level for the type being researched.
- *Tool up* to build a given unit type.
- *Create* a unit of the given type. The unit will usually be incomplete.
- *Build* a given unit towards completion.
- *Repair* a given unit, restoring lost hp.

Unit Manipulation Group:

- *Disband* a given unit, causing it to disappear.
- *Transfer part* of a unit, either to another given unit, or creating a new unit.
- *Change side* of a given unit to a given side.
- *Change type* of a given unit to a given type.

Material Manipulation Group:

- *Produce* a given quantity of a given material type.
- *Transfer* a quantity of a given material type to a given unit.

Terrain Manipulation Group:

- *Add terrain* of a given type to a given location.
- *Remove terrain* of a given type from a given location.

Normally, you as the player and the side simply tell units to perform these actions themselves. However, some games will allow the unit to cause the action to done as if another unit were doing the action. For instance, a transport can pick up or drop off a non-moving unit.

Not all interfaces can be guaranteed to allow the most general forms of all these actions; you must consult the interface's documentation to find out which of these actions is available.

2.7.3 Movement

Movement into a cell is easy to request, but each game will have many rules constraining possible moves, depending both on the unit and the terrain it is moving over. Certain kinds of terrain cost extra points to enter, leave, or cross. The destination must almost always be adjacent to the unit's current location.

The other kind of action is to enter/leave a transport. The only argument is the unit to enter, but again the constraints are complicated. The transport must have sufficient space, both the entering unit and the transport must have sufficient mp and acp to complete the move, and the entering unit must be able to cross the intervening terrain. The transport may be able to *ferry* the would-be occupant over any barriers; possibilities include no ferrying, ferrying only over the transport's terrain, ferrying over any borders, and ferrying over all terrain between the would-be occupant and the transport.

In some games, you may be able to make one of your units leave the world entirely. Sometimes this will seem like a good idea, perhaps to keep a trapped unit from falling into enemy hands, or because you win the game by leaving through a designated place. To do this, you just direct your unit (which must already be at the edge of the world) to move into one of the cells along the edge. If the departure is allowed, then the unit will simply vanish and be out of the game permanently.

In other games, you may be able to do a *border slide*. This is where a unit can jump to a non-adjacent cell if the two cells have a border whose endpoints touch the starting and ending cell. This is typically allowed in games so that ships can go through narrow straits.

2.7.4 Combat

> War is a matter of vital importance to the State; the province of life or death; the road to survival or ruin. It is mandatory that it be thoroughly studied. – SUN TZU (ca 400 BC)

There are two basic kinds of combat, each with two versions. A unit can attack or overrun, meaning that it comes to grips with an enemy in some way, or it can fire, meaning that it keeps its position and throws rocks or whatever at a target.

Attack is directed at a particular unit, while *overrun* is a more complex action where the unit attempts to clear enough units from a given location so that it can move in.

A unit wishing to attack picks a position or unit to attack, *Xconq* computes the defender's response, then the outcome is computed.

In many games, that will be the end of a fight. In others, the units remain engaged in a *battle*, and they cannot do any other type of action until they have disengaged completely.

Firing can happen at long ranges, up to the `range` of a unit. It may or may not involve using a specific material as ammunition; if the game gives you a choice, you will have to choose which, or else all possible types will be used. You can *fire at* a specific unit if you can see it, otherwise you will have to *fire into* a cell; perhaps without knowing whether or not you're actually hitting anything in it.

Some units are capable of capturing other units, with a probability depending on the types of both units involved. If the capture attempt is successful, the capturer will move into the cell if possible, either as occupant or transport. In some games, the capturer may be all or partially disbanded, to serve as guards. Capture may also occur as a side effect of a normal attack.

Detonation is a special kind of "combat" available to some units. The action requires a location - either the unit's position or a nearby cell. Upon detonation, the detonating unit may lose some hp and even die (changing to its "wrecked type", if defined, or else vanishing). At the same time, it makes one hit on any units within its radius of effect. Detonation may also be triggered automatically, such as by damage to the unit or even by another unit appearing nearby.

2.7.5 Research

> Knowledge is power. – FRANCIS BACON (1597)

Research increases a side's tech for the unit type being researched. Although you can only research a specific type of unit, some game designs allow for a crossover effect, where increases in the tech level for one type also increases the level in others.

You can have more than one researcher researching the same type, and thereby speed up your progress, but some games put a ceiling (`tech-per-turn`) on how much progress you can make in one turn.

2.7.6 Construction

We must be the great arsenal of democracy. – FRANKLIN ROOSEVELT (1940)

Tooling up prepares a unit to create or construct the desired type. As with research, game designs may allow a crossover effect for tooling. Tooling may also decline gradually over time; this is called *tooling attrition*.

Actual construction of a unit happens in two steps; creation and building towards completion. Most interfaces will also schedule research and toolup actions if a unit is told to build something that needs tech or tooling first.

Creation is the actual step of bringing a new unit into existence. If the new unit is *complete*, then it can be used immediately. If not (the usual case), then the incomplete unit will exist and belong to your side, but be unable to do anything at all. Incomplete transports cannot have any occupants, unless they are types capable of helping complete the transport.

Completion is achieved by doing build actions on the unit. Multiple units can all work on completing the same unit, but they must be sufficiently close, within a range defined by the game (usually the same or an adjacent cell). In some games, there is a level of completion past which the unit will start working on itself automatically, and eventually become complete without any further action.

It is *usually* the case that the same unit will be able to both create and complete a unit, but if not, you will have to pay special attention to your construction plan, since an incomplete unit cannot act in any way.

Note that multi-part units will be considered "complete" when just one of their parts is completed. Most interfaces will have the builder continue growing the just-completed unit as long as it remains within construction range.

2.7.7 Repair

Repair restores lost hit points to a unit. Repairs can be done by the damaged unit itself, if it is not too badly damaged, or by another unit that is close enough.

Some games also feature automatic hit point recovery, so you don't always have to remember to do explicit repair actions.

2.7.8 Disbanding

Disbanding is a voluntary loss of hp, ultimately resulting in the disappearance of the unit. Most games only allow it for a few types of units. Depending on the game, you may be able to disband the unit in one turn, or you may need several turns before the unit actually goes away.

Units with occupants can disband, but only if the occupants are unaffected by the action. If the unit would vanish or lose transport capacity, then the occupants must be disbanded or removed first. The interface may arrange to do this for you automatically.

You always get back all of the disbanded unit's supplies, and they will be distributed to other units nearby. In addition, the disbanded unit itself may become a source of materials. A percentage of the total material will become available after each action, if disbanding takes several turns to accomplish.

2.7.9 Transferring Parts

In games where units can vary in size, you can shift one or more parts of a multi-part unit to another unit, or else create an entirely new unit.

You would use this action if, for instance, you wanted to detach a survey party from an exploring expedition, then rejoin later.

2.7.10 Changing Side

In many games, you can give some of your units to another side. You may also be able to take them from another side, if you control that side.

Unlike most actions, you may be able to cause a unit to change side without actually expending any action points, if the game definition allows.

2.7.11 Changing Type

A few games allow you to change the type of a unit.

For instance, you might have this ability in a construction-oriented game, where you can take a town that has accumulated sufficient building materials and change it into a city. Another possibility is that you have increased your technology level and are now able to transform a low-tech ship into a higher-tech ship.

2.7.12 Producing Materials

Production is how a unit can produce a quantity of a material.

In many games, units already have a *base production* that is the amount of material that they produce automatically each turn. This will often depend on the terrain, so that explorers in the forest will always “produce” enough water to drink each turn, but will start to use up their water supply when in the desert.

2.7.13 Transferring Materials

Often there will be plenty of some type of material in the world, but the problem is getting it from the units that have a lot, to the units that need it badly. The *transfer* action is how you move supply from one unit to another.

As with production, many games have some automatic transfers set up. For instance, games involving aircraft generally refuel them automatically whenever the aircraft has landed in a place with fuel to spare.

2.7.14 Changing the Terrain

In some games, units can add or remove borders, connections, or coatings, or may even be able to change the overall type of terrain in a cell. The actions are *add-terrain*, *remove-terrain*, and *alter-terrain*, respectively.

The change happens immediately (for the sake of simplicity), but in practice, you may find that preparing for the change may take awhile. For instance, the unit executing the change might have to accumulate acp or materials required for the change.

2.8 Automation of Units and Sides

Specifying the exact sequence of actions and their operands for every single unit would be mind-numbingly complex. It's not very realistic either! Therefore, *Xconq* includes several levels of automation for human players.

The elements of automation are the *task*, the *plan*, the *doctrine*, and the *strategy*. These are related to each other by *goals*.

Tasks are single activities of a unit that require one or more actions to accomplish. Examples of tasks include moving to a given position, or waiting 15 turns to be picked up by a transport.

A *plan* is the unit's object that expresses its decided-upon behavior. Elements of a plan include a type, goal, and *task agenda*, as well as more specific slots, such as a pointer to the unit currently under construction. All units that can act and that are on a side will have a plan, while independent units that can act may have one if preset by a scenario. Plans primarily govern individual behavior, in many cases allowing the unit to act on its own, without needing any explicit direction from the player.

The *doctrine* is the set of parameters governing how the side will play and how its units should work generally. For instance, per-unit doctrine specifies the point which a unit low on supply should start to look for a place to replenish itself.

The *strategy* and associated subobjects is what an AI uses to make all the decisions about what to do. This object is not directly visible, unless the AI is acting as your assistant and the interface includes a display of its current strategy.

Of all these types of objects, only the doctrine can be manipulated directly; all others are implicitly changed as a result of player commands, which are different for each interface.

2.8.1 Doctrine

There is a doctrine for each type of unit on your side. Several types may share a single doctrine, so changes to it will affect all types equally.

- wait-for-orders

  This is true if a unit should wait for explicit orders to be issued. If false, the unit should make up some sort of default plan and follow it.
- resupply-at
- rearm-at

[more doctrine info]

2.8.2 Plans

A unit's plan must be one of the types listed here.

- None. This type of plan does absolutely nothing.
- Passive. Units with a passive plan will execute any tasks they have been given, but will not add to the task agenda on their own.

  [auto-add tasks if required by doctrine?]
- Offensive. Units with an offensive plan will look for favorable combat opportunities, usually within an area specified as their goal to hold.
- Defensive.
- Exploratory. Exploratory units will seek to collect information about unknown parts of the world.

2.8.3 Tasks

Each task in a plan's task agenda must be one of the types listed here.

- Do nothing.
- Build new units, a given number of a given type. This task will do research actions if necessary and possible, and toolup actions if necessary. Also, if there is an incomplete unit of the given type nearby, this task will complete it before creating a new unit.
- Stand sentry at the present location for a given number of turns.
- Move in the given direction up to the given distance.
- Move to within a given distance of the given location.
- Move towards another given unit.
- Patrol an area around one or two given points.
- Attempt to hit a unit at a given location.
- Attempt to capture a unit at a given location.
- Resupply.
- Repair.

2.8.4 Time Limits

One reason to automate your units is that some game designs define real-time limits on the length of a game. For instance, the game might be set to end in one hour, a single turn might be limited to always last at most 2 minutes, or your side might be limited to 15 minutes of playing time, in the manner of a chess clock. If such limits are in effect, your display should be able to show you how much time you have left at any moment; pay attention!

When you run out of time, you are not automatically taken out of the game, but you can no longer do anything with your units. Units that already have plans will continue to act on them.

The game design may give you a limited number of "timeouts" that you can call to stop the clock. The timeout ends when you order a unit to do something.

[how do players find out about time limits?]

2.9 Standard Keyboard Commands

These commands should be available in all versions of *Xconq*. Additional commands may be defined for some interfaces; see the interface's documentation for more details.

`' '` `reserve` put into reserve for this turn

`'?'` `help` display help info

`'!'` `detonate` detonate

`'.'` `recenter` center around the current point

`'#'` `distance` display distance to selected place

`'a'` `attack` attack

`C-A` `auto` toggle AI control of unit

`'b'` `southwest` move southwest

`'B'` `southwest mult` move southwest multiple

`'C'` `clear plan` clear unit plans

`C-C` `end turn` end activity for this turn

`'d'` `delay` delay unit action until after others have moved

`'D'` `disband` disband a unit

`'f'` `fire` fire

`'F'` `formation` set formation

`'g'` `give` give supplies

`'G'` `give-unit` give unit to side

`'h'` `west` move west

`'H'` `west mult` move west multiple

'j' `south` move south

'J' `south mult` move south multiple

'k' `north` move north

'K' `north mult` move north multiple

'l' `east` move east

'L' `east mult` move east multiple

'm' `move to` move to a place

'M' `message` send a message to another side or sides

C-M `end turn` end activity for this turn

'n' `southeast` move southeast

'N' `southeast mult` move southeast multiple

'o' `other` other commands

'p' `produce` set material production

'P' `build` set up construction tasks

'Q' `quit` get out of the game

'r' `return` return to a resupply point

C-R `refresh` refresh display

's' `sleep` go to sleep

't' `take` take supplies from unit or terrain

`'T'` `take unit` take unit from another side

`'u'` `northeast` move northeast

`'U'` `northeast mult` move northeast multiple

`'w'` `wake` wake units up

`'W'` `wakeall` wake units and all their occupants up

`'y'` `northwest` move northwest

`'Y'` `northwest mult` move northwest multiple

`'z'` `survey` switch between surveying and moving

The following commands are not standardly bound to single keystrokes.

`add player` allow another player to come into the game

`ai` toggle the AI

`copying` display the copying rules

`name` set the name of a unit

`print` print

`version` display the version and copyright

`warranty` display the non-warranty

If designing is enabled, then the command

`design`

will enable and disable design mode. See Chapter 3 to find out more about what you can do in this mode.

If debugging hooks are enabled, then the commands

`D`

`DG`

`DM`

will be available. They cause detailed transcripts of general computation, graphics, and AI behavior, respectively. They act as toggles, and are independent of each other, so you can control what kind of information is output. The transcript will go to stdout or to a file, depending on the interface and system.

2.10 Environmental Conditions

Some games include *environmental effects*, which includes what we normally think of as weather; the temperature, clouds, wind, rainfall, snowfall, and snow cover on the ground.

The temperature falls in a range specified by the game, and may be computed in different ways depending on the game design, but typically depends on terrain, latitude, the severity of the seasons, and elevation. Temperature may also vary randomly from turn to turn and cell to cell. The contribution of each of these to the final temperature is up to the game design, as is the *effect* of temperature.

For each type of unit, there is both a *comfort range* and a *survival range* of temperatures. Units within their comfort range are unaffected by the temperature. Units outside the comfort range, but within the survival range, may experience reduction in acp and an increase in attrition. Units outside the survival range die instantly. [add a prob, a al starve?]

A game may include clouds. Their chief effect is to affect the seeing of units on the other side of the clouds.

Wind affects the weather by causing clouds and storms to move around. Certain unit types, such as sailing ships and balloons, may depend on the wind to move around.

Games may assert that the playing area represents part of a sphere, possibly tilted on its axis, and that poles and equator correspond to various latitudes, which has the effect of producing seasons. The game specifies the temperature extremes for poles and equator, for both midsummer and midwinter, then the weather phase interpolates to get the average temperature for the current turn and at each latitude in the world.

2.11 Economy

The economy in *Xconq* is based upon materials. Games that do not include any material types do not have any of the activities described in this section.

2.11.1 Consumption

Units consume their supplies, both in the course of existence, and by motion/combat. The rate depends on game and unit type; it consists of an overhead consumed each turn without fail, and consumption for each cell of movement. The total is a max, not a sum, since units with a constant consumption rate are not likely to need additional supplies to move (consider foot soldiers who eat as much sitting around as they do walking). Supplies may also be consumed for production and repair, again depending on game and unit types, but this consumption happens during the build phase. Consumption is not affected by the situation of the consuming unit; armies in troop transports eat just as much as when in the field.

2.11.2 Movement of Materials

Excess production is discarded, unless it can be unloaded into the producer's occupying units, or distributed to nearby units via *supply lines*. Supply lines automatically exist between units that are close enough (as set by the game), and there is no need for explicit manipulation.

Supply line length depends on the game and the units on both ends, but is not affected by the intervening terrain. Supply redistribution is managed by logistics experts, who are ignorant of the war effort and seek only to even everything out. The redistribution method is rather adhoc; units try to get rid of all their excess supply, and try to take up supply from other units within supply range. Each direction is controlled independently, so for instance airplanes can get automatically refueled from a nearby city, but not from each other. No unit will transfer all of its supply via supply lines. Normally units in the same cell can exchange supplies, but some games can disable this behavior, so that explicit transfer using the give and take commands is always necessary.

2.12 Random Events

Some games may include *random events*. These are usually rare, but not always – be sure you know the odds!

2.12.1 Accidents

For some types of units in some types of terrain, there is a chance for an accident to wreck or eliminate the unit instantly. This depends on both unit type and terrain type. If the accident occurs, the unit is wrecked or vanishes along with all its occupants. "Wrecking" and "vanishing" have separate probabilities. Occupants may survive wrecking, but never vanishing.

2.12.2 Attrition

Attrition is "slow death"; it takes away some number of hit points each time it occurs. The rate of attrition depends on unit type and terrain or transport type. Very low attrition rates may only take away one hp once in a while.

2.12.3 Revolt

In a revolt, a unit changes sides spontaneously, perhaps to independence, perhaps to the side of a nearby unit. Occupants will either change over or be killed. Any plans will be cancelled.

2.12.4 Surrender

Surrender only occurs if a unit is capable of attack or capture is close enough to attempt it. The capturing unit does not move. Occupants of the surrendering unit also change over or die.

2.13 Scoring

> Victory at all costs, victory in spite of all terror, victory however long and hard the road may be; for without victory there is no survival. – WINSTON CHURCHILL (1940)

Different games can have different ways for players to win or lose. Some games may not have any scoring at all. You should be aware of the scoring rules *before* you start to play the game!

In *Xconq*, scoring is implemented by a game design's *scorekeepers*. Each scorekeeper tests some sort of condition and/or maintains a numeric score. Scorekeepers also define when they run (perhaps only during certain turns or certain times within a turn) and which sides to look at. Each scorekeeper is independent of the others, meaning it only takes one to decide if you win or lose.

In a game with many players, winning and losing can be a complicated issue; read the conditions carefully. A scorekeeper can also decide to declare a game to be a draw and end it on the spot.

Once a side has won, it is out of the game. Some scorekeepers only allow one winner, others allow several; in those cases, the scorekeeper will say what happens to the winning side's units.

Once a side has lost, it cannot be brought back into a game, even if another side tries to give it some more units or otherwise to reverse things.

It may also be possible to declare a draw, but all players in a game have to agree to this. While human players just have to enter the appropriate command (or answer appropriately when asking to quit the game), AIs may not always be willing to go along, particularly if they think they still have a chance to win. If that happens, you must continue fighting. (Some cowards have been known to abort the program or reboot the machine; unfortunately *Xconq* cannot prevent such slimy tricks.)

Finally, some games may record everybody's final scores into a file.

2.13.1 "Last Side Wins"

The most common form of scoring in *Xconq* is called `last-side-wins`. It is basically a fight to the death; any side that loses all of its units loses, and the last side with any units remaining is declared the winner. It is possible that more than one side will lose all of its units at the same time, in which case the game is declared a draw.

Since this would sometimes lead to bizarre stalemates (a submarine could hide at sea, thus preventing the side from losing, for instance), many games also define a *point value* for units. In such cases, `last-side-wins` makes a side lose when the sum of point values of all its units is zero, and the interface will have some way to display your current points.

[following sections should help player interpret scorekeeper displays]

2.13.2 Occupation

Occupation means that you have one of your own units in or near a fixed location or unit.

2.13.3 Unit Counts/Sums

This is a simple count of units, or else a summation of the values of some property, such as hit points.

2.14 Advanced Play

This section covers additional features that may interest experienced players.

2.14.1 Mixing Game Modules

Some interfaces (such as those using Unix-style command lines) may let you ask for more than one game design when starting up. This is sometimes useful, for instance, if you want to play on the `steppes` world with a non-standard set of units; your command line might look like `-g my-hacked-standard -g steppes`. You can also turn things around and load a file with your own changes after a complete game, as in -g gettysburg -g my-tweaks.

Be aware, however, that this cannot be guaranteed to work always, since the mixed-together game designs may have mutually conflicting definitions, or interfere with each other in subtle or not-so-subtle ways. Just imagine the disaster if the world consists entirely of terrain that is instant death to your initial units! Worse, *Xconq* may start up and run OK for awhile, then at the moment you're about to win—the object that you must capture simply cannot be captured by any unit at all.

So be careful about mixing designs!

2.14.2 Personalizing Your Side

Many games will pre-assign your side's name, emblem, enemies, and so forth. However, many others allow you to change all that to suit your tastes.

The name is a proper noun such as "Poland", the noun is what you would call an individual, such as "Pole", the plural is for more than one, and `<adj>` is the adjective for things on that side, such as "Polish". The color scheme is a comma-separated list of color names, and `<image name>` names some sort of image file (like a bitmap).

The image may be of any size and combination of colors, with the caveat that it may not always work correctly. For instance, two subtly different shades may get fused into a single solid color. The emblem should also be small enough to fit reasonably into unit icons. As a rule, most national flags will fit into a 7x5 rectangle, and coats of arms into a 7x9 region. The color scheme should be useful by itself, when the unit icons are too small to fit the emblem.

Xconq will not allow you to have the same name, color, or emblem as another player in the same game.

The interface-specific side configuration uses the favored mechanism for that interface (if one is defined). You should check with the interface documentation for more details.

2.15 Playing Hints

This section is a collection of bits of information and advice derived from players' actual experience playing *Xconq*.

2.15.1 Alliances

Informal alliances frequently happen in games involving more than two people, so I have a few words of advice. First, an alliance between two of the players is almost certain in a three-person game, and inevitably results in the "odd man out" being quickly defeated. In four-person games, the alliances could be decided after looking at the map via a command-line option such as `-v`, so that one pair is not hopelessly separated. Five or more players is going to be a free-for-all of formal and informal alliances. Some scenarios are designed with a particular number of players in mind.

2.15.2 Advantage

When you set the advantage, *Xconq* multiplies the desired advantage with the normal number of starting units, then divides by the default advantage and ROUNDS DOWN. This means that you might end up with a lot fewer units than you thought. For instance, suppose that you have a game where each player starts with one large city and five towns, and this is considered to be an advantage of 10, because one large city is worth about as much as 5 towns. Then if you select an advantage of 8, and your opponent selects 14 (because you're a better player perhaps), *Xconq* will give you 8/10 of the normal setup, which means four towns and NO large city. Your opponent will get 14/10 of the setup, which works out to one large city and seven towns, which is really a 1 to 3 disparity, much more than the planned 4 to 7.

2.16 Cheating

There is none. The standard builtin AI `mplayer` does not cheat; it always plays according to the same rules as you do. This should be true of any AI that has been added to *Xconq*. If you have evidence that would seem to indicate that any AI is using information it should not have, or is otherwise cheating, that is a bug and should be reported.

2.17 Technical Details

The coordinate system is "oblique", with the X-axis in the usual horizontal, and the Y-axis vertical, but tilted to the right at a 60-degree angle.

```
          Y
    \   /
     \/
---------X
    / \
   /   \
```

The additional left-leaning axis is the x = - y line.

2.18 Introduction to X11 Xconq

2.18.1 Installing

No special installation is required.

2.18.2 Resources

For instance, in your X resources, you would say:

```
Xconq.*,<game>.SideName: <string>
Xconq.*,<game>.SideNoun: <string>
Xconq.*,<game>.SidePluralNoun: <string>
Xconq.*,<game>.SideAdjective: <string>
Xconq.*,<game>.SideColorScheme: <color names>
Xconq.*,<game>.SideEmblem: <image name>
```

where `<game>` identifies the specific game or game design, and the resource names correspond to side slots in the obvious way.

2.19 Playing X11 Xconq

2.19.1 Starting a New Game

2.19.1.1 Command Options

[xref generic options - or put in sep file?]

2.19.2 Maps

Once the game has started, you have at least one "map window" open. (For brevity, these can be called just "maps"). Each map window has identical capabilities, so you can play by using just one, or have one for each area of interest, or have some of them serve specialized purposes, such as a map of the whole world.

Each map window consists of a number of panes, whose size you may adjust by dragging the small square grips that are somewhere along the pane boundaries.

The most important subwindow of a map is the map view itself. [etc]

Each map may have one *current position* and one *current unit*. The current position is specially highlighted, and the info subwindow displays information about it. If in addition there is a current unit at the current position, then it will be highlighted rather than the entire cell, and the info subwindow will describe it in detail.

[details/example of info subwindow]

2.19.2.1 Scrolling

It will nearly always be the case that the world is too large to be seen all at once. You can scroll around in two ways. First, if you are in survey mode, and click near any edge of the view, *Xconq* will put the position you clicked at the center of the view. By clicking in the same place repeatedly, you can "walk" the view in any desired direction.

If you want to go directly to a particular part of the world, use the panner in the bottom right corner of the map window. To use the panner, click and drag the shadowed box inside the panner. The panner is sized to match the map, and the shadowed box is sized to match the view, so you can get a general idea of much is visible of the whole world.

2.19.2.2 View Control Popup

If you click the button "More..." in the leftside controls, you will get a popup dialog that is a full set of viewing controls. Unlike the leftside view controls, these do not act at once; instead, you toggle them on or off, then click on "Apply" or "Done" to see the effects. This allows you to make a number of changes and have the map be redrawn only once. "Apply" leaves the popup in place, while "Done" makes it go away. You may leave the popup up permanently and continue play, if you like.

Note that each map will have its own separate view control popup, and that they're not clearly distinguished from each other. Fortunately, these only affect display, not the game itself.

2.19.3 Play

The basic idea of play is to be in move mode, let the program select the next unit to do something, then give it a command, either by clicking the mouse or by typing on the keyboard.

Each map may have two modes; "survey" and "move". In survey mode, the default actions are not to do anything, while in move mode, the default actions are to do things. This principle applies to both mouse and keyboard commands. For instance, 'h' in survey mode moves the current position west by one cell, but in move mode it causes the current unit to try to move west by one cell.

2.19.3.1 Using the Mouse - er - Pointer

Although in classic X style, all the actions may be arbitrarily rebound, for simplicity the default *Xconq* setup uses the left button for the most important actions.

2.19.3.2 Using the Keyboard

The general commands all work as described in Chapter 2.

Commands that work on units will be applied to the current unit of the map that the pointer is over. If the map has no current unit, then you will get an error message.

Commands that need further input will generally request it from the prompt subwindow that is sandwiched between the history subwindow and the date subwindow. You do not need to put the cursor over the prompt window to type into it however; when a prompt is up, any typed characters will be considered to be part of the answer to the prompt.

2.20 Designing with X11 Xconq

There are two ways to get into designer's mode; you can either specify '`-design`' on the command line, or else enter the long command '`design`' after doing the o command. When you do either of these, several things will happen: the side list will mark your side as a designer, the "see all" button in the view control panel will be enabled, and you will get a popup with designer's tools.

The popup applies to all maps equally.

The identification in the side list is for the benefit of other players, since it is actually possible to enable designing in a multi-player game.

The “see all” view control allows you to see everything accurately. It is switchable so you can compare what things look like to the player vs what they are in reality.

Shortcuts:

* Clicking on the "sides" subwindow will select the current side.

* Clicking on the "unit types" subwindow will select the current unit type.

* Clicking a cell with the right mouse button in terrain-drawing mode will select the current terrain.

* Clicking a unit with the right mouse button in unit-adding mode will select the current unit type.

* Clicking a unit with the right mouse button in people-drawing mode will select the current side.

* Clicking a cell with the right mouse button in feature-drawing mode will select the current feature.

2.20.1 Xshowimf

Install the resource files XShowimf.ad and XShowimf-co.ad as XShowimf and XShowimf-co [or XShowimf-color, depending on the value of the customization resource] in the proper directory, or load the relevant file with "xrdb -merge".

Display of three-color images (mono+mask) is controlled by the resource "maskColor" (see XShowimf-color.ad) or by the command-line argument "-mc"

On a sun, compile with gcc, not with cc.

To use as an image family editor, use as mkdir tmp xshowimf [imf/xbm/xpm files...] -o tmpdir & (cd tmpdir; xpaint) & and see help window; don't forget to "update" edited files.

2.21 Introduction to Mac Xconq

2.21.1 Installing

Xconq requires no special installation once you have unpacked it. Since the distribution consists of a number of files and folders in addition to the application proper, it will be less confusing to keep *Xconq* in a separate folder.

'`Xconq`' is the game application. It has been tested on nearly every kind of Mac (including PowerMacs), all running 7.0 or better. No init/extension compatibility problems have been reported so far.

'`lib`' is a folder of game modules.

'`lib-mac`' includes Mac-specific files referenced by game modules. *Xconq* doesn't absolutely have to have these files, but you may lose some pictures or sound effects without them.

'`doc`' is the folder of generic documentation for players and game designers. It does not have any Mac-specific information. The format is Texinfo, which is based on TeX, which you'll need TeXtures or OzTeX and the texinfo.tex file to format, sorry.

'`IMFApp`' is a small utility for game designers to display and convert images used in *Xconq* games. See below for information on how to use '`IMFApp`'.

2.21.2 Playing an Introductory Game

Double-click on the *Xconq* icon. You get an initial screen with several buttons. Click on New. You get a list of available games. The "Introductory" game should be highlighted; if not, then find it in the list and click on it. Then click OK. You then get a list of sides and players; click OK again. After some activity, you see several windows open up.

The most important window is the map. It's mostly black, indicating that you don't know anything about what's out there. You should see a small picture of a city, and a small picture of a diagonal line of soldiers below it; the soldiers have a heavy black outline around them. This means that they are ready to move. Note that the cursor is a arrow, and that it always points away from the soldiers. This means that if you click anywhere in the map, the soldiers (your "infantry unit") will try to move to the place you clicked. This happens the same whether you click in an adjacent hex (or "cell") or one far away, or even somewhere out in unknown regions. Your infantry is smart enough to find its way around some obstacles, and will stop and wait for new orders if they are blocked.

Click to move the infantry a few times. Notice that the turn number is changing at each move. Then all of a sudden *Xconq* will highlight a new infantry! Your city was working on it quietly while you were moving the first one; now *Xconq* will ping-pong between the two, asking first about one, then the other. Get used to it - it won't be long before you have dozens or even hundreds of units at your command! Fortunately for your sanity, *Xconq* provides some assistance - more on that later.

At this point you should have several infantry wandering around. Use them to explore the world - send each out in a different direction so as to learn the most the fastest. Eventually they will have spread out so much that *Xconq* has to scroll over to each one before asking about it. This can be a little disorienting. One thing you can do is to go to a lower magnification for the map; either click on the small mountains picture in the lower corner of the map window, or go to the "View" menu and pick something from "Set Mag". With a little experimentation, you will see that you can magnify so much that a single hex nearly fills the window, or make the whole world appear to be the size of a postage stamp.

[control options]

[finish describing startup of intro in great detail]

2.22 Playing Mac Xconq

2.22.1 Starting a Game

The splash screen gives you four choices: New, which brings up a list of games; Open, which allows you to pick a file; Connect, which will (when it works) allow you to pick a game to join; and Quit, which lets you escape.

Usually you will want to choose New, which brings up a dialog listing all the games. You can select one and see a brief description of it.

You can also load a game from a file by clicking on the "Open" button. This just uses the standard Mac file-opening dialog. You can restore a saved game this way. Double-clicking on a saved game or other game file works too.

2.22.1.1 Loading a Game

Whether you've chosen from New Game or Open Game, *Xconq* will go through a loading process, which may take a while if the game is large or complicated.

You may get some warning alerts, which are often benign (such as an inability to find some images), but others are indicative of disaster ahead. If you see one and continue anyway, don't be surprised if the game goes up in a cloud of smoke later!

2.22.1.2 Variants

If the game includes any "variants", you will then get a dialog with assorted buttons and checkboxes to choose from. For instance, some games let you choose whether the whole world is visible when you start, or what kind of scoring system to use.

Different games have different variants, but there are several used by many games.

The "World Seen" checkbox, when set, makes the whole world seen right from the beginning of the game. This only affects the initial view, and you will only see some types of units belonging to other players, such as their cities.

The "See All" checkbox makes everything seen all the time, right down to each occupant of each unit of each side. This makes *Xconq* more like a boardgame, where everything is "on the table" (so to speak).

The "World Size" button brings up a dialog that you can use to change the dimensions of the world in everybody will be playing. In *Xconq*, the available area of the world is either a hexagon, or a cylinder wrapping entirely around the world. You get the cylinder by setting the circumference equal to the width of the area. See the generic player's manual for more details about world size and shape.

["Real Time" button]

2.22.1.3 Player Setup

The player setup window shows the sides that will be in the game and who will play each side. As with the variants, you will often just want to accept it (click "OK"), since the game's designer usually sets the defaults reasonably.

If you want to change the setup, you first need to understand the current set of sides and proposed players. Each entry in the list of sides starts off with the side's emblem (if it has one), followed by the name of side, then in italics, some information about the player, and then the initial advantage for the player. You, the person sitting in front of the screen, is described as "You", while players that are actually run by the computer are described as "AI mplayer", "AI" being short for "artificial intelligence" (In some games, a player may be a specialized AI, named <name>, in which case it will be described as "AI <name>".)

In games that allow you to have more than the default number of sides, you can just click the "Add" button. All the other controls require you to have selected a side/player pair. You can do this by clicking anywhere in one of the boxes describing the side/player pair, which will be highlighted in response.

The "Computer" button toggles the AI for that side. You can add an AI to any side (including your own side; more on that later). You can also remove the AI from any side; a side with no AI and no human player will just sit quietly and do nothing throughout the entire game.

If you don't like the side you're on, you can use the "Exchange" button to switch. The ordering of the sides is fixed, so exchange just exchanges players between the currently selected side/player pair and the next one.

2.22.1.4 Final Setup

When all the setup dialogs have been OKed, *Xconq* will finish setting up the game. For some games, this will take quite a while - *Xconq* generates random terrain, positions countries so that they are neither too close nor too far apart, and does many other things to set up the game, so just kick back and watch.

Once everything is set up, *Xconq* then opens up the game window, the instructions window, and one map window for you. The map shows you terrain with different patterns, and your playing pieces (units) with small pictures.

Note that some *Xconq* games allow the AI either to act first, or to start acting as soon as the windows come up. You may even find yourself being attacked before you know what's happening!

2.22.2 Playing a Game

The basic pattern of play is to find a unit in a map or list window, click once to select it, and then pick commands from the "Play" menu. There is also an "auto-select" mode that does the selection for you and interprets mouse clicks as movement commands; *Xconq* actually starts up in this mode [, but this is a preference you can set].

You can select units by clicking on a unit, shift-clicking a group, dragging out a selection rectangle, or by using Command-A to select all units. A selected unit is indicated by an outline box - solid black to indicate that it can move, dark gray to indicate that it cannot move, and gray to indicate that it cannot do anything at all (at least during this turn; some types of units may only get to do something once in a while). If clicking on a unit image doesn't have any effect, then it's not a unit that belongs to you.

To move a selected unit, drag the selected unit to its desired new location. The unit might not move right away if it doesn't have the action points, but it may get some in the next turn. To move all selected units, do Command-click on the desired location and all of them will attempt to move there.

To shoot at another unit, you can position the mouse over the desired target, type 'f', and all selected units will attempt to fire. This works even if all units are selected, so you can call down considerable destruction with 'f'! If the target is too far away, nothing will happen.

To find out more about a unit, pick "Closeup" from the "Play" menu or do Command-I. This brings up a window that shows all kind of data about a single unit. You can leave this window up and it will always be kept up-to-date.

To jump ahead to the next turn, do the menu command "End Turn" or `<return>`. You may have to do this if some of your units still have action points, but not enough to do any of the things you want them to do.

The Game window (Command-1) shows the status of all the sides in the game. The window shows both the emblem (if available) and name of each side. A small computer icon indicates that an AI is running the side, while a face icon indicates your side's relationship with the side (frowning = enemy, etc).

Each side also has a progress bar that shows how many actions its units have left to do.

[describe auto-selection]

2.22.3 Menus

This section describes all the menus.

2.22.3.1 File Menu

The File menu includes the usual sorts of commands that all Mac programs share.

[New Game]

Brings up the new game dialog.

[Open Game]

Brings up a standard file dialog. *Xconq* will assume the selected file to be a game definition and attempt to load it as such.

[Connect]

Use the Connect item to join in a game that is already running elsewhere. (Not implemented yet.)

[Save]

Saves the game to a file.

[Save As]

Saves the game to a file, with a name chosen from a dialog that pops up.

[Preferences]

Brings up a dialog that you can use to select various options. [need to describe preferences]

[Page Setup]

[Print Window]

Prints the front window.

[Quit]

Leaves *Xconq*.

2.22.3.2 Edit Menu

Note that there is no Undo. Hey, this game is a life-or-death struggle, and you may have noticed that you don't get an Undo in real life either...

[Cut] [Copy] [Paste] [Clear]

These are not currently supported either (who's that knocking at the door? Oh no, it's the Human Interface Police! Please don't take me away, I was going to get around to figuring what Copy was supposed to mean for a strategy game, honest! Nooooooooo...)

[Select All]

Selects all units that you are currently allowed to select. Most commands will operate on multiple unit selections, so this is a powerful (and therefore dangerous) option. For instance, if you select all units then put them all to sleep, nobody will do anything at all.

[Design...]

The Design item is for access to game designer tools. You cannot use these during a normal game; you will be asked to confirm that you want to design, and if so, *Xconq* will change all the windows appropriately and bring up a special designer's palette. This is a one-way activity; once somebody in the game becomes a designer, all players will be notified and computer-run players will no longer bother to play. (In case it's not obvious, this is because it's too easy to cheat using the designer's powers.)

2.22.3.3 Find Menu

This menu is for various kinds of searching.

[Previous Actor]

[Next Actor]

[Location]

[Unit by Name]

[Selected]

Scrolls the most recently-used map over to show the selected unit in a list.

2.22.3.4 Play Menu

This menu is the main set of commands that you can give to individual units. When you specify one of these, the units affected will be whatever is selected in the window. If the window is of a type that does not have selected units (such as a help window), then the items on this menu will be disabled.

[Closeup]

Opens closeups (see below) of all the selected units.

[Move]

[Patrol]

[Return]

Directs the selected units to return to the closest place where they can replenish supplies that have been used up and/or get repairs to any damage.

[Wake]

Wakes up the selected units.

[Sleep]

Puts the selected units to sleep.

[Reserve]

Puts the selected units into reserve.

[Build]

Brings up the construction window and selects the first of the selected units that can do construction.

2.22.3.5 Side Menu

This menu is for overall control of the side you're playing.

2.22.3.6 Windows Menu

This menu is for the creation and arrangement of windows.

2.22.3.7 View Menu

The View menu gives you control over the appearance of the window you're currently looking at. Each kind of window that has any view controls will have a different view menu. Currently only map and list windows have view menus. Each window has its own view settings, although you can set defaults for new windows from the preferences. You can find the descriptions of each view menu's items under the description of its window, below.

2.22.4 Windows

Xconq lets you have many windows open at once. Each type has its own specialized functionality.

2.22.4.1 Map Windows

[picture of map window]

A map window gives you an overhead view of some part of the *Xconq* world. As you might expect, you can scroll around to look at different parts. You can also zoom in and out using the small zoom icons in the lower left corner; zooming in ("closer mountains" icon) makes the cells larger, while zooming out ("farther mountains" icon) makes the cells smaller, so you can see more of the world. You can zoom way in or out!

The optional "top line" of the map window supplies you with information about what the cursor is currently over, plus the current game date.

The map control panel is along the left side. At its top is the auto-select/move-on-click button. Below that is a set of find-next/previous buttons. The next set of buttons is controls for how the map will be displayed. These behave identically to the map's View menu items.

Map View Menu

Since maps are the main interface to *Xconq* games, you have many options for controlling their appearance.

2.22.4.2 Game Window

The game window shows you the turn number or date of the current turn, as well as any realtime clocks that may be counting down, and a list of all the sides. For each side, you see its name, the emblem for that side, a progress indicator, and icons indicating the side's attitude and whether it is being played by the computer. You may also numbers indicating scores and/or real time remaining.

The progress bar shows how much movement a side has done during the turn. This usually goes down during the turn, but because it indicates a percentage rather than an absolute number of actions, the percentage may go up sometimes. For instance, if some of your units that have already acted are captured, then the percentage goes up because the *total* number of actions has gone down! A gray bar indicates that the side has finished all movement for this turn. There may also be a dashed vertical line in the bar, which indicates the percentage of units that are asleep or in reserve. Note that a player can always wake up sleeping or reserve units at any time before the end of the turn.

2.22.4.3 List Windows

A unit list window just lists all the units, one line each. This is useful for getting a more organized look at your assets. A unit listing shows the icon for the unit, its name and type, action points, hit points, supply, etc.

You can create more than one list window.

List View Menu List view controls typically either affect what will be listed, or the sorting of the list. There is also an item to control the size of the unit icons.

2.22.4.4 Unit Closeup Windows

For any unit, you can create a closeup window. This window supplies full details on the unit.

2.22.4.5 Construction Window

You use the construction window to control the construction of new units. The window comes in two parts; the left side is a list of all the units that can do construction, while the right side is a list of all the types that can be constructed.

2.22.4.6 Instructions Window

The instructions window is the basic info about what game you're playing and what you're supposed to do. Many game designs have few or no instructions. There is a Help button that just brings up the help window.

2.22.4.7 Help Window

Xconq's help information is organized into a list of topics. When you first open the help window, you will see the list, and a row of buttons. To look at a specific topic, just click on it.

The "Topics" button goes straight to the list of topics, while the "help" button shows you the topic describing the help system itself. "Prev" and "Next" buttons take you through the topics in order, while "Back" goes to the last topic you looked at.

2.22.5 Keyboard Commands

[list them all - get from help info automatically]

2.23 Designing with Mac Xconq

Designer's tools and capabilities are available via the Design item in the Edit menu. The display changes to show you everything in the world, you get a number of special privileges (such as the ability to look at and move ANY unit), and you get a designer's tools palette.

2.23.1 Using the Palette

In general, palette items use option-click and option-shift-click to cycle through possible values, and many also have a popup menu. Also note that the selection can be changed without changing the tool; you have to click in the tool and get a heavy border around it before the tool is actually changed. Each type of designer tool has a distinct cursor when over a map window, so look for that as well.

2.23.1.1 Painting Terrain

If you select the terrain item in the palette, then clicking/dragging in a map paints the current terrain type displayed in the palette (Option-clicking the terrain item cycles through all the types, shift-option-click does the same thing in reverse order). The background terrain type can be cycled via command-click and command-shift-click.

2.23.1.2 Creating Units

The side item creates a popup with the sides - use it to set the side with which a new unit will be created. Not all units are allowed on all sides - *Xconq* marks the sides that are allowed with the current unit type, and the unit types that are allowed with the current side.

2.23.1.3 Painting People

If you select the side item in the palette, you can click/drag in a map to set the side of the people in the cells clicked or dragged over.

2.23.1.4 Painting Material

You can create materials in the terrain by selecting the materials item in the palette, then painting in any map.

2.23.1.5 Creating Named Features

The features item in the palette includes several buttons and a popup menu that displays all the features currently defined. You can click on the buttons to create, destroy, and rename a feature. You can add or remove cells of a feature by painting in any map.

2.23.1.6 Painting Elevations

You can set the elevations of terrain by selecting the elevations item in the palette, then painting in any map.

2.23.1.7 Painting Temperatures

You can set the temperatures by selecting the temperatures item in the palette, then painting in any map.

2.23.1.8 Painting Winds

You can set the direction and force of winds by painting them. The values of direction and force are controlled by option-[shift-]click and command-[shift-]click, respectively.

2.23.1.9 Painting Clouds

If you select the clouds item of the palette, you can set the density of cloud cover by painting.

2.23.2 Beyond the Designer Palette

The designer palette and privileges just scratch the surface of what you can do with *Xconq*. You can define your own complete games with *Xconq* using its Game Design Language (GDL), a declarative Lisp-like language with many more capabilities than could be provided interactively (in much the same way that HyperTalk scripting adds to basic HyperCard)/ The *Xconq* manual chapter "Designing Games with *Xconq*" goes into much more detail about *Xconq*'s programmability, and the chapter "Reference Manual" is the complete description of GDL's abilities.

For the Mac specifically, there are some additional customizations that you can do.

If the Resources file has a PICT whose name is "<game-name> game", then if a player selects <game-name> in the new game dialog, this PICT will be displayed in the dialog. This is useful to give prospective players more of an idea of what the game might be like, plus it's a chance to show off your artistry! (If you're a lousy artist, just make a screen shot of the map and use that.) The area for the picture is about 200 pixels across and 100 high; pictures smaller than that will be centered, while larger pictures will be clipped to fit.

2.23.3 Images

If you want to change the icons and pictures in an existing game, or if you want to define new ones, you can do this either with a resource editor such as ResEdit, or with a resource compiler such as Rez, or by editing the portable "imf" files. *Xconq* can use PICTs, icons, and patterns to draw with; collectively these will be called "images". A group of images that all represent different views of the same thing is an "image family". The concept is similar to the icon families used by Macintosh programs, but is much more general, since you can have any number of images of different sizes and depths, as well as images embedded in each other.

The resource file :lib-mac:Images is the main repository of Mac imagery. You can resource-edit this file, close it, start up *Xconq*, and see the results. For instance, if you want to improve the appearance of the "city20" image, you will find a 32x32 'cicn' with that name as well as a 16x16 'sicn'. You can modify these as you like. Note that the 'sicn' includes a mask sicn, as does the 'cicn'. Masks are optional, but improve the appearance of the images. If you have a new type in a game and you want an image for it, just create a resource, such as a 'cicn', give it the name of the type or its "image-name" property, and you're done.

The standard resource types for images are 'cicn', 'ICON', 'PAT ', 'ppat', 'SICN', and 'PICT'. If a resource of one of these types has a name matching the "image-name" (or the type name if no image name is defined) of a type in a game, then *Xconq* will use that resource. There are two additional resource types: 'XCic' and 'XCif'. The 'XCic' resources are named colors that can be referred to via "color" properties of types and sides. 'XCif' resources are raw image family forms in GDL syntax. If the contents of an image family can be totally defined by the standard resource types, then there need not be an 'XCif' resource, but if an image has any embedded subimages, or a specific location for subimages, then this information will be in the XCif resource. You can edit by defining this resource type as 'TEXT', which will give you a text editor for the resource, or by just opening the 'TEXT' editor on the resource. For more details about the syntax of image families in GDL, see the general manual.

Warning: don't ever make a 'cicn' with id 256! For some reason, if there is a 'cicn' 256 in either the application or in any of the resource files that it has open, the small color icons in both the apple menu (far left) and the application menu (far right) become small unrecognizable blobs. This effect seems to be innocuous behaviorwise, but the appearance is poor, and users choose menu items by looking at those icons, so trashing them is a bad thing to do.

2.23.4 IMFApp

IMFApp is a utility for examining and converting the image families used by Xconq.

It is intended for use by game designers, as an aid to importing and exporting game-related imagery.

The general concept of image families is similar to the Finder icon families, where several depths and sizes of an icon are available for use in different situations. Xconq adds extra complexity to the concept by including tiling patterns, by allowing icons to be embedded in each other, and by using the same image families on several different platforms.

The platform-independent format is not very efficient to use and would be hard to edit, so IMFApp includes functions to convert between the common format and Mac resources. These functions are available from the File Menu.

In addition, IMFApp also allows you to look at the images. You can control the images' size, overlay, and use of color. The primary use for this is to test how an image works in various situations. Also, if you design games, this is a good place to start when choosing images for your game. Since there are hundreds of images available already, the chances are good that the images you want have already been designed.

The File menu has items to open and save imf and resource files. New clears any existing images. Opening multiple files merges their contents.

The Edit menu is not used.

The View menu includes all image appearance controls. Everything in this menu affects only the *display* of the images, and does not, in any way, affect the image families themselves. Display options include size of images, color/mono, name, mask, background/emblem images, and more.

2.23.5 Sounds

Mac *Xconq* handles sounds in a very simple way. The resource file '`:lib-mac:Sounds`' includes a number of named ''`snd `'' resources.

[not actually useful yet - names wired into macmap.c]

2.24 Troubleshooting Mac Xconq

If *Xconq* crashes, that is a serious problem; please report it, and include as much information about your setup, what you were doing, etc.

Xconq will sometimes display "error" or "warning" alerts. These can be caused either by bugs in *Xconq*, or more likely, by mistakes in the design of the game you're playing. For instance, you may be playing a version of a game that has been modified by one of your friends, but the modification was not done correctly, and you'll get an alert unexpectedly.

"Error" alerts are fatal; you may be able to save the game at that point, but don't count on it. Common ones include errors because you're loading a text file that is not an *Xconq* game, and running out of memory. Most error alerts occur during game startup, while *Xconq* is checking out the game definition that it's loading. Error alerts that appear during a game, and do not involve running out of memory, are more serious, and may indicate bugs in *Xconq*, so you should save the game and report what happened.

There are many kinds of "warning" alerts. Warnings are not fatal, but they do indicate that all is not well. If you get a warning alert and don't know what it means, it's safer to quit than to try to struggle on. Most warnings indicate mistakes in the design of the game you're playing, and should be reported to the game designer. The following describes several common types of warnings:

* Missing images: A game design may not have had images defined for all types of units and terrain. *Xconq* will warn about this, then make up some (ugly) default images itself. Actual game play will be unaffected.

* Sides have undesirable locations: A game can specify how close and how far away each side should be from all the others, and the kind of terrain each will start on. If the world is too small, or doesn't have the right kinds of terrain, then *Xconq* will warn about this. The game will still play normally, but it may be grossly unfair, and if the sides start out hidden from each other, it may be a while until it becomes obvious how unfair it really is.

You may also run into these bugs:

* Units sometimes appear or disappear unexpectedly. Type Control-R, which recalculates visibility of everything.

* In the construction window, clicking on "Build" doesn't always result in a unit being created immediately. It may be that the builder has used up its acp and can't start construction until the next turn, or that it hasn't come up for executing actions in the current turn. Clicking a second time will make the construction start immediately.

2.25 Introduction to Curses Xconq

Curses *Xconq* is a version that requires only an ASCII terminal and a *curses* library for cursor movement and screen management. As a result, it will run almost anywhere. [including DOS - should try to build it under go32 or some such]

However, in exchange for this higher degree of portability, you lose a lot in display power, and games may become much more difficult to play. For instance, roads and rivers cannot be represented directly, and you will have to rely on the textual displays to see which directions have them.

(Incidentally, this curses interface is the oldest one in Xconq, predating even the X10 interface that was part of version 1's release in 1987.)

2.25.1 Installing

The name of the executable is '`cconq`'.

[pathname to library info]

2.26 Playing Curses Xconq

When '`cconq`' starts up, it takes over the whole screen, in the traditional fashion of curses programs.

[include 60-char-wide term snap shot]

[verbal description of subwindows]

[move and survey modes]

2.27 Designing with Curses Xconq

Although '`cconq`' does not have the richness of display that is best for designing games, it does have a basic set of commands for that purpose.

3 Designing Games with Xconq

In this chapter, you'll learn how to design new kinds of games with *Xconq*. *Xconq* has been designed to support the use of a variety of techniques to design, construct, and test your game idea. These techniques range from text file editing to online painting, and you will likely find a combination of techniques to be most effective.

As the person customizing *Xconq*, you will be called the *designer*. This term also indicates the primary activity, which will be to Design The Game. The capabilities described below are merely tools; it is up to you the designer to exercise discretion and judgement in using them. Some principles of game design will be discussed in at the end of this chapter. Note that this chapter is merely an overview of game design machinery; for precise definitions, see Chapter 4. The glossary defines all the terms.

You design games using *Xconq*'s Game Design Language (GDL). GDL is *Xconq*'s common language for defining all parts of a game, from the entry in the menu that players select games from, down to the last tiny detail of a saved game. GDL resembles Lisp, although (at the present time) it is not a procedural language; there are no functions or even any control constructs. Instead, the contents of a file guide the creation or modification of *Xconq* objects representing types, tables, units, and so forth. While a game is being played, *Xconq* uses this data to decide what to do and what to allow players to do.

(People often have trouble with parentheses in Lisp, but if you follow the same kinds of indentation rules that you always use in C or Pascal, then you will encounter no additional trouble. Also, many editors such as Emacs are intelligent enough to indicate when parentheses match, and automatically do proper indentation.)

In this chapter, "you" always means means you the designer, and players will be referred to as "players" or "users". The distinction is important; as the game designer, you will encounter and deal with many technical issues relating to the inner workings of *Xconq*, but if you master those issues, your players will see only a fun game to play.

A final caveat before plunging in: *Xconq* is an experiment in the design and construction of configurable games. This means I have had limited prior art on which to build, and there are lots of odd corners that have never been tested or even thought about. In this spirit, I would like to hear about weird cases, and ideas for how to handle them.

3.1 A Tutorial Example

Before delving into the depths of the language, let's look at an example. Suppose you just finished watching a Godzilla movie, complete with roaring monsters, panic-stricken mobs, fire trucks putting out flames, and so forth, and were inspired to design a game around this theme.

3.1.1 Basic Definitions

Start by opening up a file, calling it something like `g-vs-t.g`, or some other name appropriate for your type of machine, and then type this into it:

```
(game-module "g-vs-t"
  (title "Godzilla vs Tokyo")
  (blurb "Godzilla stomps on Tokyo")
)
```

This is a GDL *form*. It declares the name of the game to be `"g-vs-t"`, gives it a title that prospective players will see in menus, plus a short description or *blurb*. The blurb should tell prospective players what the game is all about, perhaps whether it is simple or complex, or whether it is one-player or multi-player. Both title and blurb are examples of *properties*, which are like slots in structures.

The `game-module` form is optional but recommended; some interfaces use it to add the game to a list of games that players can choose from.

The general syntax of `game-module` form is similar to that used by nearly all GDL forms; it amounts to a definition of an "object" (such as a game module or a unit type) with *properties* (such as name, description, speed, etc). Some properties are required, and appear at fixed positions, while others are optional and can be specified in any order, so they are introduced by name. The general format, then, looks like

```
(<object> ... <required properties> ...
  ...
  (<property name> <property value>)
  ...
  )
```

There are very few exceptions to this general syntax rule.

Now the first thing you'll need is a monster. In *Xconq*, each unit has a type, and you define the characteristics attached to the type.

```
(unit-type monster)
```

This declares a new unit type named `monster`, but says nothing else about it. Let's use this more interesting form instead:

```
(unit-type monster
  (image-name "monster")
  (start-with 1)
)
```

This shows the usual way of describing the monster. In this case, `image-name` is a property that specifies the name of the icon that will be used to display a monster. The property `start-with` says that each side should start out with one monster. This isn't quite right, because there should only be one side with a monster, and this will give *each* side a monster to start out with, but we'll see how to fix that later on.

We also need at least one type of terrain for the world:

```
(terrain-type street (color "gray"))
```

Streets are to be gray when displayed in color, and get nothing if they are being displayed on a monochrome screen.

These two forms are actually sufficient by themselves to start up a game. (Go ahead and try it.) However, you'll notice that the game is not very interesting. Although each player gets a monster, and an area consisting of all-street terrain is displayed, nobody can actually *do* anything, since the defaults basically turn off all possible actions.

3.1.2 Adding Movement

Well, that was dull. Let's give the monsters the ability to act by putting this form into the file:

```
(add monster acp-per-turn 4)
```

The `add` form is very useful; it says to *modify* the existing type named `monster`, setting the property `acp-per-turn` to 4, overwriting whatever value might have been there previously. The acp-per-turn property gives the monster the ability to act, up to 4 actions in each turn. By default, the ability to act is 1-1 with the speed of the unit, so the monster can also move into a new cell 4 times each turn. If you run the game now, you will find that your monster can now get around just fine. Why 4? Actually, at this point the exact value doesn't matter, since nothing else is happening. If the speed is 1, then the turns go faster; if the speed is 10, then they go slower and more action happens in a single turn. In a complete design however, the exact speed of each unit can be a critical design parameter, and for this game, I figured that a speed of 4 allowed a monster to cover several cells in a hurry while not being able to get too far. Also, I'm planning to make panic-stricken mobs have a speed of 1, which is the slowest possible. Making actions 1-1 with speed is usually the right thing to do, since then a player will get to move 4 times each turn (later on we will see reasons for other combinations of values).

The add form works on most types of objects. It has the general form

```
(add <type(s)/object(s)> <property name> <value(s)>)
```

The type or object may be a list, in which the value is either given to all members of the list, or if it is a list itself, then the list of values is matched up with the list of types.

3.1.3 Buildings and Rubble Piles

To give the monster something to do besides walk around, add buildings as a new unit type:

```
(unit-type building (image-name "city20"))

(table independent-density (building street 500))
```

The building type uses an icon that is normally used for a 20th-century city, but it has the right look. The independent-density table says how many buildings will be scattered across in the world. The table form consists of the name of the table followed by one or several three-part lists; the two indexes into the table, and a value. In this case, one index is a unit type building, the other is a terrain type street, and the value is 500, which means that we will get about 500 buildings placed on a 100x100 world (look up the definition of this table in the index). You need some for testing purposes, otherwise you won't see any when you start up the game.

We're going to let buildings default to not being able to do anything, since that seems like a reasonable behavior for buildings (although Baba Yaga's hut might be fun...).

By default, buildings act strictly as obstacles; monsters cannot touch them, push them out of the way, or walk over them. In real(?) life of course, monsters hit buildings, so we have to define a sort of combat.

```
(table hit-chance
  (monster building 90)
  (building monster 10)
  )

(table damage
  (monster building 1)
  (building monster 3)
  )

(add (monster building) hp-max (100 3))
```

The `hit-chance` and `damage` tables are the two basic tables defining combat. The hit chance is simply the percent chance that an attack will succeed, while the damage is the number of hit points that will be lost in a successful attack. The unit property `hp-max` is the maximum number of hit points that a unit can have, and by default, that is also what units normally start with.

Note that the `add` form allows lists in addition to single types and values, in which case it just matches up the two lists. The `add` tries to be smart about this sort of thing; see its official definition for all the possibilities.

The net effect of these three forms is to say that a monster has a 90% chance of hitting a building and causing 1 hp of damage; three such hits destroy the building. A monster's knuckle might occasionally be skinned doing this; a 10% chance of 3/100 hp damage is not usually dangerous, and feels a little more realistic without complicating things for the player.

Now you can start up a game, and have your monster go over and bash on buildings. Simulated wanton destruction!

By default, a destroyed building vanishes, leaving only empty terrain behind. If you want to leave an obstacle, define a new unit type and let the destroyed building turn into it:

```
(unit-type rubble-pile (image-name "???"))

(add building wrecked-type rubble-pile)
```

In practice, you have to be careful to define the behavior of rubble piles. What happens when a monster hits a rubble pile? Can the rubble pile be cleared away? Does it affect movement? Try these things in a game now and see what happens; sometimes the behavior will be sensible, and sometimes not.

For instance, you will observe that the default behavior is for the rubble pile to be an impenetrable obstacle! The monster can't hit it, and can't stand on it, and in fact can't do anything at all. OK, let's fix it. Monsters are agile enough to climb over all sorts of things, so the right thing is to let the monster co-occupy the cell that the rubble pile is in. The default is to only allow one unit in a cell, but this can be changed:

```
(table unit-size-in-terrain (rubble-pile t* 0))
```

This says that while all other units have a size of 1, rubble piles only have a size of 0. By default, each terrain type has a capacity of 1, so this allows one unit and any number of rubble piles to stack together in a cell.

If you try this out, you'll find that the monster can now cross over rubble piles, but still has to bash buildings in order to get them out of the way.

Incidentally, it can cause problems to set a unit size to zero, because it allows infinite stacking. Since buildings and rubble piles don't move, there will never be more than one in a cell, but *Xconq* will happily let hundreds of units share the same cell, which works, but causes no end of headaches for players confronted with overloaded displays.

3.1.4 Human Units

Now you've got an "interactive experience" but no game; there's no challenge or goal. You could maybe make a two-or-more-player game where the players race to see who can flatten the mostest the fastest, but that's still not too interesting to anyone past the age of 5. Instead, we need to make some units for the people bravely (or not so bravely) resisting the monster's depredations:

```
(unit-type mob (name "panic-stricken mob") (image-name "mob"))
(unit-type |fire truck| (image-name "firetruck"))
(unit-type |national guard| (image-name "soldiers"))
```

Note that a type's name may have an embedded space, but then you have to put vertical bars around the whole symbol (a la Common Lisp). Things are starting to get complicated, so let's define some shorter synonyms:

```
(define f |fire truck|)
```

```
(define g |national guard|)

(define humans (mob f g))
```

You can use the newly defined symbols `f` and `g` anywhere in place of the original type names. The symbol `humans` is a list of types, and will be useful in filling several propertys at once.

As with monsters, all these new units should be able to move:

```
(add humans acp-per-turn (1 6 2))
```

The speeds here are adjusted so that monsters can chase and run down (and presumably trample to smithereens) mobs and guards, but fire trucks will be able to race away.

Also note the use of a three-element list that matches up with the three elements in the `humans` list. This is a very useful features of GDL, and used heavily. It can also be a problem, since if you add or remove elements from the list `humans`, every list that it is supposed to match up with also has to change. Fortunately, *Xconq* will tell you if any lists do not match up because they are of different lengths.

We still need to define some interaction, since monsters and humans can make faces at each other, and get in each other's way, but otherwise cannot interact.

```
(add table hit-chance
  (monster humans 50)
  (humans monster (0 10 70))
  )
```

This time we have to say "add table" because we've already defined the `hit-chance` table and now just want to augment it.

As with the addition of properties, we can use a list in place of a single type.

Last but not least, we need a scorekeeper to say how winning and losing will happen. This is a simple(-minded?) game, so a standard type will be sufficient:

```
(scorekeeper (do last-side-wins))
```

The `do` property of a scorekeeper may include some rather elaborate tests, but all we want to is to say that the last side left standing should be the winner, and the symbol `last-side-wins` does just that.

There might be a bit of a problem with this in practice, since in order to win, the monster has to stomp on all the humans, including fire trucks. But fire trucks can always outrun the monster, and cannot attack it directly either, which leads to a stalemate. You can fix this by zeroing the point value of fire trucks:

```
(add f point-value 0)
```

Now, when all the mobs and guards have been stomped, the monster wins automatically, no matter how many fire trucks are left.

3.1.5 The Scenario

As it now stands, your game design requires *Xconq* to generate all kinds of stuff randomly, such as the initial set of units, terrain, and so forth. However, we *are* doing a monster movie, so random combinations of monsters and people and terrain don't usually make sense. Instead of trying to define a "reasonable" random setup, we should define a scenario, either by starting a random game, modifying, and saving it, or by text editing. Since online scenario creation is hard to describe in the manual, let's do it with GDL instead.

To define a scenario, we generally need three things: sides, units, and terrain. Now the basic monster movie idea puts one monster up against a bunch of people acting together, so that suggests two sides:

```
(side 1)

(side 2 (name "Tokyo") (adjective "Japanese"))
```

The `1` and `2` identify the two sides uniquely, since we'll have to match units up with them in a moment. The side that plays the monster is really a convenience; players should just be aware of the one monster unit, so we don't need any sort of names. The other side has many units, which should be qualified as `"Japanese"`, and the side as a whole really represents the city of Tokyo, so use that for the side's name.

Now for the units:

```
(unit monster (s 1) (n "Godzilla"))

(unit firetruck (s 2))
(unit firetruck (s 2))

(building 9 10 2)

(define b building)  ; abbreviate for compactness' sake

(b 10 10 2)
(b 11 10 2 (n "K-Mart"))
(b 12 12 2 (n "Tokyo Hilton"))
(b 13 12 2 (n "Hideyoshi's Rice Farm"))
(b 14 12 2 (n "Apple Japan"))
;; ... need lots of buildings ...
```

This example shows two syntaxes for defining units: the first is introduced by the symbol `unit` and requires only a unit type (or an id, see the definition in xxx), while the second is introduced by the unit type name itself and requires a position and side. The second form is more compact and thus suitable for setting up large numbers of units, while the first form is more flexible, and can be used to modify an already-created unit. In both cases, the required data may be followed by optional properties in the usual way.

Also, since the word "building" is a little longwinded, I defined the symbol "b" to evaluate to "building". GDL has very few predefined variables, so you can use almost anything, including weird stuff like "&" and "=". Property names like `s` and `n` are NOT predefined variables, so you can use those too if you like.

At this point, you should have a basic game scenario, with one player being Godzilla, and the other trying to keep it from running amuck and flattening all of Tokyo. Have fun!

You can enhance this scenario in all kinds of ways, depending on how ambitious you want to get. Given the basic silliness of the premise, though, it would be more worthwhile to enhance the silliness and speed up the pace, rather than to add features and details. For instance, name the buildings after all the laughingstock places you know of in your own town.

To see where you could go with this, look at the library's `monster` game and its `tokyo` scenario, which include fires, different kinds of terrain, and other goodies.

3.2 Types

Types are the foundation of all *Xconq* game designs. Types are like classes in object-oriented programming but simpler; each set of types is fixed and used only in a particular way by *Xconq*. A game design defines types of units, materials, and terrain. Only materials are optional; every game design must define at least one unit type and one terrain type.

Types in GDL are simple compared to most other languages. There is no inheritance, no subtyping, no coercions or conversions. This is not a real limitation, since game designs are usually too small to make effective use of any sort of inheritance. Also, game design is an exacting activity; inheritance is often difficult to control satisfactorily. You can use lists of types to simulate inheritance as necessary; this is actually more flexible, because you can have any number of lists with any set of types in each. It may not seem as efficient, but GDL is only used during startup, and is almost entirely array- and struct-based during the game. (A few places, such as scorekeeping, examine GDL forms during play.)

Types are defined one at a time in the game module file. Each type gets an index from 0 on up, in order of the type's appearance in the file. Although this is not normally visible to you or to the player, some error messages and other places will make reference to raw type indices. Each category of type - unit, material, and terrain is indexed individually.

3.2.1 Unit Types

Unit types define what the players get to play with. Unit types can include almost anything; people, buildings, airplanes, monsters, arrows, boulders, you name it.

The basic form of a unit type definition is so:

(`unit-type` *type-name* (*property-name property-value*) . . .)

The appearance of this form in a file means you are adding a new and distinct type, which has no relation to any other types defined before and after this one. The *type-name* must be a unique symbol, such as `building` or `|fire truck|`. (Note that you can set things up so that players never see the *type-name* anywhere, so don't worry if your preferred name conflicts with something else, just choose another name.) The *property-name* and *property-value* pairs are entirely optional. They can always be defined or changed later in the file. There is little advantage one way or another.

This particular syntax - keyword followed by name or other identifier followed by property/value pairs - will be used for most GDL definitions.

The number of unit types is limited. The exact limit depends on the implementation, but is guaranteed to be at least 127. This is a huge number of types in practice; the only situations where this might be needed would be a fantasy-type game with many types of items and monsters. For empire-building games, 8-16 unit types is far more reasonable. Keep in mind that with lots of types, players have more to keep track of, internal data structures will be larger and take longer to work with, and designing the game will take more time and energy. Consider also that *Xconq* gives you a lot of properties that you can set individually for each unit type, so that when other game systems might require a distinct types, *Xconq* lets you use the same type with different propertys. For instance, in a fantasy game you wouldn't need to define "young dragons" and "old dragons" as distinct types, instead you can vary the hit points or experience of a generic "dragon" type.

3.2.2 Terrain Types

Each cell in the world has a terrain type. This type should be thought of as the predominant contents of the cell, whether it be open ground, forest, city streets, or the vacuum of deep space. The type can be anything you want, and should be adapted to fit the game you're designing. Sure, the real world has swamps, but if you're designing a game set in the Sahara, don't bother defining a swamp terrain type. Also, the type doesn't carry any preconceptions about elevation or climate, so you can have swamps at 20,000 feet just as easily as at sea level.

The limit on the number of terrain types is large (up to about 127, depending on the implementation), but in practice, 6-10 types offer variety without being confusing. Ideally, several of those types will be uncommon in the world, so that map displays will consist mostly of 3-4 types of terrain.

Some game designs involve entities that are very large and do not move around. Such entities could plausibly be represented either as non-moving units or as a distinct terrain type. To make the right choice, you need to consider the special characteristics you want to implement. Terrain cannot (usually) be changed during the game, nor can it be moved, but units can be damaged or belong to different sides. A realistic example of this choice occurs in the monster game - should a destroyed building become a "rubble-pile" unit or should the building stand on rubble-pile terrain and vanish when it is destroyed? Both choices are plausible; if the rubble-pile is a unit, then the original building is then on top of an empty city block, and after the building is destroyed, the rubble-pile unit can itself be cleaned off, exposing the empty city block again. However, you have to decide whether the rubble-pile unit belongs to a side, how it interacts with other units, and so forth. Rubble-pile terrain is simpler, but the players then get descriptions of brand-new buildings sitting in the midst of rubble-piles, which is confusing. This is a case where there is no "right" answer.

3.2.3 Material Types

Material types are the simplest to define. They have only a few properties of their own; most of the time they just index tables along with the other types. Materials do not act on their own in any way; instead, players manipulate materials as part of doing other actions. For instance, you can specify that movement, combat, and even a unit's very survival depends on having a supply of some material, or that some material is ammo and consumed gradually when fighting.

The use of materials is pretty much up to you. You don't have to define any material types at all, and game designs with materials are usually more complicated. However, the increase in realism is often worth it; with materials you can limit player activity and/or make some actions more "expensive" than others.

As with the other types, you can define up to about 127 material types, but that would be enough to model the entire global economy accurately! (and take all week to compute a single turn...) 1-3 types is reasonable.

3.2.4 Static Relationships Between Types

The next sections describe the "static" relationships between types of objects, meaning those relations which must always hold, both in the initial setup and throughout a game.

3.2.5 Stacking

By default, *Xconq* allows only one unit in each cell at a time. This has the advantage of simplicity, but also makes some bizarre situations, such as the ability of a merchant ship to prevent an airplane from passing overhead or a submarine from passing underneath.

To fix this, you can allow players to stack several units in the same cell. This is governed by several tables, which give you control over which and how many of each type can stack together in which kinds of terrain. The basic idea is that a cell has a certain amount of room for units, as specified by the terrain type property `capacity`, and each unit has a certain size in the cell, according to the table `unit-size-in-terrain`.

```
(add (plains canyons) capacity (10 2))

(table unit-size-in-terrain
  ((indians town) plains (1 5))
  ((indians town) canyons (1 2))
  )
```

In this example, a player can fit 10 indians or 2 towns into a plains cell, or else one town and 5 indians, while canyons allow only 2 indians or one town.

In addition, some unit types may be able to count on a terrain type providing a guaranteed place; for this, you can use the unit/terrain table `terrain-capacity-x`. This table (which defaults to 0) allows the specified number of units of each type to be in each type of terrain, irrespective of who else is there. For instance, a space station could be given space via

```
(table terrain-capacity-x (space-station t* 10000))
```

So while units on the ground are piling together and being constrained by capacity, space stations overhead can stack together freely (space is pretty big, after all).

3.2.6 Occupants and Transports

Occupants and transports work similarly to stacking in terrain; there is both a specialized capacity and a generic capacity that units' sizes count against.

```
(add (transport carrier) capacity (8 4))

(table unit-size-as-occupant
  ((infantry armor) transport (1 2))
  ((fighter bomber) carrier (1 4))
  )

(table unit-capacity-x
  (carrier fighter 4)
  )
```

It may be that all the different sizes interact so that you can't prevent huge numbers of small units being able to occupy a single transport. To fix this, use `occupants-max`.

Transport is a physical relationship, so for instance one cannot use transports to define a convoy whose acp-per-turn is determined by its slowest member. (This doesn't mean you can't define a convoy type, but you will have to pick an arbitrary speed for it.)

Watch out for unexpected side effects of setting the `capacity` but not the `unit-size-as-occupant`! Since `unit-size-as-occupant` defaults to 1, then a unit with a nonzero capacity can by default take on *any* other type as an occupant!

Also, don't let units carry others of their own type. Not only is this of doubtful meaning, *Xconq* is not guaranteed to cope well with this situation, since it allows infinite recursion in the occupant-transport relation. Ditto for loops; "A can carry B which can carry C which can carry A".

3.2.7 Hints on Types

It is tempting to try to define independent sets of types, each in a separate module, and glue them together somehow. However, this doesn't work well in practice, because in a game, the types interact in unexpected ways. Suppose, for example, that you define a set of airplane types that you want to be generic enough to use with several different games. The assessment of those types may vary drastically from game to game; in one, airplanes are 100 times faster than any other sort of unit, so that moving airplanes takes up 99% of game play, while in another, the same set of airplane types are too weak to be of any interest to players.

There is a standard set of terrain types called `"stdterr"`. This set has a mix of the types found most useful for "Empire-type" games, and Earth-like percentages for random world generation.

3.3 Setting up a Game

You have a spectrum of options for how *Xconq* will set up a game based on your design. At the one end, you can build a scenario that specifies everything exactly, down to the last unit. Lest you think this is too restrictive to be interesting, consider that this is how chess works... At the other end of the spectrum, you can let *Xconq* manufacture everything, starting only with a handful of numbers that you supply.

The next several sections describe the alternatives available for game setup. It is important to understand what is possible, because in general the character of an *Xconq* game will depend strongly on the initial setup, and players will be very angry (with you!) if they discover, several hours into a hard-fought game, that they've been given a grossly unfair starting position.

3.4 Designing the World

The *Xconq* world/area is a two-dimensional grid of fixed shape and size. You can treat it as representing part of a planet in space, and set up parameters simulating that, or just make it be itself and not address the question of the surrounding context. The appropriate choice depends on how much realism and complexity you need. Most computer games don't bother with this detail; for instance, a game set in an underground dungeon doesn't usually need to compute daylight, weather, or seasons. However, these same details may be very useful for games set outdoors.

3.4.1 World Shape and Size

Once you've decided whether the area is to be part of a planet or not, you can address the question of size and shape. You have two choices for shape: hexagon and cylinder. (See the players chapter for pictures of these.) The important thing for you as a designer is that the cylinder wraps around, while the hexagon is bounded on all sides. One consequence is that games involving pursuit will be quite different; on a cylinder, the chase can go 'round and 'round forever, while on a hexagon, a fleeing unit could be cornered. Cylinders have a disadvantage in that there is no obvious "starting place" for coordinates, scrolling, etc, so there is a navigation and orientation problem for players, especially if the world is randomly generated and not the familiar continents of the Earth. In fact, players will often not even realize that a world is a cylinder and will assume that the edge of the display is the edge of the world! To make a cylindrical area, set the circumference of the world equal to the width of the area. Otherwise, the area will be handled as a hexagon.

You can choose either to set a fixed size using the `area` form, or allow players to set the actual size via the `world-size` variant, in which case you can define the allowable range of sizes.

Worlds need not be really large. Larger worlds are harder for players to manage, they take longer to display, and can consume prodigious amounts of memory (since they are represented as arrays internally, for speed). The ideal range of sizes depends primarily on the size and speed of units. A 60x60 area in a game with units whose speed is 1 means that they will take 60 turns to cross, while units with a speed of 20 take only 3 turns, so they make the world "feel smaller". As another example, in the standard game, a 20x20 area allows player to come to grips quickly, but it also means that each player's units might be within attack range right from the outset, which has a drastic effect on strategy. For exploration-oriented games, larger worlds are more interesting.

3.4.2 World Terrain

The best technique for designing the terrain of a world is to use the designer tools provided with *Xconq*. The details of how these tools work depends on the interface, but in general they resemble the tools found in paint programs. Some interfaces also give you the option of rescaling the map, so that you can fine-tune the size and positioning of the terrain.

Another technique is to write a program that translates data from another source (such as NASA satellite data) into *Xconq* format. However, if you take a rectangular array of data and just wrap an `area (terrain ...))` form around it, then everything will appear to be tilting to the left. To fix this, have your program map the cell at `x, y` in the rectangular array to `x - y / 2, y` before writing. You must discard values whose new `x` coordinate is negative, or else wrap them around to the right side of the area, although that is usually only reasonable for cylindrical areas.

The crudest technique is to try to build terrain by using a text editor. The coordinate system is Cartesian oblique, with the y axis tilted to form a 60-degree angle with the x axis, so it can be difficult to relate typed-in characters to the final appearance. Landforms in the file should appear to be leaning to the left, if they are to appear upright during play. However, sometimes text editing is necessary, for instance when you need to change every instance of a terrain type to something else. (Incidentally, some of the large real-world maps in the library were produced by coding all the terrain types from an atlas onto graph paper, typing them in, then fixing the tilt as described above.)

Incidentally, areas should have some distinguishing terrain around the edges; this prevents player confusion that sometimes happens when there is no other clue as to where the edge might be. However, this is not enforced by *Xconq*, and you can put whatever you like along the edges. Randomly generated worlds normally use the value of the global variable `edge-terrain`.

3.4.3 Synthesizing World Terrain

The random way to get terrain for a world is to use one of several synthesis methods built into *Xconq*.

Totally random terrain is available via the synthesis method `make-random-terrain`. This just randomly chooses a terrain type for each cell, using the weights in the `occurrence` property of each type. An `occurrence` of 0 means that the type will never be placed anywhere. This method produces a sort of speckly-looking world, and is better for testing than for actual play. Still, if you have two types `vacuum` and `solar-system`, then a form like

```
(add (vacuum solar-system) occurrence (20 1))
```

will give you a nice starfield for a space game.

The fractal world method `make-fractal-percentile-terrain` descends from the most venerable part of *Xconq* (it was once a piece of Atari Basic code). It uses a fractal algorithm along with percentile-based terrain classification to make realistic-looking worlds with terrain and elevations.

To use this method, you first specify how many, what size, and what height of blobs to splash onto the world, and how many times to average cells with their neighbors. Then you specify the subdivision of all the possible altitudes and moisture levels into different kinds of terrain. For instance, desert in the standard terrain ranges from sea level (`alt-percentile-min` = 70%) to high elevations (`alt-percentile-max` = 93%) but only in the lowest percentiles of moisture (`wet-percentile-min` = 0%, `wet-percentile-max` = 20%). It is important that all percentiles be assigned to some terrain type, or the map generator will complain and subsitute terrain type 0 (the first-defined type); when designing terrain percentiles, it is helpful to make a chart with altitude percentiles 0-100 on one axis and moisture percentiles on the other. Note that overlapping on this chart is OK, and the terrain generator will pick the lowest-numbered terrain. Also note that you don't have to include every terrain type.

The `alt` numbers are also used to compute elevations for games that need them, but the `wet` numbers need not have anything to do with water at all; they could just as easily represent smog levels or vegetation densities. If you only want to use one of the two layers, just set the percentiles for the other to be 0 - 100 for all terrain types.

[should have an example]

The method `make-maze-terrain` produces a maze consisting of a mix of "solid", "passageway", and "room" terrain. It uses the `maze-room-density` and `maze-passage-density` properties of each terrain type to decide how much of each to use for rooms and passages. The method first does random terrain generation, using the `occurrence` property to decide how much of each terrain to put down (remember that `occurrence` defaults to 1 for all terrain types). Then it carves out rooms, and passageways between them. The passages and rooms are guaranteed to be completely connected.

The method `make-earth-like-terrain` attempts to model the natural processes and generate terrain as similar as possible to what is observed on Earth today.

You should note that at least one method for synthesizing terrain must be available, unless you can guarantee that terrain will be loaded from a file. The following subsections describe optional additional synthesis methods that you can include.

3.4.4 Rivers

You can use the `make-rivers` method to add rivers to the world. Rivers are basically water features that depend on terrain elevations, so they won't be generated unless both a river terrain type (either border or connection) and elevation data is available. You get them by specifying a nonzero chance for some type of terrain to be the location of a headwater (`river-chance`).

Xconq doesn't have any intuition about the behavior of water; it will happily trace rivers all the the way down to the bottom of the sea. Use the `liquid` property to tell `make-rivers` what types that rivers cannot touch. The method still traces the river's course, and resumes modifying terrain when possible, which means that the river can appear as both the inlet and outlet from a lake.

3.4.5 Roads

The `make-roads` method is a fairly generic method. It just picks pairs of units randomly and runs a road between them, attempting to share road segments and route through favorable terrain. Although simplistic, the results look pretty good.

You can make short bridges by tweaking the road density appropriately. Just allow roads from land to water, and water to land, but not from water to water.

Note that this method is only useful if there are actually units for the roads to connect.

3.4.6 Independent Units

For many games, it is useful to have independent units scattered randomly across the world. For instance, gold mines and treasure hoards would be good for an exploration game, and independent castles for a medieval game. You can set this up with the `make-independent-units` method.

3.5 Altitudes and Elevations

Xconq is basically a 2-dimensional game, but you can emulate a third dimension by defining elevations for terrain and altitudes for units above and below the terrain.

The main use of altitudes is to control interactions between certain kinds of units, particularly aircraft. For instance, a high-altitude bomber should be able to pass over a ship and under a satellite with impunity. In general, you define the "operating altitudes" of a unit, so in the example above, you could say that a ship is always at the surface, bombers operate at 1-10 km, and satellites at 100-10,000 km. If a unit has more than one operating level, then it can move up and down by normal movement actions.

Also, most details such as speed and material consumption are the same for a unit at any altitude. (Yes, such things vary in real life, but the effects are usually minor within the unit's normal operating range.)

Altitudes have a significant effect on combat. A unit at some altitude can only attack units at a specific range of altitudes up and down. Using the example again, you could define fighter aircraft to operate at 0-20km and be able to attack up and down 5km, while bombers can attack up to 10km down (i.e. down to the ground), but not up. Satellites remain invulnerable.

All this applies equally to units underground and undersea.

[need info about setting up other layers]

3.6 Designing the Sides

Sides represent the players in a game. They also serve as a repository of information shared by units, such as technology and knowledge of the world.

You should first decide how much about the sides will be predefined. If you're doing Eastern Front scenarios, it's very easy; you have Russians and Germans and that's it. If you're doing a science-fiction empire-building free-for-all, you may not have to specify anything more than a random side name generator.

3.6.1 Predefined Sides

For scenarios and similarly-restrictive games, the game design should create the sides directly, as in this example:

```
(side (name "Germany") ... (colors "black,gray") ...)

(side (name "Russia") ... (colors "red") ...)
```

Since the initialization machinery allows matching any player with any side, you can get away with being really vague. This will create four sides but not say anything about them:

```
(side)
(side)
(side)
(side)
```

If you're going to have predefined units on each side, then you should add an id to each side:

```
(side 1 (name "Germany") ... (colors "black,gray") ...)

(side 2 (name "Russia") ... (colors "red") ...)
```

Instead of `1` and `2`, you can also use, say, `ge` and `ru`; ids can be either symbols or numbers.

3.6.2 Side Library

If your game design does not predefine all the sides, you can define a *side library* using the `side-library` variable. Basically the library is a weighted list of collections of side properties, each formatted as a side definition. *Xconq* will use this library for any player that is allowed in the game but who does not have a side already, and select a side with a probability determined by the weights. Each item in the library will be used up to a limit that can be specified with each item; if the library has been exhausted before all the sides have been created, then the extra sides will just be assigned general defaults for their properties.

The side library here makes futuristic sides for players, making two of the sides most likely, but allowing others as well:

```
(set side-library '(
  (10 (name "Federation") (adjective "Federation") (class "fed"))
  (10 (name "Klingon Empire") (noun "Klingon") (class "klingon"))
  (5 (noun "Romulan") (class "romulan"))
  ((noun "Ferengi") (class "fed"))
  ((noun "Vulcan") (class "fed"))
  ))
```

Note that if the game design limits certain unit types to certain sides, the choice of sides will be more than just a cosmetic issue.

3.6.3 Limits on Sides

So that you can put upper and lower bounds on the number of sides in your game, GDL includes the variables `sides-min` and `sides-max`. As you might expect, every game design must allow at least one side. The upper limit on sides depends on the implementation, but is at least 7. Large numbers of sides can make a player's life very complicated, not to mention consuming vast quantities of memory, so you should try to limit the number of sides as much as possible.

Another important limit is based on the notion of *side classes*. Each side can have a side class, and multiple sides can belong to the same class. For instance, sides named `"Hyperborean"` and `"Germanic"` could both have class `"barbarian"`. The value of side classes is that unit types have a property `possible-sides` that limits which side class(es) a type can belong to. This is very important for any game in which different players should have fundamentally different sorts of units. To continue the barbarians example, it is basically impossible for any barbarian side to have even one Roman legion, whether by construction, capture, or even surrender. So you can do something like

```
(add legion possible-sides "roman")

...

(side 1 (name "Rome") (class "roman"))
(side 2 (name "Germania") (class "barbarian"))
(side 3 (name "Hyperborea") (class "barbarian"))
```

and ensure that Roman legions are always Roman.

3.6.4 Hints on Sides

Note that players tend to identify with the sides they're playing, so a game should allow for as much personalization as possible. On the other hand, some scenarios derive part of their flavor from predefinitions. For instance, a scenario with sides named "German" and "Russian", with appropriate colors and emblems, doesn't have quite the same feel when players rename them to "Subgenii" and "Simpsons".

A side can have a huge amount of state data, such as the current view. This rarely needs to be included in its entirety; synthesis methods will usually suffice to set view data correctly. Since total security is impossible with a predefined world, setting a side to have only a partial view won't necessarily be useful to keep players from knowing what that world really looks like.

3.7 Designing the Units

Once you've decided how to handle sides in your game, you can move on to the initial unit setup. Initial unit setup is very important, since it has a major bearing on how the rest of the game will go, and can be done in a number of different ways.

3.7.1 Predefined Units

GDL allows you to define everything about every starting unit in the game. This is a powerful approach, but requires much preparation. An advantage of predefined units is that there are no unpleasant surprises. For instance, suppose you designed an empire game with ships and cities, but a random setup leaves some players entirely landlocked. Not only will those players be *very* unhappy, they might come looking for you *before* they've calmed down!

Asking for initial units is pretty easy, you can either type them into a file or create them directly, using the appropriate designer tool in a game.

```
(city)
(city 11 12 1)
(city (n "Brigadoon"))
(city (@ 10 10) (n "New York"))
(city (@ 20 10) (n "London") (hp 22))
```

The only info that you absolutely have to supply is the unit's type. If the position is missing, the unit will be placed at a random location. If the side number/name is missing, the unit will be independent or on the first possible side.

While the type, position, and side of units is important, exact values of the other properties are rarely important for a scenario. Also, a unit with fewer filled-in properties can be used in different games. For instance, a list of the present-day major cities worldwide really needs only name and location for each; the game design can fill in everything else. One way to do this would be to set up an appropriate `unit-defaults` just before including the module.

To make units start inside transports, you need to specify the `t#` property for the occupant, and have its value be the id number or name of some other unit. Your players may get an error message if the occupant is not of an allowed type for the transport to hold.

3.7.2 Making Countries

Despite the advantages of predefining initial units, this doesn't help when you want variable groups of units to appear in a randomly-generated world. Instead, you should use the `make-countries` synthesis method. The basic idea is that the method picks a good location for each side's country, scatters an initial set of units around that location, then possibly grows the country outwards. You can do anything from small widely-separated countries to an interlocking nightmare resembling pre-Bismarck Germany. Because of this, and because of the requirement that this method generate random setups that are as fair as possible, you have a great many parameters to work with. These parameters should be tuned carefully - you will probably need to generate and study lots of initial setups, especially if your parameters constrain the countries very tightly; the method cannot backtrack to fix a poor combination of placements.

The first step in country generation is to select a location for each side's country. The location is a point that is the "center" of the country (the exact value will be unimportant to players, and is not used outside this method). The constraints are that the center of each country is farther than `country-separation-min` from the center of every other country, that the center is within `country-separation-max` of at least one other country, and that the given initial area of the country (as defined by `country-radius-min`) includes numbers of cells of each terrain type bounded by `country-terrain-min` and `country-terrain-max`.

The reason for the separation constraints is that having countries too close together or too far apart can create serious problems. Consider the poor soul who gets tightly sandwiched between two enemies, thus becoming lunchmeat, ha ha, or the not-quite-so-poor-but-still-unlucky player who ends up on the wrong side of a very large world. (Keep in mind that your players may ask for a much larger world than you were thinking of when you designed the game.)

The terrain constraints help you put the country in a reasonable mix of terrain. For instance, if you want to ensure that your countries include some land, but be on the coast rather than inland, then you should say that the country must have a minimum of 1 sea cell and 1 land cell. (In practice, the values should be higher, so you don't get small islands being used as entire countries and lakes being considered the ocean.) Keep in mind that these constraints may be impossible to satisfy, for instance if a particular world does not have enough of the sort of terrain that is being required in a country. If the basic placement constraints fail, *Xconq* will just pick a random location, warn about it, and then leave it up to the players to decide on whether to play the game "as it lies".

```
;;; Keep countries close together, but not too close.

(set country-separation-min 20)
(set country-separation-max 25)
```

Once *Xconq* has decided on locations for each country, it then places the initial stock of units. You define this initial stock via the unit properties `start-with` and `independent-near-start`. The `start-with` units start out belonging to the side, while the `independent-near-start` units are independent. The locations of these units are random within `country-radius-min` of the center, but are weighted according to the table `favored-terrain`. This table is very important; it is the percent chance that a unit of a given type will be placed in terrain of the given type. 100 is guaranteed to work, and 0 is an absolute prohibition. Since `make-countries` tries repeatedly to place each `start-with` unit until it succeeds, then even terrain with a `favored-terrain` value of only 10% will get used if there is no other choice, so the table affects the distribution of units rather than the number that get placed. If a starting unit cannot be placed on any available terrain, but can be an occupant, then *Xconq* will attempt to put it inside some unit already present. This is a good way to begin a game with aircraft at airports rather than in the air.

The upshot is that all this will do a reasonable layout if the parameters are set reasonably. If, however, `favored-terrain` is never `> 0` for the `start-with` units and the country terrain, but there is some other terrain type for which this would work, *Xconq* will change the terrain. If even that doesn't work, the method will fail [or just complain?].

This example is from the standard *Xconq* game:

```
(set country-radius-min 3)

(add city start-with 1)
```

```
(add town independent-near-start 5)

(table favored-terrain 0
  ((town city) plains 100)
  (town (desert forest mountains) (20 30 20))
  )
```

The net effect is to give each player one city outright and 5 towns nearby. Although created independent, these towns can be easily taken over right at the beginning of a game, so they are a kind of "warmup" (like the pushing of pawns at the beginning of a chess game). The `favored-terrain` table allows cities to appear only in plains, while giving more options to towns, since they can appear in deserts, forests, and mountains. Even so, towns are 5 times more likely to be in plains, which is reasonable.

The optional last step in country generation is to grow the countries outwards from the initial area. This is basically a simple simulation of the historical forces that give countries their variety of shapes. The algorithm works by deciding whether to add to the country each cell at each distance from the country's center. The chance depends on the terrain type and whether the cell has already been given to another country. Once a cell has been given to the country, then the method decides whether to add a sided or independent unit to the cell, or whether to change the side of an existing unit. Country growth stops when either the absolute maximum radius has been reached, or too few cells have been added to the country, whichever comes first.

This example is from one of the variants of the standard game:

```
(game-module "standard"
  ...
  (variants
   ...
    ("Large Countries" eval
     (set country-radius-max 100)
     )
  ))
```

The resulting effect is to make all the countries border on each directly.

3.8 Setup Miscellany

This section describes random things.

3.8.1 Technology

Technology, or tech for short, is useful when technological development is important to a game. There are several ways to use it.

One use of tech is to track the results of research. You do this by setting the initial tech of a side to (say) 0, then requiring a certain tech (say 60) in order to build a desired type. If a research action adds 1 to a side's tech, then it will take 60 research actions to gain the necessary level. The number of turns, of course, depending on how many actions the researcher can do each turn, and how many researchers are available. So for instance, 10 researching units results in the work being done in 6 turns instead. You can limit this schedule acceleration by setting `tech-per-turn-max`.

Another use of tech is to differentiate sides. Suppose you want to do a game involving earthlings and space aliens. The aliens can have satellites overhead that earthlings don't even know are there, they have equipment earthlings couldn't use even if they were able to capture it. However, earth scientists might learn something from it. To do all this, use `tech-to-see` and friends.

Tech is fundamentally tied to unit types. However, many games have a number of unit types that share technology. For instance, advances in bomber technology usually lead to advances in fighter and surveillance aircraft. The `tech-crossover` table is available for this purpose.

3.8.2 Creating Self-Units

Normally a player runs the side as a whole, and all the units on that side are disposable and interchangeable. However, you require one unit to represent the player personally among the units of the player's side; this unit is the *self-unit*. What this means is that if that unit is captured or dies, the player loses the game instantly. All the other units on the side will behave normally as for losing, either going over to the side that captured the player, becoming independent, or disbanding.

The idea is to increase the player's motivation for self-preservation. This is useful to introduce a risk of capture, assassination, and so forth. It also prevents bizarre and unrealistic strategies in some games.

For instance, it sometimes happens in empire-building games that players end up switching countries, because each captured another's country and neglected to defend their own. If each player got one capital city, and that city were to be a self-unit, then the owner would have to defend it at all costs!

To make this happen, you could do something like this:

```
(set self-unit-required true)

(add capital-city can-be-self true)

(add capital-city start-with 1)
```

3.9 Units and Actions

Players can do all kinds of things with their units. They can push the units around, they can make units build things, they can get into fights, or they can just let them sit around. You as the designer decide which kinds of things make sense in your game, then set up the action parameters appropriately. Is moving through swamps going to be slow? Can a small town build any kind of ship, or just small ones? How often can Godzilla breathe fire?

Now, what the players work with is the interface, which can do all kinds of intelligent things – whatever makes sense for that interface. However, no matter what the interface, no matter what kind of play automation, player input eventually breaks down into unit actions. The set of action types is predefined and can't be changed. They are also very primitive. Each action takes a number of arguments, such as the type of unit to build or the location to move to, the action just happens and either succeeds or fails on the spot. There are no actions that take longer than one turn to complete, and a unit can perform only one action at a time. This may seem horribly restrictive, but actions are just the low-level building blocks; players rarely see actions directly. You have to be aware of them because the game design specifies which unit types are capable of which actions. Each *Xconq* interface will adjust itself to disallow input that would result in types of actions that you have prohibited.

The number of actions that a unit can do in one turn is limited by its action points. A unit with zero action points cannot do anything at all. A unit with lots of action points can do lots of actions, unless each action costs many action points. You can define the action point cost of each type of action for each unit type. In some cases, the cost will also depend on the action's arguments.

Acp is actually a little like a bank account, since by not doing anything for awhile, a unit can accumulate extra acp (up to `acp-max`), and it can go into debt temporarily, down to `acp-min` (which may be a negative value). A unit in "action debt" at the beginning of a turn cannot move or do anything else, and must wait for a turn when its acp goes positive again. This can be a simple way to implement both fatigued units and units that can do more if they plan for it.

Actions always include both an actor and an object. The actor is the brains, and that is whose acp gets used up, but the object has the action actually happen to it. This is so animate units (like humans) can manipulate inanimate units (like swords). You enable this by setting the acp of the inanimate to zero, but requiring nonzero acp in the various `acp-to-` tables.

In most cases, the actor and actee are the same unit.

3.10 Movement of Units

Movement is the most important action type. There are actually two distinct types of actions; one to enter a cell, and one to enter a unit.

Each unit has a speed which is determined at the beginning of the turn and determines how many cells it can enter during the turn. However, terrain, borders, and other obstacles can consume extra movement points.

3.10.1 Unit Speed

Units have a base speed `speed` which is the ratio of mp to acp. You can set damaged units to move more slowly. You can also allow occupants to add to the speed, up to the `speed-max` limit.

You can define wind-affected units by defining speed in each direction (max-speed only, do others proportionally). Would need 4 distinct mp costs plus a formula to relate to wind strength. Wind speed defined as "how far a particle of air moves in a turn". Unit examples include balloons, dirigibles, sailing ships, floating cities.

3.10.2 Movement Costs

Typically the cell entry cost will be the most useful to adjust, although the departure cost can be useful in representing units mired in jungle mud and taking a long time to escape onto clear terrain.

Be aware that complicated entry/exit costs are confusing to players, and AIs may not take them into account very well either. Using `free-mp` helps players use up all their acp.

3.10.3 Entering Transports

Different kinds of transports have different ways for units to get on and off. For instance, ships can dock, or use their boats to enable land units to get on and off. The tables `ferry-on-entry` and `ferry-on-departure` specify how much terrain units will have to cross on their own.

[example]

Observe that enter/leave costs can be used to make one-way trips. For instance, paratroops jumping out of a plane should be able to leave cheaply, but have an entry cost so high that they can only reboard in a later turn.

3.10.4 Border Slides

One of the problems with *Xconq* borders and connections is that neither works exactly like a sea strait. Consider the Straits of Gibraltar. They are so narrow that one can see the other side, but nevertheless impose a formidable barrier to landlubbers. At the same time, ships can pass through readily, if not secretly. If cells in the world are 60 miles across, then making an all-sea cell is a gross exaggeration. However, adding a water border only prevents both land and sea movement! To get around all this, *Xconq* allows a special kind of move called a "border slide". Basically, if both the destination cell and the border whose endpoints touch the start and end cells are allowable terrain for a unit, then the unit can move to the destination cell in one move. However, it incurs a special cost in addition to the normal entry and leave costs for the terrain in the two cells (but *not* the border crossing cost, since the border is not being crossed, exactly). This cost is in the table `mp-to-traverse`. Border sliding should usually be somewhat expensive, both because of the distance (the unit ends up two cells away after only one move), and because of the real-life difficulties of passing through a narrow strait. Note that border sliding does not escape the units on either side of the border, since the unit doing the sliding will still be adjacent to the cells on each side of the border it slid through.

3.10.5 Leaving the Area

This feature can be useful in allowing a non-disbandable unit type to escape capture or otherwise retire from action.

3.10.6 Free Moves

This is most useful in emulating some board games, or to prevent clever players from exploiting a mess of move costs. The default of `-1` is the most playable, since player will always be able to use all of their mp. Otherwise, there may be situations in which a unit has a few acp left, but not enough to go anywhere, and so they end up being wasted. The free move does not actually get subtracted from the unit's acp, it just doesn't let lack of acp forbid the move.

3.10.7 Zone of Control

Sometimes a unit can by its presence alone affect the movement of unfriendly units in the vicinity, perhaps by requiring them to hide or to move carefully in order to pass by, or even to prevent entry altogether. This is called the "zone of control" or ZOC.

Exerting a ZOC requires no action, nor any particular capability on on the part of the unit exerting the ZOC. For instance, a toothless fort could still cause raiders to sneak by carefully (at least if they didn't know that it was toothless).

3.11 Unit Construction

Construction is very important to empire-building and similar strategic games. The construction of a unit may involve as many as four different kinds of actions. This is so you can make construction be an expensive long-term process.

The basic construction is unit creation. A player might have to do research and toolup actions in order to prepare for creation, and might also have to do completion actions, if the created unit is not ready to use.

Normally the interface will just have a single `"Build <type>"` command, which then results in a task that issues appropriate actions, so players don't necessarily see all these different actions.

3.11.1 Researching

Some types of units may be relatively easy to build, once you know how, but at the same time that type totally changes the balance of the game. The atomic bomb in WWII is the classic example; once it became available, everything changed.

To allow research, set `acp-to-research` to 1 or more.

3.11.2 Tooling Up

Toolup costs are what you use to represent the overhead of changing construction. Quite often it does not need to be set. Its primary use is to encourage players to commit to grand strategy once chosen, because the cost of changing would be prohibitive.

3.11.3 Creation

You enable creation of new units by setting `acp-to-create` to 1 or more. The location of the newly created unit will depend on both the types involved and how the interface works, since both `create-in` and `create-at` actions are available. For instance, the new unit immediately takes up space, so if creating unit is already full, then the interface should have issued a `create-at` action to put the new unit outside the creator but still stacked in the same cell. If this is still too restrictive, and you want to allow players to create units in nearby cells, you can set `create-range` to values higher than the default of 0.

In order to represent the material costs of creation, you can set a minimum requirement, via `material-to-create`, and an amount to be consumed, via `consumption-on-creation`. You could think of `material-to-create` as representing catalysts or work force, while `consumption-on-creation` is the raw material that becomes part of the new unit.

Finally, you can set the `supply-on-creation` to have *new* material created and given to the new unit. This is useful for abstract materials (such as "enthusiasm") that are somehow ubiquitous. You should be careful with this one, because if the new material is transferrable between units, then players could collect a stockpile of the material by creating units, stealing their supply, and never finishing them.

3.11.4 Completion

By default, newly created units are complete and ready-to-use. This is rarely a good idea in a game design, since even 1 acp-per-turn creators can then create another brand-new unit on each turn. If you're going to allow that, then you should include something else to keep players from being swamped by overpopulation. You can set high accident or attrition rates, make creation require scarce materials, or make the creators be scarce.

The best way to slow down unit creation is to create incomplete units and then require `build` actions to finish them. Completeness is defined in terms of completeness points (cp) that you can set for each type. A build action then just adds to completeness points. Incomplete units do in fact exist as units, so for instance they can be captured and completed by another side.

As with creation, you have to set `acp-to-build` to 1 or more just to enable build actions.

In order to regulate the rate of completion, you have to set the `cp-max` of the unit types being constructed, which defines the point at which the unit will be complete, and then fill in `cp-on-creation` and `cp-per-build`. The most straightforward approach is to set `cp-max` to be the number of turns you want to have between each unit being constructed, then let `cp-on-creation` and `cp-per-build` both be 1.

You can set `build-range` so that several units can cooperate to accelerate construction of a unit. There are no maximum rate limits set on this, but it's unlikely that players will ever be able to achieve much acceleration, because of the limit on the distance between the builder and the unit. For instance, the default range of 0 implies that multiple builders of a unit have to be in the same cell, which may in turn be constrained by stacking limits.

As with creation, you can also set values in `material-to-build` and `consumption-per-build` to govern material requirements and usage.

You can also allow units to complete themselves. For instance, large ships often use part of their soon-to-be crew to help finish the last stages of fitting out. You set this up via `cp-to-self-build` and `cp-per-self-build`. Since incomplete units are incapable of doing any actions, this is a totally automatic process that happens at the beginning of each turn. Self-building and normal building can proceed simultaneously, so you can use this to accelerate the final stages of construction.

Finally, newly completed units can have materials created for them, as defined by `supply-on-creation`.

3.11.5 Repair

Players' units will inevitably become damaged, whether in combat, from accidents, or from other causes.

There are two ways that units recover hp; either automatically, as defined by `hp-recovery`, or by the explicit action `repair`. Automatic recovery is good for that part of damage that a unit can fix just by the passage of time. It's always good for playability, since a player just needs to "rest" the unit in order for it to get better.

On the other hand, the decision to repair may need to be a difficult one, and impact both tactical and strategic planning. For instance, a badly damaged battleship can choose to go on fighting and risk being sunk, or withdraw for repairs and perhaps jeopardize the campaign it is supporting.

In such cases, you can allow explicit repair actions, via the table `acp-to-repair`. You can set the repair rate via `hp-per-repair`. You can also specify how healthy the repairer must be, via `hp-to-repair`. Units can repair themselves.

3.12 Combat Actions

Not all games require fighting. Races and exploration can be lots of fun, and don't require players to be bashing each other. However, the excitement of most *Xconq* games derives from the chances of going up against an opponent directly.

Combat includes five distinct action types that a player may choose from, not counting detonation, and you specify the characteristics of each. "Attack" is hand-to-hand with another unit, "capture" attempts to change the side without damaging, "fire-at" hits a unit without getting entangled, while "fire-into" hits everything in a targeted cell. Finally, "overrun" is an attempt to occupy a cell, doing whatever combination of attack, capture, and movement is necessary.

To specify what kinds of battles are possible, you begin by setting the `hit-chance` of some unit vs another unit to any value greater than zero. A hit probability of zero completely disallows attack. A hit probability of 100 is a guaranteed hit. In practice, you will probably need to specify most hit probabilities individually.

[describe mods to hit prob?]

Next you need to set the damage done by a hit. The default value is 1 hp, which is a good starting place but not always particularly realistic.

[describe variation parms]

As usual, you can define the action point cost of combat, via `acp-to-attack` and `acp-to-defend`. The use of separate tables for attacker and defender allows for some extra flexibility.

This is important, because sometimes you want to allow combat to keep a defender busy and soak up its acp, while at other times attempts to engage in combat should be shrugged off. Consider battleships vs infantry; although combat between the two rarely causes much damage, an attack by a battleship will cause the infantry to keep their heads down, and preventing them from doing much else, while the return rifle fire is unlikely to disturb the battleship much!

Describing simple hit probabilities and damage is oftentimes sufficient for a game. It's simple; players can learn the numbers by heart. It's more efficient, because there's no need to manage lots of ongoing battles. However, there are endless numbers of situations where this basic model is unsatisfactory, so let's move on to the available enhancements.

The basic parameter for the firing actions is `range` of the unit, which is the greatest reach possible. You can also set a `range-min`, which is useful for ballistic missiles, certain kinds of artillery, and magic spells that can't be used for close-in fighting; you can't fire at a unit that is less than `range-min` cells away.

Also, you can define how transports and occupants affect each other in combat. The effects can be both positive and negative, and extend both from occupants to their transport and from the transport to its occupants. The table `transport-protection` defines the percentage of hit damage (by any unit type) that gets passed through to each occupant. If 0, then the transport is perfect protection. If 100, then each occupant gets the same hit as the transport did. [Ideally, protection is a prorating on a table value from occupant vs attacking unit.] Note that an occupant cannot be attacked directly from outside its transport.

If you want to make combat dependent on having a supply of ammo, use the tables `hits-with` and `hit-by`. The material type need not be explicitly designated as ammo, but both the hitting and hit units must agree that the same type is effectual (we assume that the attacking unit is smart enough not to use material types that have no effect on the target unit).

[need a combat-supply usage in addition]

3.12.1 Multi-Round Battles

[Multi-round battles are not yet available.]

3.12.2 Capture

Capture is both a distinct action type and a possible consequence of normal combat. As an action, it is useful for both "bloodless" captures and the collecting of objects from a dungeon floor.

To allow explicit attempts to capture, set `acp-to-capture` to 1 or more.

Whether the capture attempt is explicit or a consequence of combat, its basic probability of success is derived from the table `capture-chance`. If the unit being captured is independent, there is a separate table `independent-capture-chance`; if its value is the default of -1, then the value of `capture-chance` will be used instead.

For capture attempts that are going to succeed, you can allow the victim a chance to wreck itself first, by setting `scuttle-chance`.

The main effect of capture is simply to change the side of the unit that was captured. If the unit cannot be on the capturing side, then it will vanish instead. In any case, the occupants will also be captured or vanish, although you give them a chance to escape first via `occupant-escape-chance`. They will also attempt to scuttle themselves if possible.

You can also require a sacrifice from the capturing unit, via the table `hp-to-garrison`. This is the number of hp that will be taken from the capturing unit. You can set it to the unit's `hp-max` to make it disappear entirely. Although this table is inspired by realism, it can also serve a pragmatic purpose, namely to prevent a single unit from capturing an entire country without being affected at all! You should set this table according to the "feel" you want for the game, since it can have a major effect on speed and pacing of the play.

As with normal combat, the experience of both the capturing and captured unit may change. For the capturing unit, this is a gain defined by `cxp-per-capture`, while the effect on the capturing unit is set by `cxp-on-capture-effect`, which is a multiplier (defaulting to 100) that may increase or decrease experience. In practice, a decrease is more realistic, representing perhaps the replacement of ship or airplane crews, although a increase might be more appropriate for mercenaries whose response to capture is simply to go to work for the new bosses!

3.12.3 Detonation

Detonation is both a type of action `detonate` and an automatic behavior.

Detonation can damage both the detonating unit (though it need not) and any units around its point of detonation, which may or may not be its location. You set it up by defining `acp-to-detonate` to one or more, set `hp-per-detonation` to express the amount of damage done to the detonating unit, then fill in the detonation damage tables `detonation-damage-at` and `detonation-damage-adjacent` to say how badly each type of nearby unit will be hit. You can define the exact radius of effect via `detonation-range`. The effects on occupants of nearby units will be adjusted according to the same protection/ablation tables as for combat.

You can also set detonation to trigger on various kinds of events, such as damage to the detonating unit (`detonate-on-hit`, death of the detonating units (`detonate-on-death`), impending capture (`detonate-on-capture`), and proximity of certain types of units (`detonate-on-approach`). You can also set a chance that a unit will detonate spontaneously, via `detonation-accident-chance`.

In order to model the catastrophic effects of the worst explosives, you can set `terrain-damage` to indicate how terrain types will change.

A minefield could be implemented by defining a detonating unit that loses some small percentage of its hp every time a unit hits it, while hitting the other unit automatically.

A simple trap would auto-detonate only once, then change to a "sprung trap" type. Then the right kind of unit could come along and do a change type action to reset it.

3.13 Unit Manipulation

The actions in this group are a mixed bag of manipulations. If they need to be in your game, then the need will be obvious, otherwise they are pretty much optional.

3.13.1 Transferring Unit Parts

Any unit whose `parts-max` is greater than the default of 1 is a multi-part unit, and its hp denotes size rather than amount of damage. Armies and fleets are two kinds of units which can be usefully defined as multi-part.

Players will very often want to merge or detach parts of a multi-part unit, and there is an action `transfer-part` provided for that. You can control the cost of the action by setting `acp-to-transfer-part`.

3.13.2 Changing Side

Side changing is like capturing, but players can only do it to units that they control. The action is `change-side`, and you enable by setting `acp-to-change-side` to 1 or more. This will also enable side changing for units that cannot normally act.

Side changing is especially useful for alliances in multi-player games, so it should usually be enabled. On the other hand, it should not be too cheap; you should consider what side changing really means in the game's context.

For instance, even in the close British/American alliance during WWII, armies never actually changed sides; British ground units were always British, and American ground units always American. On the other hand, ships and bases could be traded back and forth with only a cost in time and expense.

3.13.3 Changing Type

In some games, it will be useful to have a notion of promotion or upgrade for units. You can implement this by allowing players to do a `change-type` action.

You enable this via the `acp-to-change-type` table.

3.13.4 Disbanding

Sometimes a player will want to get rid of a unit, perhaps because some type has been overproduced and is tying up valuable resources, or to prevent it from falling into enemy hands.

You can allow this by setting `acp-to-disband` to 1 or more.

You can control the rate of disbanding with `hp-per-disband`. You may, for instance, want to allow the deliberate destruction of large units, such as battleships, but you don't necessarily want disbanding to be a convenient way of preventing their capture. Setting `hp-to-disband` so as to require several turns to get rid of a unit will accomplish this. The table `supply-per-disband` will allow you to govern the rate of recovery of the unit's supplies during the disbanding process.

It is also possible to make disbanding a way to recover materials that were consumed in the construction of the unit, by using the table `recycleable-material`. Care should be taken that creation and disbanding of units is not a convenient way to manufacture lots of a material; players *will* use the loophole if it exists!

It should usually not be possible to disband something large like a city, otherwise a clever player might try to eliminate it as a strategic target, but most mobile units should be easily disbanded. This is especially helpful in an "construction spiral" game, where the winning player(s) can accumulate large numbers of useless units.

3.14 Material Manipulation

You can allow players to produce materials by explicit action, and you can control how they transfer materials between units.

Note that you can usually have a reasonable game without requiring all the players to become shipping clerks. The automated production and transfer parameters (see xxx) are almost always sufficient for a game. Explicit action should be limited to games where material limitations are so severe that they impact strategy directly, and players have to make hard choices between producing materials and doing other actions, on a turn-by-turn basis.

You can define "stevedore" units by setting both rate and acp such that the u1 -> stevedore -> u2 transfer is faster and cheaper than the basic u1 -> u2 rate. Then players can use the stevedores to speed up transfers.

3.15 Terrain Manipulation

In a few games, you will want to let players alter the terrain. This needs to be done judiciously, since a cell of terrain generally represents a vast area, and the simulated time in *Xconq* is generally too short for major terraforming operations. However, building bridges and digging moats can be reasonable additions to a game.

Since actions are always completed quickly, and there is no concept of "partly modified terrain", you will probably have to come up with a trick to make terrain modification be slow. One way is make the acp (or material?) cost very high. Another way is to make the alteration happen by removing a material, such as clearcutting a forest, then letting the action make the actual change to clear terrain.

3.16 Vision

Vision is an important part of *Xconq*. Information need not come for free in your game design, and you can design the parameters to control how much players can get. The possibilities range from total knowledge as in board games, where nothing is secret except the enemy's heart, to games where much of the play hinges on who knows what, and when.

3.16.1 Seeing All

The simplest thing to do is to set `see-all` to `true`. Then every player sees all the terrain, everybody's units, everybody's occupants, the whole world and everything in it. This makes *Xconq* like a conventional video or board game, which is sometimes just what you want. Also, since the view matches the world, the game is simpler for players, who need not concern themselves with possibly out-of-date information. Finally, `see-all` is more efficient in time and space, since the general visibility calculations need never be done or recorded. Many games include `see-all` as one of their variants.

You may also find `see-all` to be a useful game debugging aid, since you can watch what is happening everywhere in the world. But, remember that any AIs will most likely adjust their strategy and not bother with patrolling or guesswork about the enemy, and you won't be able to debug the other viewing parameters either!

3.16.2 Coverage

Still, much of the fun in *Xconq* is the potential for surprise. The theory of visibility in *Xconq* is that each side has a layer of coverage, which basically just counts the eyeballs looking at each cell. As your units move around, the coverage in each cell goes up and down. Any cell with a coverage of zero is not currently being viewed by any of the side's units.

The unit property `see-always` is useful for units like towns, which are unlikely to disappear secretly.

These two parameters apply recursively, so for instance a city could be `see-always` and `see-occupants`, while a building in the city is `see-always` and not `see-occupants`, with the net effect that units inside a city can be seen by everybody, but not when they enter a building.

3.16.3 Initial View

The initial view represents the knowledge assumed to have been gathered over the period of time preceding the game. *Xconq* lets you set a radius around each initial unit, within which the side knows everything. Also, any people on your side view both their cell and all the adjacent cells.

`already-seen` should usually be true of things like cities, independently of their `see-always` setting.

3.16.4 Vision Range

The default vision range (`vision-range`) is 1, which basically means that a unit can see into adjacent cells but no further. You can set this to higher values, which is useful for tactical- and person-level games with line-of-sight (LOS) rules [if they ever get implemented].

You can also set the vision range of a unit to 0, which means that it can only see things in its own cell. However, as a special case, when such a unit enters a new cell, *Xconq* will show the terrain of each adjacent cell, but not any units that might be present. This is so players can decide which way to move without having to plunge blindly into unknown terrain or do some sort of awkward "adjacent cell examination" action before moving. This only provides information about terrain and units that are seen if the terrain is seen.

3.17 Backdrop Weather

[The four temperature extremes are independent of each other, so you can make higher latitude temperatures vary drastically with the season, while equatorial temperatures are much more stable; or vice versa.

Average temperature usually varies more slowly over some kinds of terrain than others. For instance, oceanic circulation moderates temperature swings in terrain that is near open ocean.]

3.18 Backdrop Economy

Economy in *Xconq* means pushing materials around. So if you want an economy in your game design, you have to define at least one type of material. To define the economy, you have to decide where materials come from, how they get moved around, and how they get used up.

3.18.1 Creating Materials

Materials come into existence by being placed in units or terrain during setup, by being produced by units or terrain, and by appearing in newly-created units.

3.18.2 Movement of Materials

Once in existence, players can move materials around by explicit action. You can also define automated material movement that uses supply and demand. The tables `in-length` and `out-length` control the distance over which materials will move each turn.

3.18.3 Consuming Materials

Materials exist to be consumed (unless they are relevant to a scorekeeper). You can set how much each kind of action uses, as well as how much is needed as a prerequisite, sort of like a catalyst. You can also set consumption due to existence alone, plus what happens to a unit when its supply of a material runs out.

3.19 Random Events

What simulation game would be complete without random events? Random events are handled somewhat similarly to synthesis methods, in that you set the value of the variable `random-events` to a list of the methods that you want run. Note that you must still ensure that the probabilities for the events on your list are nonzero!

Superficially, random events just introduce some unpredictability into a game. However, adding it just for its own sake is not a good idea; in the worst case, the game becomes the infamous "dice-rolling contest", where nothing matters except luck. Random events are more valuable when they introduce risk, and players have to balance that risk against their goals. As an example, random losses of cities in the standard game would be pointless, since players have to have them, and there would be a chance that all of a player's cities would disappear, causing the player to lose for no good reason at all. On the other hand, the chance of losing an expensive capital ship in shallow coastal waters is enough to motivate the player to keep them well out to sea.

In the past, bugs or unexpected behavior in random event routines have resulted in hard-to-reproduce problems. For the sake of debugging, you should test the game with random event probabilities set very high, perhaps as a variant so it can still be played normally.

3.19.1 Accidents

The name of the accident method is `accidents-in-terrain`. Accidents should be restricted to definite hazardous situations, to go along with movement constraints - for instance, carriers and battleships in shallow water should have a small chance to hit a rock and sink.

You can specify two kinds of accident; a damaging accident, which hits the unit as if it were in combat, or a vanishing accident, in which the unit disapppears instantly.

Damaging accidents occur according to the `accident-hit-chance` table, and damage the unit according to `accident-damage`. The interpretation of these is similar to their combat counterparts. The `accident-vanish-chance` table sets the probability for the unit to simply vanish without a trace.

3.19.2 Attrition

Attrition is a sort of higher-probability/lower-damage type of accident. It is useful for armies in hostile terrain, where deserters and casualties slowly reduce its strength.

Attrition can be useful for "aging" a unit, if you need to keep the unit from being around too long.

3.19.3 Revolts

Revolts are spontaneous changes of side, independent of any other consideration. Since there is no way to protect against this, the chance should usually be very small, less than .01; even a small chance of will cause players to maintain reserves just in case.

3.19.4 Surrenders

The method's name is `units-surrender`; when it runs, it checks each unit to see if it is within `surrender-range` of a unit on an unfriendly side, and if the `surrender-chance` occurs, then the unit will change to the side of the other unit. Occupants will also evaluate their surrender/scuttle/escape chances, and behave accordingly.

3.20 Designing the Interface

So far, the game design machinery has been focused on semantics. The other part of the game design defines how it actually appears to the players. This part of the design can be more loosely designed, which is good, because you cannot guarantee that your game design will only ever be run with a particular interface, and there is a wide variety of interfaces. You could, for instance, define an elaborate set of color graphical icons and patterns, only to find that most of your players only have black-and-white displays. *Xconq* itself will always be able to cope with your omissions, but it will be forced to synthesize inferior substitutes.

Game designs have three general categories of interface elements that they can specify: text, graphics, and animations. Text elements are just strings describing objects and events in a readable form, while graphics consist of small icons and patterns primarily representing units and terrain. Animations are used to illustrate events as they happen, and may include sounds.

3.21 Designing Text

Although *Xconq* is primarily a graphical game system, it is complex enough that the graphics alone are insufficient to describe what is going on.

All text that players see is issued by *text generators*, which are objects that, when given appropriate inputs, produce text fragments that can be used by the interface to produce a textual display. Each text generator has a number of parameters that may be used to select one of several rules [etc]

3.21.1 Describing Objects

3.21.2 Describing Events

3.21.3 Generating Names

One of *Xconq*'s special features is its extensive machinery for generating names of things. You can generate names for sides, units, and geographical features. The possibilities range from a simple list of strings up to context-free grammars and arbitrary code modules. Naming happens throughout the game, as nameable objects are created, but is mostly done during initialization.

3.21.4 Grammar Examples

Here is a very simple grammar:

```
(namer (grammar root 40
  (root (or 1 (the animal in the thing)))
  (animal (or cat dog sheep))
  (thing (or hat umbrella fold))
  ))
```

It makes phrases like `"the cat in the hat"`, `"the dog in the umbrella"`, and `"the sheep in the hat"`.

This example is more realistic:

```
;;; German-like place name generator.

;;; Conventional combos most common, random syllables rare.
;;; Needs more conventional words to combine?
```

```
(namer german-place-names (grammar root 50
  (root (or 95 (name)
             5 ("Bad " name)
             ))
  (name (or 40 (prefix suffix)
            20 (both suffix)
            20 (prefix both)
             5 (prefix both suffix)
            10 (syll suffix)
            10 (prefix syll suffix)
        ))
  (prefix (or
        schwarz blau grun gelb rot roth braun weiss
        wolf neu alt alten salz hoch uber nieder gross klein
        west ost nord sud
        ;; from real names
        frank dussel chem stras mut
        ))
  (suffix (or
        dorf torf heim holz hof burg stedt haus hausen
        bruck brueck bach tal thal furt
        ;; these aren't so great
        ach ingen nitz
        ))
  (both (or
        feld stadt stein see schwein schloss wasser eisen berg
        ))
  ;; Generate random syllables
  (syll (or 40 (startsyll vowel endsyll) 5 (vowel endsyll)))
  (startsyll (or 30 startcons 10 startdiph))
  (startcons (or b k d f g l m n r 5 s 3 t))
  (startdiph (or bl kl fl gl 5 sl 3 sch 2 schl
                 br dr kr fr gr 2 schr 3 tr 2 th 2 thr))
  (vowel (or 6 a ae 2 au 5 e 2 ei 2 ie 6 i 3 o oe 2 u ue))
  (endsyll (or 4 b 5 l 3 n 4 r 4 t
               bs ls ns rs ts 3 ch 3 ck
               lb lck lch lk lz ln lt lth ltz
               rb rck rch rn rt rth rtz
               ss sz 2 th tz
   ))
  ))
```

This generator usually takes normal German words and glues a couple together, making names like `"Schwarzburg"`, `"Nordbruck"`, and `"Bad Salzwasser"`, but it will occasionally make a completely random syllable using common German phonemes, then glue it into a name, resulting in names like `"Biefeld"` and `"Salzgloelthach"`. Yes, that last one is unpronounceable even for Germans, but the generator doesn't know that!

Since there is no special handling to ensure non-garbled names, it generally does not work particularly well to try to build names from vowels and consonants. Either random selection from a list or putting together syllables seems to do better, with perhaps a single totally random syllable thrown in. Don't forget that this is a generator, not a recognizer or parser, so you don't have to be able to handle every possible name; just enough to make an interesting variety.

Recursive rules, where a symbol expands into a sequence mentioning that same symbol, will work, but they are not recommended. Although the generator has a builtin limiter to keep from looping forever, in general there is no way to avoid getting awful names like `"Feldbruckbruckbruck"`. Instead, you can just add extra rules, one for each desired length, so for instance you have a rule for 2-syllable names, one for 3-syllable names, one for 4 syllables, etc. Another advantage is that you can set the probability of each length of name separately, and thus lower the probability of longer names, so that they only appear once in a while and you save the poor players from being continuously tongue-tangled!

3.22 Designing the Graphics

Xconq is fundamentally a graphical game; fortunately, you don't have to do gnarly graphics hacking to get the pretty pictures! The basic graphics handling is built into the interface subroutines of *Xconq*. What you *do* have to do is to choose or design the basic images.

Xconq will always attempt to generate some sort of default display for your new game design, but it's likely to be pretty ugly. So your goal here is just to make the display look good. First off you should decide about the overall appearance. Do you want things to be generally light or dark? Garish or subtle? Conventional or exotic? This is a good time to cruise the image libraries and to look at the graphics of other games. Sometimes the theme decides a lot for you - how could you display anything other than a red star on a Soviet tank? You also need to think about whether you want to concentrate on b/w or color displays, although again *Xconq* will try to do something reasonable for both.

You have to choose three sets of images: terrain patterns or images, unit icons, and side emblems. The terrain patterns have to tile properly, since they may be used to fill in large areas, while both unit icons and side emblems are single icons. You can optionally choose solid colors for terrain, and to "colorize" unit icons and side emblems.

Once you have chosen and specified a set of images, you have to try them out in various combinations in real games. What you'll most likely discover is that they don't always mix like you imagined. That cool-looking emblem for a side disappears against the background of space, or two unit icons are nearly indistinguishable on the map. At this point, you have to start making some choices. Either substitute some different images, or design new ones of your own.

Color choices are tricky. Again, the total effect can be quite different from what you imagined, plus you should be careful about the variety of displays that your game runs on, or you may be getting complaints about how your "olive" more closely resembles "puke gray"!

Here is an example of unit icons:

```
(add (infantry town city) image-name ("soldiers" "town20" "city20"))
```

In general, an icon name should describe the literal appearance of the image, instead of the type that you want it to represent. The `"soldiers"` icon, for instance, just shows a row of soldiers; in one game the icon can be used to represent infantry, in another, armies in general, and in another, the national guard. There is an `"infantry"` image also, but it is the standard "crossed bandoliers" symbol, and is really only sensible for specialized military games.

Here is an example of a terrain pattern:

```
(terrain-type plains
  (color "green") (image-name "plains") (char "+")
  )
```

The `"plains"` is defined in `lib/terrain.imf`, as basically a blank 8x8 tile with two pixels turned on, which textures things somewhat:

```
(imf "plains" ((8 8 tile)
  (color (pixel-size 1) (row-bytes 1)
   (palette (0 7969 46995 5169) (1 0 25775 4528))
   "00/40/00/00/00/04/00/00")
  (mono "00/40/00/00/00/04/00/00")))
```

For extra fine control on color displays, you can also set the colors of unseen terrain and the grid separating cells, via the globals `grid-color` and `unseen-color`.

Note that some display systems (such as the X Window System) allow users to customize most or all of their colors, so individuals may override your choices. Not much you can do about that though!

3.22.1 Image Format

```
(imf "example" ((8 8) (mono "0011223344556677")))
```

[describe when fleshed out]

3.22.2 Image Design Hints

The design of each graphical image can and should be somewhat independent of the basic game design; this allows for reuse of pictures.

The first thing you should do is to check the image library on your machine. The image you're looking for may already be there, but perhaps under a different name. Even if you don't find it, you may notice an image that is close enough to be a good starting point. The *Xconq* image library presently includes hundreds of images, so the chances are pretty good that you'll find something useful.

Designing good images and patterns is a specialized and demanding category of artwork that I'm not going to go into here. My best advice is to learn from the pros, and don't be afraid to experiment.

3.23 Game Module Organization

Each separate file is known as a *game module* or just *module*. A module has a name, displayed name, an advertising-style blurb, a version, and designer notes.

This is an example of an elaborately-declared game module with no actual content:

```
(game-module "foobar"
  (title "Foo of Bar")
  (blurb "An exciting game with lots of cliffhanging suspense")
  (version "1.3")
  (program-version (>= "7.0.3"))
  ;; other properties?
  (complete-game true)
)

;;; contents here
```

```
(game-module (notes (
  "This is just a sample game."
  ""
  "It's not really as interesting as the blurb makes out."
  )))

(game-module (design-notes (
  "This is commentary addressed to other designers."
  "Also a good place to mention things to work on."
  )))
```

The `notes` and `design-notes` could have been supplied with the first `game-module` declaration, but in practice, putting the player and designer notes at the end of the file keeps them out of the way. You can supply any number of `game-module` declarations in a file. Only the first need include a name.

The game module format is only loosely structured. In general, anything that you might want to reuse or combine in different ways should be a separate module. Good candidates include text generators and maps of real terrain. Unfortunately, they don't always mix-and-match as well as you might like!

The following are the generally preferred module names:

Terrain-only modules should be named `t-`*xxx*.

Lists of units should be named `u-`*xxx*.

Name generators should be name `ng-`*xxx*.

When supplying a year in the module name, use four digits, unless the rest of the name makes the century clear (WWII scenarios are pretty much guaranteed to be in the 20th century!).

3.24 Building New Games

There are at least three ways to make a new game design: use *Xconq* commands to "play" a game and then save it, create and text-edit the text files defining a game, or write and run special-purpose programs that create games. A combination of these techniques will likely prove the most useful, since each alone has both strengths and weaknesses. For instance, text editing may seem like a crude approach, but is the only way to produce certain types of scenarios, and text editors have many facilities (such as regular expression replacement) not directly available in *Xconq*. On the other hand, maintenance of the correct transport/occupant relationships between units is hard to do while editing text, but comes for free when using *Xconq* itself.

3.24.1 Building Scenarios

The easiest way to customize *Xconq* is to build a scenario. A scenario is basically a saved game from which irrelevant details, such as the list of players, has been omitted. Typically this will include tweaking details, removing random irrelevant junk, and generally tuning things.

One way to do this would just be to start a normal game, save it, and then dig through the saved game and edit it, since the saved game is itself a game module. Sometimes this is easy, more likely it will be quite hard and error-prone. A better way is available, in the form of "designer mode".

3.24.2 Designer Mode

There are two ways to get into designer mode; one is to start up a game with the appropriate option (`-design` under Unix), which makes every player with a display a designer, the other is to switch on a flag after the game has started. Being a designer is a property of a side, so in theory a game could have a designer and several other human players, or even multiple designers (this might be useful in having assistants to help with the construction of large scenarios, or just to have displays open to each side's view of the scenario). AIs effectively sit out the game while designers are present.

Designer mode enables an additional set of commands on the menu or map control panel, as well as removing some restrictions on the use of normal commands. It also enables more elaborate game saving machinery, so you can save only the parts of a game that you want to make into a scenario.

Modifications to normal commands include the permission to look at and do any command on any unit, including independents and units belonging to other sides. For instance, any unit can be renamed at any time by any designer in the game. The modications include the following:

- Move commands can put any unit at any destination instantly.
- Any unit can be put on any side.

- Any unit can be disbanded instantly.
- Any terrain can changed to any type.

Some interfaces may also provide additional tool palettes and the like.

3.24.3 Saving Scenarios

If you're not in designer mode, then saving the game will save absolutely everything. In designer mode, the interface should ask you what parts of the game you want to save, and what to name the module.

If you don't save everything, then you should start up another game just to confirm that you got what you wanted, *before* shutting down the *Xconq* that you're designing with. Sometimes you won't have saved what you thought you did... It's also a good idea to keep a backup copy of data, especially the indecipherable area layers; use the nesting comments `#| |#` around the old stuff, only delete when you're sure it's no longer of interest.

3.24.4 Conversion from Xconq 5

There are many scenarios extant from the version 5 of *Xconq*. Many of them are good games despite some of the quirks of version 5 that they had to work around. Converting these scenarios to the new GDL syntax should provide some great new modules and at any rate provide a goldmine of ideas for updated *Xconq* game modules.

A set of conversion scripts are provided that will help to ease the transition from version 5 to version 7, but they won't save you from learning the new GDL syntax or features. These scripts will NOT generate working games modules, but they will generate valid GDL syntax, and thereby spare you much tedium in conversion.

The first thing to consider is the naming of the files/modules. There are already some loose guidelines for naming version 7 game modules (see Section 3.23 [Game Module Organization], page 104). Terrain or worlds should be in modules named `t-xxx.g`. These are roughly equivalent to version 5 `.map` files. Collections of units, such as the cities to populate world maps, should be in files named `u-xxx.g`, where `xxx` generally identifies which map they go with in addition to a general identifier (e.g. `1942`). Name generators are in files of the form `ng-xxx.g`, but you probably don't know or care about these yet. And finally, if you are building a set of scenarios based on a core set of rules, you should consider a naming scheme that will link them all together so that players can find them easily.

Having said all that, let's get on to the conversion. The conversion scripts go somewhat blindly on the assumption that you've split everything up in the "standard" way. That is, assuming that you've got a spiffy big scenario, that it comes in three parts: a period definition, a map and a scenario file. If not, if you've *dared* to combine some of these files, you should split them manually before starting the automated part of the conversion.

Convert the map using `map2g`. You want to use the -o option and your new t-something name and the -b with a full pathname to the period file that has the terrain type definitions in it. This allows `map2g` to set the default base module and the get the appropriate character list for creating the map file. The generated world will have its circumference set to match the width of the generated area, i.e. it will wrap from side to side. This is because all maps are cylindrical in version 5.

Next, do a pass over the `.scn` file with `scn2g`. Again you should use -o to get the naming the way you want it. This should leave you with a very pretty set of units and a very rough hack at a set of victory conditions (i.e. scorekeepers). The scorekeepers will need to be completely reworked, since they work rather differently in version 7.

Now the home stretch, convert the `.per` file with `per2g`. Keep an eye on the output. If it complains about "unknown keywords" then you've probably used one of the more obscure features of version 5. Don't panic because your obscurity will be preserved–commented out–in the resulting game module. Now you have to edit the module and start sorting out the bits that `per2g` couldn't handle. Search for occurances of FIX. These are lines inserted by `per2g` to note places that need your attention. `per2g` may have done nothing to the line except comment it out, or it may have done a partial (or partially correct) conversion, or it may have done a complete and valid conversion but wishes to call your attention to related forms that can be added.

For this process you are going to need to have the documentation close at hand to make sure you get the syntax right. The best thing to do is read thru this chapter of the manual and then have the Reference Manual chapter on hand while editing the module.

Generally the place to start will be the `make` and `maker` lines from the old period definition. These are not converted at all by `per2g` (because the machinery has changed so radically in version 7), but are often essential to being able to start up a game. From there you can work your way through the rest of the file with frequent references to the manual and occasional test runs. Check out the debugging tips in this chapter.

3.24.5 Preparing a Game for Use

Once you've constructed a game, you should bring it to a state where it can be given to other *Xconq* players. I recommend copying a standard software release strategy. This means documenting how to play the game, documenting how it works internally, removing unused junk and dubious features, simplifying where possible, resolving open issues if possible, documenting them as known problems if not. This gets you to the point of having an "alpha" or "beta" version (the terms are not precise!). These can be given to other people for testing, but should be clearly identified as test versions, because your testers may pass copies along to others without you knowing about it. After some playtesting (see below), edit your game into its final form, call it 1.0 and release it to the world!

After you release your game, you may get some feedback about unanticipated problems. When you resolve these, and want to make a new release, be sure to give it a distinct version number. This will be important to deciding whether subsequent complaints are about your new release or some older one. If you always put the version number into the `version` property of the `game-module` form, then it will be displayed to players when they ask for help on the game.

3.24.6 Installing Scenarios

Once the scenario is constructed and saved, you can install it in the library and otherwise do as you like with it. See the interface documents for platform-specific installation details; in general, just copying the files into the `lib` directory will suffice.

3.24.7 Safety

While generally safe – *Xconq* shouldn't crash while you are designing nor upon starting up your scenario – you can do silly things, like loading a submarine with battleships as passengers. *Xconq* won't complain, but it may behave very strangely. For instance, a unit might be able to travel with a transport and leave it, but not be able to get back on again.

One way to test a game is to remove all the scorekeepers and make all the players be AI-controlled. The AI code will then act totally randomly, thus exercising parts of your design that you may not have thought much about. A convenient way to try out various scorekeepers is to put them in variants, then select them upon startup.

3.24.8 Balance and Playtesting

Scenario design can involve subtle questions of balance which will only be revealed by repeated play of the scenario. Playtesting is extremely important, even for simple scenarios! You should try as many combinations of startup options as possible - for instance, the combo of two humans and one machine might reveal a peculiarity that is not observed in a two-person game. You can solve many problems by adding more restrictions. Since the scenario is your concept, you are free to make whatever decisions are necessary to realize that concept; if somebody complains, they are free to make their own designs.

Playtesting is also the time when you may have to sacrifice realism and favorite theories for playability. Listen to and watch yourself and your testers as the game is played. For instance, you might have included a city out in the boonies, but in the game it never does anybody much good, while still requiring some amount of attention regularly. Lose it.

Game startup can be confusing to players if they all start out with lots of units needing to be told what to do. One solution is to put most units on automatic behaviors that expire in a turn or two, so that novices gradually hear from all the units, while experts can still override right from the outset. Another approach is to make units independent and allow them to be captured early on. Still another approach is to make units come in as reinforcements at preset times and locations.

Although as many of the game parameters as possible are checked, there is plenty of room for subtle loopholes. You should think carefully about the consequences of each parameter, being particularly sensitive to degenerate winning strategies. Most common are units that are too powerful, too fast, or are built so quickly that they overwhelm any opposition. Players should always be a little "hungry"; not able to get quite as many units or as much material as they would really like.

3.24.9 Complexity

Although GDL is a powerful language, you should avoid designing a game that is too complex to be humanly playable. A single game can literally define millions of different parameters, each with a range including 100 to 10,000 distinct values. It is clearly possible to spend many years exploring just a single set of these! For more playable and enjoyable games, either pick a single thing to treat in detail, or else do everything in a simplified way. For instance, if you want elaborate movement and combat rules, avoid or even eliminate materials and associated material handling rules.

Another thing to keep in mind is that the introduction of a new type may have far-reaching consequences – for instance, a new unit type will need its interactions with *all* other unit types defined. One approach is to introduce a new type as a slight modification of an existing type, then to share most of the definitions. Another thing you can do is to put complexity into the variants, so players with a taste for punishment can indulge themselves, while leaving the basic game as more of a fun thing.

3.24.10 Combinations

Many of the 700-plus game parameters were chosen for their ability to combine in interesting ways, rather than for obvious usefulness. For instance, construction in a city can by default generate an infinite stream of units. But suppose you want to put a limit on the numbers of that type of unit? One way is to define a material that is essential for construction of that type, let the builder have an initial supply, but provide no way to get more of that material. When it runs out, no more units!

Another trick is to motivate an activity by making it a prerequisite to the basic builtin goal of defeating the other player. The age of discovery worked this way. The kings of that time weren't interested in new lands per se; they wanted exploitable possessions that could be used to get gold to buy armies big enough to defeat their neighbors. You could describe this situation almost exactly, by making gold a material, obtainable only by the discovery and capture of independent gold mine units, which are thinly scattered over the world and can be found only by careful exploration.

Be inventive! Studying the predefined games should suggest many tricks; the "Problems and Solutions" section below describes even more. Be sure to document the trick carefully, or the next time you work on the game, you might break it, resulting in unhappy players wondering why their usual strategies don't work anymore.

3.25 Debugging

Completely new game designs usually have a number of bugs. There are several stages of trouble that you may encounter. First, the *Xconq* may fail to read a game module completely. It will try to report what happened, but if for instance you left out a closing parenthesis, you may get some strange error messages. This is just plain old syntax error trouble.

Once you've successfuly read in your new game, bring up the online help and scan through to see if the values present are what you thought. Sometimes the reader does not interpret a module in the way you thought it would. The `print` form is useful for debugging at this point; it can show you whether a defined symbol has the value you thought it did.

However, the most serious problems with games are play balance issues. Some can be found out by watching a machine player attached to a display, since its decisions are based on perceived values of the units. The most subtle bugs can only be uncovered by extensive play interspersed with judicious alteration of parameters. I find it useful to play for a while, then review and adjust the game parameters all at once, thus avoiding tweaking one parameter only to find that it results in another being inconsistent. Parameters interact in many ways - you should keep this in mind when experimenting.

Something else to keep in mind at this point is that playability should outweigh realism. For example, real-life airplanes can travel 1,000 times faster than a person walking on the ground, but airplanes that could move 1,000 cells in a turn would be ridiculous (try it out, *Xconq* will let you do this!).

3.26 Problems and Solutions

This section discusses specific kinds of design problems and ways that you might solve them in *Xconq*. These are merely suggestions; in the past, game designers have come up with all sorts of ingenious ideas. If you come up with one yourself, please pass it along!

3.26.1 Limiting Unit Quantities

In some cases you may want to constrain the total number of units in play, perhaps because of performance reasons, or because some type tends to proliferate more than is desirable, or because your game concept requires a hard limit on the number of units. You have several ways to do this.

Xconq does give you several parameters that put a simple cap on total numbers, either by unit type or for all units, and per side or for all sides together. You can also define a material type that is essential to the creation, completion, or operation of units, and make that material be hard to come by. Iron to make ships, gold to pay armies, or food to feed armies could all work this way. If the only source of the limiting material is an initial supply in a starting unit, then this is a hard limit; if production of the limiting material is slow, then the limit is softer but still very real.

Limits on unit quantities have some interesting uses beyond the obvious ones. For instance, a useful type that is limited to at most a single instance could be a sort of "football" where the side that has the one unit finds itself being chased after by all the other sides trying to get it. You could make a WWII-era game with "Oppenheimer" as the only scientist who knows how to make an atomic bomb (I know, it's not realistic), and have the different sides trying to kidnap him.

3.26.2 Handicapping

Very rarely will the *Xconq* players in a game all be at the same skill level. Sometimes this is OK, since weaker players really do learn more from their losses than their wins. However, when the goal is to have fun, or when the difference in abilities is extreme, you can balance things out in several different ways.

One simple approach is just to design an imbalanced scenario, document it as such, and let players choose the stronger and weaker sides as desired. In many cases this should be sufficient; for instance, accurate historical simulations.

The next most simple solution is to set up sides or side classes and fill random properties differently. Weaker players could choose a side with more technology or whose class allows more powerful units. This isn't very adjustable, since all the sides and their property values have to be predefined.

To enable the most precise match of player abilities, you can use the `initial-advantage` property of player objects. This property is a relative value, defaulting to 1, and indicates how strong the initial unit setup should be relative to the other players. For instance, if a three-player game includes advantages of 2/3/7, then the second player will have three units for each two of the first player while the third player (the weakest) will have seven. The implementation of relative advantages is up to game synthesis, so for example the `make-countries` will adjust all the numbers of initial units to match the requested advantages. Note that this affects only the initial setup, and only certain synthesis methods. Once a game has started, all sides are always on an equal footing.

3.26.3 Buying the Initial Setup

A common form of game setup is to give each player a quantity of "money" of some sort, then give them a menu from which to buy things. The way you would implement this in *Xconq* is similar to the method for limiting unit quantities - make the money be an initial supply of a special material type not used for any other purpose. This initial supply should be given to a first unit that each player starts with. This first unit could be something like the adventurer in a fantasy game who starts with a pot of money, so the first unit is also the most important one, or perhaps a little dummy unit that buys the other units and then is of little interest thereafter, sort of like the national bank for the player's country.

Here's an example:

```
(unit-type adventurer
  (start-with 1)
  )
(unit-type shop
  (start-with 1)
  )

(unit-type sword)
(unit-type armor)
(unit-type boat)

(material-type money)

(table initial-supply (adventurer money 200))

(table acp-to-create (shop (sword armor boat) 1))

(table material-to-create ((sword armor boat) money (20 100 1000)))
```

The shop can't do anything besides create items when given money. The adventurer starts with the money and has to give it to his/her shop, then order the shop to create the items desired. The shop will create completed items instantly, ready for the adventurer to use.

Note that this can't be extended to buy extra intrinsic qualities, such as hit points or action points.

3.26.4 Leaders

Some games, particularly wargames set in Napoleonic times or earlier, feature the concept of a "leader" as the sole individual who can make things happen. Without a general or field marshal, the army won't move. Whether or not this is truly realistic, it does have the effect of focusing the game on key individuals!

One way to do this is to make the leader be a self-unit and limit the distance of direct control over other unit types. Another way is give armies 0 acp and allow leaders to push them around, and still another way is to use leaders as occupants that add to an army's speed.

3.26.5 Navigable Rivers

The concept of a navigable unbridged river is a real problem for *Xconq*. Non-navigable rivers are easily done as border terrain, and navigable rivers with lots of bridges can be connections (since by their nature, connections can never prevent movement). But a navigable river that can't be crossed easily is more of a problem. One way is to make a chain of adjacent cells of a water terrain type. However, this can be quite unrealistic if cells represent large areas, say 10-100 km across; you can end up with continents consisting of more river than land. In some cases, you can define a "river valley" terrain type where both vessels and ground units can exist, with the river border terrain along just one edge of the valley.

You can also allow border sliding. Border sliding allows a ship to pass along the length of a border, but it does require the ship to be in compatible terrain at both ends of the border. So define the river as a chain of alternating water cells and water borders connecting them together. Then the river acts as a barrier to units wanting to cross, while allowing them to see over to the other side, and at the same time ships can pass up and down the river freely (modulo any ZOC exerted by units on either side).

3.26.6 What Ranges for Values?

One of the problems that you encounter when defining a lot of interrelated units with lots of properties and tables is to decide where to start out with the numbers. There are a couple ways to get started.

First, you can start from real-world numbers. Let's say your game concept is based on turns that last about one day, and you want to use worlds with cells that are about 10 miles across. Now a person in good shape can walk about 2 miles per hour, or 20 miles in a day, which comes out to 2 cells/turn as `acp-per-turn` for units on foot. This allows a speed of 1 cell/turn for injured, tired, or overburdened persons, via the various speed modifiers. However, if this same game includes automobiles and airplanes, then using the same calculation, we get automobiles that can move 60 cells/turn and airplanes that can move 600 cells/turn! The massive disparity in speeds makes for poor playing; every turn each airplane will make 300 moves while the foot traveller makes 1. To make the game work, you'd have to make airplanes slower (they have to refuel a lot perhaps) or make people faster (nobody walks anywhere anymore). So the real-world numbers approach isn't foolproof.

Another way to go is to start with the smallest values and work up. For instance, in the monster game above, you could assume that the mob moves the slowest, and give it a speed of 1. Then you say that the national guard should be able to move twice as fast, and give it a speed of 2. Then the monster should be able to chase and catch mobs and guards that run away, so you give it a speed of 3 or more. This approach is more painstaking, particularly when lots of numbers are involved.

You can use both approaches together as well, working with real-world numbers until they get too weird, then adjust to make relative values sensible, then do some more real-world calculations. As always, only playtesting is the final arbiter. Once the numbers "feel" right in a game, only the obsessive-compulsives will care about their exact values.

3.26.7 Fatigue

Players are often unmerciful to their units, moving them nonstop, going into battle after battle, never a thought for how tired the poor units might be. Although *Xconq* does not include fatigue as a basic concept, it does have several ways to implement the effects of fatigue.

One way is to use acp debt. If you allow the acp to go negative during a turn, then the player can work the unit really hard for one turn, then it has to rest until its acp builds up to positive levels again. While acp is negative, the unit can take no action on its own. Over a period of time, the effect is that of a unit that can only do so much, but can exert itself when needed.

Another way to do fatigue is via a material type, perhaps called "energy" or "enthusiasm". As an abstract sort of material, don't let energy be passed around (unless you want to have "infectious enthusiasm", might be useful sometimes for leaders and morale builders). Units need energy in order to move, and can consume energy faster than they produce. For instance, if a unit has a speed of 3 hexes/turn, consumes 2 units of energy per move, and only produces 4 units of energy each turn, then on the average the unit will only be able to move 2 hexes in each turn, although if it saves up energy, then it can move the full 3 hexes. Since different kinds of terrain can have differing productivity, you can also make some kinds of terrain be more tiring than others. A resort hotel unit could also be allowed to transfer energy to its residents, restoring them faster than a Motel 6.

3.26.8 Brainless Units and Scorekeeping

One special case to watch out for occurs in games with "unintelligent" units, that is, they have an acp of 0. If a side loses all of its units except for the unintelligent ones, the player will not be able to do anything except wait for the game to end. This might be OK, for instance if the idea of the game allows for a side to own a particular unit, whether or not it can do anything with it (perhaps the unit is a fort, and a side can win if it owns the fort, even at the cost of all its other units). Usually, however, the side ought to just lose, in which case you will need to define a special scorekeeper that requires each side to have at least one of some sort of unit with acp > 0, or else it loses.

3.26.9 Days and Years

[should go elsewhere]

The *Xconq* world can be made to revolve around its sun and to rotate on its axis. [etc]

To get a realistic hour-by-hour simulation, say

```
(world
  (day-length 24)
  (year-length 8766) ; this is 365.25 days
  )
```

3.26.10 Xconq 5.x Setproduct

Xconq version 5 had a sometimes-useful flag called "setproduct" that could be set to false, with the effect that any attempts to *change* construction were disabled. So for instance, a city that was set by a scenario to build bombers would then build bombers throughout the game. The advantages were both in realism (retooling a factory can be very time-consuming) and in playability (no construction planning required).

To emulate this in version 7, you can set `acp-to-toolup` to be zero for cities, but at the same time require 1 tp for each type that the city can construct. In the scenario, set the value of the city's tooling to be 1 for the one or more types that you want it to specialize in (maybe switching between fighters and bombers should be possible, but not to submarines). Players can then start and stop construction as desired, but are limited to only particular types. Even captured independent cities can be limited in what they can be used to construct.

3.27 Optimization

The `add` form is very powerful and very useful for making groups of objects share some data. The grouping also helps the designer to see how sets of numbers compare to each other. In other words, instead of having multiple forms:

```
(unit-type foo
  ...
  (acp-per-turn 3)
  ...)
```

```
(unit-type bar
  ...
  (acp-per-turn 49)
  ...)
(unit-type baz
  ...
  (acp-per-turn 2)
  ...)
```

you can say

```
(add (foo bar baz) acp-per-turn (3 49 2))
```

to get the same effect.

To get an inheritance-like effect, you can append lists of types together, as in

```
(define mammal (dog cat cow))
(define bird (hawk eagle condor))
(define animal (append mammal bird fishie))
```

which results in a list of seven types. It is possible to append different kinds of objects together.

3.28 Miscellaneous Tricks and Techniques

An unwanted unit in a shared library file could be gotten rid of by matching on id or name and then setting hp to 0; `(unit "Corinth" (hp 0))`, for instance, would eliminate Corinth from an ancient Greek game.

Elevation data, while interesting to include, can take up a lot of space and be more detailed than necessary. The parameters here allow you to restrict elevations to a smaller range of values, which will allow for more compact encoding and simpler games. For instance, a game set in rolling countryside doesn't need a huge range of elevations; you could set elevations to range from 0 to 300 meters, in 30-meter increments. Then only 4 bits will be needed to encode each value, and yet the player will still see reasonable values like "150 meters", and formulas for temperature and other elevation dependent data will be correct.

Note that just because a player controls a side doesn't mean that the controlled side can be taken out of the game; for one thing, certain types of units will not change sides under any circumstances.

People materials should usually not be directly movable between units.

ZOC should be less than combat range usually, since it means that exerter should be able to control ground (but could attack further in multiple turns). ZOC levels should be only those reachable by the unit.

With all the costs of moving around, it may be that a unit has movement points left, but not enough to meet the full cost of a desired move action. You can allow player extra movement points to complete the action by setting `free-mp` to effectively add the needed mp.

A hit on a complete unit should reduce by whole cp/hp, otherwise it will appear to be incomplete. *Xconq* will not fix this, you have to arrange all the numbers yourself, or run the risk of player confusion.

Bases should "anti-protect" aircraft in games involving both, but fighters should protect the base.

Veason temp values of 40, 20, 5, -40 make earthlike.

4 Reference Manual

This manual is the complete description of GDL. The style is somewhat terse; for more detail on how to use GDL to design games, see Chapter 3.

Please note that the current version of Xconq may not fully implement all of the constructs or combinations of constructs described here. Any such omissions should be regarded as bugs.

4.1 Language Syntax

GDL resembles Lisp, but instead of defining functions, the contents of a file declare certain objects (such as units and unit types) to exist, and specify values for their properties. In other words, GDL is *nonprocedural*. This means that most of the time, you can list the various forms in any order you like. The main restriction is that any symbol, such as a variable or the name of a type, must be defined before it is used. Also, forms such as `set` and `add`, that set the value of a variable or property, always overwrite the previous data irreversibly, so ordering of these is very important.

4.1.1 Lexical Elements

Numbers are introduced by a decimal digit, plus, or minus signs. They may contain only decimal digits, a decimal point, and be followed (immediately, no whitespace allowed) by a percent sign or a recognized unit of measure.

Strings are sequences of characters enclosed by doublequotes ("). They may contain any character except ASCII NUL (`'\0'`). To include a doublequote, use backslash, as in `"a \"quoted\" string"`. To include a nonprinting or eight-bit character, use backslash followed by three octal digits, which will be interpreted as an eight-bit character code. (This is mostly the same syntax as in C.) Note that game design files may be passed over networks and between different kinds of computer systems, so non-ASCII characters should not be inserted verbatim into strings.

Symbols are sequences of characters that don't include any of the other special characters. If you wish to include such characters in a symbol, enclose it in vertical bars, for example `|foo bar|`. (The bars are not part of the symbol.) Symbols are case-sensitive, but this will be changed eventually.

Lists are a sequence of expressions enclosed in parentheses. The empty list is either `nil` or `()`. "Dotted pairs" are not allowed. Anything that is not a list is an *atom*.

All of these objects may range up to a very large size. (You may still run into bugs if you make strings or symbols over about 100 chars in length.)

Comments are enclosed either within `#| |#` (which nests properly, like Common Lisp and unlike C), or else extend from a semicolon `;` to the end of the line. A comment is equivalent to whitespace, so `(a#|bcd|#e)` is the same as `(a e)`, not `(ae)`.

`#` by itself is a normal token.

True/false values are just the integers 0 and 1, with no special characteristics.

`true` GlobalConstant

`false` GlobalConstant

These constants are symbolic forms for `1` and `0`. They are identical to numbers, but more descriptive for parameters that are boolean-valued.

Unit, material, and terrain types are distinct objects. However, they can be considered to have numeric "indices" assigned in order of the types' definition. These numbers are not directly visible in GDL, but they often affect sorting and ordering.

4.1.2 Conventions Used

Descriptions of values in this manual follow the conventions listed here.

For parameters described as *t/f*, both `1`, `0` and `true`, `false` may be used. Parameters described as *n* and *n%* are numbers. Parameters described as *dist* or *length* are also numbers, but are in the unit of measure for lengths. Parameters described as *str* or *string* are strings.

Parameters described as *u* or *ui*, *m* or *mi*, and *t* or *ti*, are values that must be unit, material, or terrain types, respectively.

Parameters described as *utype-value-list* match unit types with values. They can have several forms:

`(n1 n2 ...)` matches `n1` with type 0, etc in order.

`((u1 n1) (u2 n2) ...)` evaluates `u1` to get a unit type, then matches it with `n1`. `u1` etc may also be a list of types, in which case all the types get matched with `n1`.

Other types of lists, such as those defined as *side-value-list*, are interpreted similarly. For all of these, multiple assignments to the same type etc will overwrite quietly.

4.1.3 Forms and Evaluation

A *form* is either any single expression that appears in the file. A GDL file consists of a sequence of forms. Most forms of interest will be lists whose first element is a symbol identifying the form. For instance, a form beginning with the symbol `side` declares a side object. When the file containing such a form is read, *Xconq* will create a side object and fill in any properties as specified by the form. (Properties are like properties or attributes - most GDL objects have some.)

In most contexts, *Xconq* will *evaluate* an expression before using it, such as when filling in an object's property. Numbers and strings evaluate to themselves, while symbols evaluate to their bindings, as set by `set` or `define`. Lists evaluate to a list of the same length, but with all the elements evaluated, unless the first element of the list is a function. In that case, the remaining elements of the list are evaluated and given to the function, and its result will be the result.

4.1.4 Tables

A *table* is a two-dimensional array of values indexed by types. Indices can be any pair of unit, material, or terrain type. The set of tables is fixed by *Xconq*, and all are described below.

`table` *table-name items...* Form

This is the general form to fill in a table. The table named by *table-name* is filled in from the *items*. If an item is an atom, then every position in the table is filled in with that item, overwriting any previously-specified values. If an item is a list, it must be a three-element list of the form `(`*type1 type2 value*`)`. If both *type1* and *type2* are single types, then *value* will be put into the table at the position indexed by the two types. If one of *type1* or *type2* evaluates to a list, *Xconq* will iterate over all members of the list while keeping the other type constant, while if both *type1* and *type2* are lists, then *Xconq* will iterate over all pairs from the two lists. The values used during iteration depend on whether the *value* is a list. If *value* is an atom, then that value will just be used on every iteration. If a list, then *Xconq* will use successive elements of the list while iterating.

If the first member of *items* is the symbol `add`, then the rest of the items will add to the existing contents of the table rather than clearing to its default value first.

The following forms are all equivalent:

```
(table foo (a y 1) (b y 2) (c y 3) (a z 9) (b z 9) (c z 9))

(table foo ((a b c) y (1 2 3)) ((a b c) (z) 9))

(define v1 (a b c))
(table foo (v1 y (1 2 3)) (v1 z 9))

(table foo ((a b c) (y z) ((1 2 3) (9 9 9))))

(table foo (a y 1) (b y 2) (c y 3))
(table foo add ((a b c) z 9))
```

4.1.5 Modifying Objects

Since forms normally define or create new objects, GDL defines the `add` form to modify existing objects.

`add` *objects property new-values...* Form

This form evaluates the atom or list *objects* to arrive at the set of objects to be modified. Then it uses the *new-values* to write new data into the property named *property* of those objects. The *new-values* may be a single number or string, or a list.

4.1.6 Symbols

Most of the symbols used in a game module are the predefined ones described in this manual. Others are attached to types when the types are defined, and still others name objects like units and sides. You can also define and set your own symbols to arbitrary values.

`define` *symbol value* Form

This form defines the symbol *symbol* to be bound to the result of evaluating *value*. If *symbol* is already defined, *Xconq* will issue a warning, and ignore this form.

`set` *symbol value* Form

This form rebinds the already-bound symbol *symbol* to be bound to the result of evaluating *value*. If *symbol* is *not* bound already, then *Xconq* will issue a warning, but proceed anyway.

`undefine` *symbol* Form

This form destroys any binding of the *symbol*. This is allowed for any symbol, including already-unbound symbols.

4.1.7 Lists

`quote` *xxx...* Function

This function prevents any evaluation of *xxx*. (This implies that the abovementioned evaluation of the argument list does *not* happen for this "function".)

`list` *xxx...* Function

This function makes a list out of all the *xxx*.

`append` *xxx...* Function

This function appends all the *xxx* (which may be lists or not) into a single list. Non-lists will appear as though they were single-element lists.

`remove` *list1 list2* Function

This function removes the members of *list1* from *list2*, returning the result.

4.2 Game Modules

The game module declaration supplies information about the file as a whole. It is optional; if missing, *Xconq* will get the module's name from its file name, and supply defaults for the other properties.

`game-module` [*name*] *properties...* Form

This form defines the properties of this game module. The optional *name* is a string that will be used to look up the module in libraries. If the *name* is supplied, then this form is considered to be the definition of the module, and overwrites any `game-module` form previously appearing in this file. If *name* is missing, then this form will modify the existing description of the module.

`title` *string* ModuleProperty

If defined, this property is the name by which the module will be displayed to players. It is not used internally, so the name can be modified freely (unlike the module's name, which may appear in other modules). Defaults to the module's name.

`blurb` *string* ModuleProperty

This property is a one-line description that users will see when they are deciding whether to play the module. It will be displayed without any modification:

```
Welcome to my nightmare! (version 1.0 with stronger goblins)
```

Defaults to `""`.

`picture-name` *string* ModuleProperty

This property is the name of a picture that may be displayed along with the module's blurb, by those interfaces that support such pictures. Defaults to `""`.

`base-game` *t/f* ModuleProperty

`instructions` *strings...* ModuleProperty

This property is a list of strings that are the instructions on how to play the game. Defaults to `()`.

`notes` *strings...* ModuleProperty

This property is a list of strings comprising the set of detailed player's notes for the module. Both the list and each string in the list can be of any length. When displayed, the strings are all concatenated together, so the division into strings here is just for convenience. How these are displayed is up to the interface, but in general an empty string signals a new paragraph. Defaults to `()`.

`design-notes` *strings...* ModuleProperty

This property is a list of strings that are notes addressed to game designers. Defaults to `()`.

`version` *string* ModuleProperty

This property is the version of the module. Defaults to `""`, which indicates that the module's version is undefined.

`program-version` *versions* ModuleProperty

This property dentifies *Xconq* versions for which this module is appropriate. If specified, then players will get a warning if they attempt to use this module with an inappropriate version of *Xconq*. Possible forms include a string, which allows the module only for exactly matching version of *Xconq*, and `(`*comparison version*`)`, which allows versions satisfying the *comparison* test, which may only be `>=` or `<=`. So for instance

```
(game-module "foo" (program-version (>= "7.0.3")))
```

is claimed to only work for versions 7.0.3 or later. Defaults to `""`, which means that the module is appropriate for any version of *Xconq*.

Notes that the `program-version` is strictly a heuristic to forewarn players; in practice it can be very difficult to know which modules work with which programs. (The problems are similar to those encountered by programmers using different compiler versions on their programs.)

`base-module` *name* ModuleProperty

This property is the name of a module that must be loaded first. It is similar in effect to `include`.

`default-base-module` *name* ModuleProperty

This property specifies the name of a module that will be loaded if this module is given as the "top-level" module, such as via `-g` on a command line.

This is to prevent disasters when a library module that is used only by other modules is instead loaded as if it were a full game design.

4.2.1 Variants

Variants are options chosen by players at the start of a game. A generic variant includes information that will be used for displaying the choice to players, the acceptable range of choices, a default choice, and additional forms that may be evaluated if particular values were chosen. Variant values are always numbers.

`variants` *items...* ModuleProperty

This property defines named variants on this module. Variants appear as startup options for the game. The items have the form `([`*name*`]` *type* `[`*range/default*`]` `[`*clauses*`])`. The *name* is a string or symbol used to identify the choice to the players, the *type* says what sort of change is being enabled, *range/default* supplies a range of values and a default value among them, and *clauses* is a list of the form `(`*value forms*`. . .)`. A game module may specify any number of variants. Defaults to `()`.

A number of commonly useful variant types are predefined.

`world-size` [*width* [*height* [*circumf* [*lat* [*lon*]]]]] [*clauses*] VariantType

This variant allows players to choose the size of the world. The sizes will default to the values in this variant's data. (*width* and *height* can be lists of the form `(lo dflt hi)`, with the obvious interpretation??)

`world-seen` [*dflt*] [*clauses*] VariantType

This variant allows players to choose whether the terrain of the world will be known at the start of the game. The default setting will be the value `dflt`, which may be either `true` or `false`.

`see-all` [*dflt*] [*clauses*] VariantType

This variant allows players to choose whether everything will be seen always, as with the global variable `see-all`. The default is set by `dflt`.

`sequential` [*dflt*] [*clauses*] VariantType

This variant allows players to choose whether to move simultaneously during a turn, or one at a time. The default is set by *dflt*.

`real-time` [*total* [*perside* [*perturn*]]] [*clauses*] VariantType

This variant allows players to choose realtime limits on the game. The value will default to the values in this variant's data.

4.2.2 Including Other Modules

You can include one game module in another.

`include` [*if-needed*] *module-name* [*variant-settings*] Form

This form has the effect of inserting the contents of *module-name* into the current position in the module. `game-module` forms in the included module are not inserted, although they are remembered and may appear in displays. *Xconq* will fail completely if the included module cannot be found.

Unlike C etc, the same module cannot be included more than once; you will get a warning and the module will not be loaded.

Note that the module names are not file names, so that system-specific features like directories and devices cannot be included. The mapping between module name and file name is interface-specific, so if you want to distribute a module, you should make sure all the module names don't have anything nonportable embedded. Alphanumeric characters and hyphens are guaranteed to be portable.

4.2.3 Conditional Loading

You can control which forms in a module are actually evaluated by using conditional loading.

`if` *test-form sym* Form

`else` *sym* Form

`end-if` *sym* Form

If *test-form* evaluates to `true`, then all subsequent forms, up until the matching `else` or `end-if`, will be evaluated. If `false`, then the forms will be read but not evaluated. All forms inside the conditional must be syntactically correct.

4.3 The World

The world consists of one *area*, which is regular in shape and consists of a number of *cells*. Each cell has a type of terrain and a number of optional data values. Each kind of per-cell data will be called a *layer* of the area.

`world` [*circumference*] *properties...* Form

This form defines the properties of the world as a whole.

`circumference` *dist* WorldProperty

This property is the distance around the entire world (as a sphere). Default is `360`.

`axial-tilt` *n* WorldProperty

This property defines the extremes of seasonal changes.

`area` [*width* [*height*]] [*restriction*] *properties...* Form

This form defines the playing area of the world. The *restriction* identifies how to get data for this area from subsequent forms that are based on larger areas.

`restrict` *w h x y* AreaRestriction

This is a special form that specifies that subsequent layers in an area of size w x h will be offset by x,y and then read into the actual area. (This is useful for setting up a game that needs only a subset of a full map.)

Note that an area restriction is not a property, and must always appear before any properties in an area form.

`width` *n* AreaProperty

`height` *n* AreaProperty

These properties are the width and height of the world, as measured in cells. Allowable values range from 3x3 up to 32767x32767, which is one billion cells! If only one of these is given, then the other defaults to the same value. If neither has been given, then they default to `60` and `30`, respectively.

In the case of a cylinder, the world wraps around in the x direction, and the width is the diameter of the cylinder, while the height is just the height in the usual sense. A hexagon world is flat on the top and bottom; its width is measured across the middle height, which is the largest span, and height is the same as for cylinders. Here are some crude pictures, first of an 8x6 cylinder:

```
# # # # # # # #
 : : + + : : : :
: : : + ^ : : :
 : : : : : : : :
: : : : ^ : : :
 # # # # # # # #
```

This world is an 8x7 hexagon:

```
   # # # # #
  # : + + : #
 # : : + ^ : #
# : : + ^ : : #
 # : : : : : #
  # : : ^ : #
   # # # # #
```

There are two kinds of properties that an area may have: scalar values such as latitude, and layer values such as terrain and elevation.

`latitude` *n* AreaProperty

This property is the offset, in cells, from the equator of the middle of the area (height / 2). Defaults to `0`.

`longitude` *n* AreaProperty

This property is the offset, in cells, from the "Greenwich Meridian" of the world. Defaults to `0`.

4.3.1 Layers

Layers constitute the bulk of data about an area of the world. Each layer assigns a value to each cell in the area; examples include cell terrain, temperatures, elevations, and so forth. Since there may be many cells in a layer with the same values, each layer uses a common run-length encoding scheme. In this scheme, each horizontal band of cells is a separate text string, and the contents of the string encode individual numeric values, one for each cell. The encoding uses the characters `a..~` and `:..[` for 0 through 63, and decimal digits followed by commas (or the end of the string) for all other numbers. An optional `-` is allowed, and indicates a negative value. Runs of constant value are prefixed with their length, in decimal. The character `*` separates run lengths from values expressed as digits. Thus, the string

```
"40adaa100,2*-99"
```

represents 46 values in all: 40 zeroes, a three, 2 more zeros, a 100, and two -99s. Although this format is quite unreadable, it has the advantages of compactness and portability; the expectation is that most layer editing will be done on-line. Note that the run encoding is entirely optional.

The following subforms at the beginning of layer data have special effects:

`constant` *n* LayerSubform

This subform causes every value in the layer to be set to *n*.

`subarea` *x y w h* LayerSubform

This subform indicates that the layer data should be positioned at the given rectangle in the layer.

`xform` *mul add* LayerSubform

This subform has the effect of first multiplying the raw value by *mul*, then adding *add* and storing the result into the layer.

`by-bits` LayerSubform

`by-char` *str* LayerSubform

This subform specifies that the characters in *str* give the encodings of values in the layer. The first character in *str* encodes 0, the second encodes 1, and so forth.

`by-name` *name-list* LayerSubform

[what is the syntax of name-list exactly?] This subform is for generic worlds that are useful across multiple game designs. The value/name pairs allow for the matching of terrain types by name, so that if, say, the "sea" terrain type was type #0 in one game and type #4 in another, the world would have sea in all the same places after it was read in. In practice, only a few worlds are this general. If a named terrain type is not present, *Xconq* will warn about it and substitute type 0.

`terrain` *layer-data...* AreaProperty

This property is the actual layer of terrain types for cells.

`aux-terrain` *terrain-type layer-data...* AreaProperty

This property fills in values for borders, connections, and coatings. For border and connection terrain, the value is a six-bit number (0..63), with a bit turned on in each direction that there is a border or connection. For coating types, the value is the depth of the coating.

`features` *feature-list layer-data...* AreaProperty

This property specifies the nature and location of all geographical features. The *feature-list* is a list of lists, where each sublist has the form `([`*id*`]` *typename name* `[`*super*`])` where *id* is the numerical id referenced in the layer data (defaults to feature's position in the *feature-list*), *typename* is a symbol or string giving the general type of feature (such as `bay`), *name* is the name of the feature (such as `"Bay of Bengal"`), and *super* is the optional id of another feature that incorporates this feature.

`material` *material-type layer-data...* AreaProperty

This property declares the quantity of the given *material-type* in each cell of the area.

`people-sides` *layer-data...* AreaProperty

This property says which side the people of each cell are on. A *side-encoding* of `exact` assigns 0 to independence (no side), 1 to the first side, and so forth; otherwise, the encoding is a list of side names/ids and numbers.

4.3.2 Distances and Elevations

`elevations` *layer-data...* AreaProperty

This property is the world elevation data itself. If any elevation falls outside the min/max elevation range for the terrain type of the cell, then it will be truncated appropriately. Defaults to `0` for each cell.

`cell-width` *dist* AreaProperty

This property is the distance across a single cell, expressed as units of elevation. Defaults to `1`.

4.3.3 Temperatures

Each type of terrain has a temperature range in which it may be found. Any calculation that would fall outside this range will be clipped.

The temperature can be set to have a given value at a given elevation. All air temperatures will be interpolated appropriately.

`temperature-floor` *n* GlobalVariable

This variable is the lowest possible temperature. Defaults to `0`.

`temperature-floor-elevation` *n* GlobalVariable

This variable is the elevation at which the temperature is always at `temperature-floor`. Defaults to `0`.

`temperatures` *layer-data...* AreaProperty

This property contains the temperature data itself. If any temperature falls outside the min/max temperature range, then it will be truncated appropriately. Defaults to `0` for each cell.

4.3.4 Winds

Winds are defined as having a nonnegative force and a direction.

`winds` *layer-data...* AreaProperty

This property contains the force and direction of the prevailing winds in each cell.

4.3.5 Clouds

Cloud cover is defined as a layer over the terrain, with a bottom and top and density for each cell. In the example below, `o` and `O` represent different densities of cloud, and `-` show the tops and bottoms, while `^` shows the ground.

```
    ----     -
  --o00o --  0
  00o00o oo--0
  --o00- --000
    ---    ---
^^^^^^^^^^^^^^^^
```

`clouds` *layer-data...* AreaProperty

This property is the degree of cloud cover over each cell. A value of `0` corresponds to clear skies.

`cloud-bottoms` *layer-data...* AreaProperty

This property is the altitude above the ground of the bottoms of the clouds.

`cloud-heights` *layer-data...* AreaProperty

This property is the vertical thickness of the cloud cover in each cell.

4.4 Sides

`side` [*id*] *properties...* Form

This form has the effect of declaring a side to exist. If the number or symbol *id* is supplied and matches that of a side that has already been created, then the properties will modify the pre-existing side. Otherwise a new side object will be created, with a arbitrarily-chosen numeric id ranging between 1 and `sides-max`. If the given *id* is a symbol, then the side's numeric id will be bound to that symbol.

`sides-min` *n* GlobalVariable

`sides-max` *n* GlobalVariable

These variables are the minimum and maximum number of sides that may exist in a game. Defaults are to `1` and the internal parameter `MAXSIDES`, which is usually around 7. `MAXSIDES` can only be changed by recompiling *Xconq*.

`side-defaults` *properties...* Form

This form sets the defaults for all newly-created sides declared subsequently. These defaults will be set before the new side's properties are interpreted. This form has no effect on existing sides or on side declarations that modify existing sides.

4.4.1 Name and Related Properties

If the game design allows, all of these properties can be set at startup by the players (see <side config> and below). Omission of some of these results in suppression or substitution, depending on the interface and the situation. Omission of all name properties allows the side to go unmentioned, which is useful when the concept of "side" is useless or confusing to a player (as in some adventure games). All of these properties may be set at any time by any player.

`name` *str* SideProperty

This property is the proper name of a side, as a country or alliance name. Examples include `"Axis"` and `"Hyperborea"`. Defaults to `""`.

`long-name` *str* SideProperty

This property is the long form of a side's name, as in `"People's Republic of Hyperborea"`. Defaults to be the same as the side's name.

`short-name` *str* SideProperty

This property is an short name or acronym for the side, often just the letters of the long name, as in `"PRH"`. Defaults to `""`.

`noun` *str* SideProperty

This property is the name of an individual unit or person belonging to the side. Defaults to `""`, which suppresses any mention of the side when (textually) describing the individual.

`plural-noun` *str* SideProperty

This property is what you would call a group of individuals. Defaults to the most common plural form of the `noun` (in English, the default pluralizer adds an "s"), so any alternative plural noun, such as `"Chinese"`, will need an explicit `plural-noun` value.

`adjective` *str* SideProperty

This property is an adjective that can be used of individuals on the side, as in `"Spanish"`. Defaults to `""`, which suppresses use of the adjective.

As a complete example, a side named `"Poland"` would have a long name `"Kingdom of Poland"`, short name `"Po"`, noun `"Pole"`, plural noun `"Poles"`, and adjective `"Polish"`.

`color` *str* SideProperty

This property is a comma-separated list of colors that represents the side. Defaults to `"black"`.

`emblem-name` *str* SideProperty

This property is the name of a graphical icon that represents the side. An emblem name of `"none"` suppresses any emblem display for the side. Defaults to `""`, which gives the side a randomly-selected emblem.

`names-locked` *t/f* SideProperty

If the value of this property is `true`, then the player cannot modify any of the side's names. Defaults to `false`.

4.4.2 Side Class

`class` *str* SideProperty

This property is a side's class, which is a keyword that characterizes the side. Any number of sides may be in the same class. Defaults to `""`.

4.4.3 Status in Game

Once a side is in the game, it can never be totally removed. However, sides can become inactive.

`active` *t/f* SideProperty

This property is `true` if the side is still actively participating in the game. If the side has won, lost, or simply withdrew, this will be `false`. Any units on a side not in the game are effectively frozen statues; they don't do anything, and are untouchable by anyone else. Defaults to `true`.

`status` *lose/draw/win* SideProperty

This property tells how this side did in the game. Defaults to `draw`.

`win` GlobalConstant

`draw` GlobalConstant

`lose` GlobalConstant

These constants are the different possible values for a side's status.

`advantage` *n* SideProperty

`advantage-min` *n* SideProperty

`advantage-max` *n* SideProperty

Initial and min/max limits on advantage for the side. All default to the values of the corresponding global variables.

4.4.4 Side Relationships

By default, sides are neutral with respect to each other.

Control is a situation where one side can observe and move another side's units, but not vice versa. The controlling side can also just take the units of the controlled side. If the controlled side loses or resigns, then the controlling side automatically gets everything. Both sides must agree to this relationship.

`controlled-by` *side* SideProperty

This property refers to the side controlling this one. If 0, then the side is not under control. Defaults to `0`.

The closest side relationship is one of trust. A trusted side unit's may do anything at any time, including entering and leaving units on the other side, consuming the other side's materials, and so forth.

`trusts` *side-value-list* SideProperty

This property is true for any side that is trusted by this side. Note that this relationship need not be symmetrical. Defaults to `false` for all sides.

Note that these parameters apply only to relationships as enforced by *Xconq*. In an actual game, both human and robot sides can make agreements and have positive/negative opinions about the other sides.

`trades` *side-value-list* SideProperty

This property defines the trading relationship with other sides. Defaults to `0` for all sides.

4.4.5 Numbering Units

`next-numbers` *utype-value-list* SideProperty

This property gives the next serial numbers that will be assigned to units acquired by this side. Defaults to `1` for each unit type (Dijkstra notwithstanding, that's still where people start numbering things).

If the unit is of a type that gets numbered (`assign-number` property is true), then any unit of that type, acquired by any means whatsoever, will be assigned the `next-numbers` value for that type and `next-numbers` will be incremented.

4.4.6 Side-Specific Namers

A side can have its own set of namers (see below) that will be used for units and geographical features associated with that side.

`unit-namers` *utype-value-list* SideProperty

This property specifies which namers will be used with which types that the side starts out with or creates new units. These will not be run automatically on captured units or gifts. Defaults to `""` for each unit type.

`feature-namers` *feature-type-value-list* SideProperty

This property specifies which namers to use with which geographical features in the side's initial country (if if has one). Defaults to `()`.

4.4.7 Tech Levels

The tech level of a side determines what it can do with each type of unit.

`tech` *utype-value-list* SideProperty

This property assigns a tech level to each unit type named. Defaults to `0` for each unit type.

`init-tech` *utype-value-list* SideProperty

This property is the tech level at the beginning of the current turn. Defaults to `0` for each unit type.

4.4.8 Views

These properties are necessary only if the relevant globals are set a certain way (`see-all` is false, etc).

`terrain-view` *layer-data...* SideProperty

This property is the side's current knowledge of the world's terrain. Defaults to `()`.

`unit-view` *layer-data...* SideProperty

This property is the side's current knowledge of the world. Defaults to `()`.

`unit-view-dates` *layer-data...* SideProperty

This property is the turn number at which the unit view data in the corresponding cell of the `unit-view` was set. Defaults to `()`.

4.4.9 Interaction

`turn-time-used` *seconds* SideProperty

This property is the number of (real) seconds that this side has been moving units during the present turn. Defaults to `0`.

`total-time-used` *seconds* SideProperty

This property is the number of (real) seconds that this side has been moving units during the course of the game. Defaults to `0`.

`timeouts` *n* SideProperty

This property is the number of "time outs" a side gets for the game. Defaults to `0`.

`timeouts-used` *n* SideProperty

This property is the number of "time outs" a side has already used up. Defaults to `0`.

`finished-turn` *t/f* SideProperty

This property is true if the side has declared that it is finished moving things during this turn. Defaults to `false`.

`willing-to-draw` *t/f* SideProperty

This property is true if the side will go along with any other side that wants to end the game in a draw. Defaults to `false`.

`respect-neutrality` *t/f* SideProperty

`real-timeout` *seconds* SideProperty

This property is the number of (real) seconds to wait before declaring the side to be finished with this turn. Defaults to `-1`, which waits forever.

`task-limit` SideProperty

This property is the maximum number of tasks a unit is allowed to stack up.

4.4.10 Doctrine

Doctrines are objects that units consult to decide about individual behavior.

`doctrines` *utype-property-groups...* SideProperty

This property is the side's unit-type-specific doctrine. Each *utype-property-group* has the form `(`*unit-types* `doctrine)`. Defaults to `()`.

`doctrines-locked` *t/f* SideProperty

This property says whether the docrine-unit type correspondence for the side may be altered during the game. This property does not control whether or not the properties of the doctrines may be altered. Defaults to `false`.

`doctrine` [*id*] *properties...* Form

This form creates a doctrine with the given id and properties.

`ever-ask-side` *t/f* DoctrineProperty

This property is the true if the unit may ask the player for what to do, instead of picking some default activity.

`avoid-bad-terrain` *n%* DoctrineProperty

This property is the probability that the unit will not enter unhealthy terrain, even if it delays meeting goals. Unhealthy means higher attrition and accident probs, materials consumed faster than replaced, slower movement. Defaults to `0`.

`repair-at` *n%* DoctrineProperty

This property indicates that when the unit's hp is at *n%* of max, make a plan to repair. Defaults to `50`.

`resupply-at` *n%* DoctrineProperty

This property indicates that when the level of a operationally-consumed material is at *n%* of capacity, try to resupply. Defaults to `50`.

`rearm-at` *n%* DoctrineProperty

This property indicates that when the level of a combat-consumed material is at *n%* of capacity, try to resupply. Defaults to `50`.

`locked` *t/f* DoctrineProperty

This property is true if the properties of the doctrine cannot be modified by the side's player during the game. Defaults to `false`.

4.4.11 Other

`self-unit` *unit* SideProperty

This property is the id of a unit that represents the side itself. Defaults to `0`, which means that no unit represents the side. See below for more details on self units.

`priority` *n* SideProperty

The order in which the side will get to act, relative to other sides and to units. Defaults to `0`.

`scores` (*skid val*)... SideProperty

This property is the current values of any numeric scores being kept for the side. It is a list of pairs of scorekeeper id and value. Defaults to `()`.

`independent-units` *properties...* Form

Like the `side` form, but sets properties for independent units.

`ui-data` *data...* SideProperty

This property contains interface-specific data for the side. This is mainly for preservation across game save/restores, and its form is defined by the interface.

`ai-data` *data...* SideProperty

This property is information about the AIs associated with a side. The format and content of *data* is determined by the type(s) of the AIs. Defaults to `()`.

4.5 Players

Player objects are rarely necessary when building game designs; they typically only appear in saved games, in order to ensure that the same players get the same sides upon restoration.

`player` *id* SideProperty

This property is the unique identifier of a player that is running this side. Defaults to `0`, which means that no player has been assigned to the side.

`player` [*id*] *properties...* Form

This form defines a player. If the *id* is supplied and matches the id of an existing player, then the player object is updated using the *properties*, otherwise a new player object will be created, using the given *id* if supplied, otherwise creating a new value.

`player-sides-locked` *t/f* GlobalVariable

This variable is `true` if the player/side assignment may not be changed while the game is starting up. Defaults to `false`.

The number of players must always be less than the number of sides (sides without players just don't do anything).

`name` *str* PlayerProperty

This property identifies the player by name. Defaults to `""`.

`config-name` *str* PlayerProperty

This property identifies a particular set of doctrine and other definitions that the player is using. Defaults to `""`.

`display-name` *str* PlayerProperty

This property identifies the display being used by the player's interface. The interpretation of this value is dependent on the interface in use. Defaults to `""`.

`ai-type-name` *str* PlayerProperty

This property is the type of AI that will play the side if requested or necessary. The set of choices depends on what has been compiled into *Xconq*. (The general-purpose AI type `"mplayer"` will usually be available, but is not guaranteed.) An `ai-type-name` of `""` means that no AI will run this player. Defaults to `""`.

`password` *str* PlayerProperty

This property is the encoding of a password that must be entered before this player object can be reused successfully. Defaults to `""`.

`initial-advantage` *n* PlayerProperty

This property is an initial relative strength at which the player should start. Some synthesis methods can use this to give more units or some other advantage to each player according to the requested strength. Defaults to `1`.

`advantage-min` *n* GlobalVariable

`advantage-max` *n* GlobalVariable

`advantage-default` *n* GlobalVariable

These variables set the bounds and default values for players' initial advantages. Default to `1`, `9999`, and `1`, respectively.

Xconq is not guaranteed to be able to be able to set up a game with any combination of player advantages; the limits depend on the capabilities and characteristics of the synthesis methods that use the requested advantages in their calculations.

4.5.1 Rules of Side Configuration

The properties of a side can come from a number of different sources (here listed in order of precedence):

Interface-specific sources (X resources, Mac preferences).

Game-specific form in player's configuration file.

Generic form in player's configuration file.

The `side` form for the side.

The `side-defaults` form for the game.

General program defaults.

Note that interface-specific and general config files can never alter certain properties of a side, and can only alter others if they are not locked.

4.6 Units

The basic `unit` form creates or modifies a unit.

`unit` *id* [*type*] *properties...* Form

This form defines a unit. If a numeric *id* is supplied and matches the id of an existing unit, then that unit will be modified by *properties*, and the optional *type* will be interpreted as a new type for the unit. Otherwise a new unit will be created, with either *id* as its id or a arbitrarily-selected one if *id* is already in use. If the unit's id is newly-generated and no type has been specified, then type #0 (first-defined type) will be the type of the unit. An id of `0` can never match an existing unit id, so effect will be as if it had been omitted.

unit-type-name x y [*side-id*] *properties...* Form

This is an abbreviated form, in which the x,y position is required, and an optional side id may be included. The side id will come from `unit-defaults` if not specified. The *unit-type-name* may be any valid unit type name or defined name. This form always results in a new unit.

Since there may be many units whose properties are similar, there is a "default unit" whose properties fill in missing properties in individual unit declarations.

`unit-defaults` [*modifier*] *properties...* Form

This form sets the default values for all subsequent units read in, in this and every other module not yet loaded. The set of defaults is additive, so for instance you can repeatedly change the default side of units. If the symbol `reset` has been supplied for the optional *modifier*, then all the defaults will be changed to the basic default values, as described in this manual.

`reset` Symbol

This is the symbol used to reset unit defaults; see above.

4.6.1 Unit Properties

This section lists properties of individual units. In general, they default to the most common or reasonable values, so need not always be specified, even in a saved game.

`@` *x y* [*z*] UnitProperty

This property is the position of the unit. Defaults to `-1,-1,0`, which causes the unit to be placed randomly. The optional altitude *z* can also be set separately with the property `z` below. If *z* is even and the unit is in the open, then the unit's altitude is *z/2*; if *z* is odd, then *(z-1)/2* is the type of connection terrain that the unit is on.

`z` *z* UnitProperty

This property is identical to the optional z part of the `@` property. Defaults to `0`.

`s` *side* UnitProperty

This property is the side of the unit. It can be either a side name/noun/adjective (string) or id (number). A value of `0` or `"independent"` means that the unit is independent. Defaults to `0`.

`#` *n* UnitProperty

This property is the unique numeric id of the unit. Defaults to a game-selected value.

`n` *str* UnitProperty

This property is the name of the unit. Defaults to `""`.

`nb` *n* UnitProperty

This property is the number of the unit, which starts at `1` and goes up. Defaults to `0`, which means that the unit is unnumbered.

`cp` *n* UnitProperty

This property is the current completeness of the unit. If negative, indicates that the unit will appear at a time and place specified by the `appear` x-property. Defaults to the `cp-max` for the type.

`hp` *n* UnitProperty

This property is the current hit points of the unit. Will be restricted to the range [0, hp-max]. An hp of 0 means that the unit is dead and will not appear in the game. Defaults to `hp-max` for the unit's type.

`cxp` *cxp* UnitProperty

This property is the combat experience of the unit. Defaults to `0`.

`mo` *n* UnitProperty

This property is the morale of the unit. Defaults to `0`.

`m` *mtype-value-list* UnitProperty

This property is the amounts of supplies being carried by the unit. Defaults to `0` for each material type.

`tp` *utype-value-list* UnitProperty

This property is the level of tooling to build each type of unit. Defaults to `0` for each unit type.

`in` *n* UnitProperty

This property is the id of the unit's transport. Defaults to `0`, meaning that unit is not in any transport.

`opinions` *side-value-list...* UnitProperty

This property is the unit's true feelings towards each side, including its own side. Defaults to `0` for each side.

`x` *obj* UnitProperty

This property is the optional extension properties of the unit. Its value may be any object. Defaults to `()`.

`appear` Symbol

`disappear` Symbol

These are extension properties that indicate when and where a unit will appear in the game, and when it will disappear. [syntax?]

4.6.2 Unit Action State

`act` *subprops* UnitProperty

This property specifies the current action state of the unit.

`acp` *n* UnitActionStateProperty

This property is the number of action points left to the unit for this turn. Defaults to `0`.

`acp0` *n* UnitActionStateProperty

This property is the initial number of action points for this turn, computed at the beginning of the turn. Defaults to `0`.

`aa` *n* UnitActionStateProperty

This property is the actual number of actions executed by the unit so far in the current turn. Defaults to `0`.

`am` *n* UnitActionStateProperty

This property is the actual number of moves (cell entries) executed so far in the current turn. Defaults to `0`.

`a` *action* UnitActionStateProperty

This property is the next action that the unit will perform.

Note that if any unit-defining form has an `act` property, *Xconq* will start at an appropriate point in the middle of a turn, giving all other units zero acp and mp, rather than starting at the beginning of the turn and computing acp and mp for all units.

4.6.3 Unit Plan

`plan` *type* [*subtype*] *properties...* UnitProperty

This property describes the unit's current plan.

`none` PlanType

A unit with this type of plan does nothing. It is used when a side has no player.

`passive` PlanType

This plan type is for units on a side that is being run directly by the side.

`defensive` PlanType

This plan type is for units that defend areas or other units.

`exploratory` PlanType

This plan type is for units that explore the world.

`offensive` PlanType

`random` PlanType

A unit with this plan type will act randomly.

`goal` PlanProperty

This property is the main goal of a unit's plan.

The possible types of goals are these:

`no-goal` GoalType

`won-game` GoalType

`lost-game` GoalType

`world-is-known` GoalType

`vicinity-is-known` — GoalType

`positions-known` — GoalType

`cell-is-occupied` — GoalType

`vicinity-is-held` — GoalType

`has-unit-type` — GoalType

`has-unit-type-near` — GoalType

`has-material-type` — GoalType

`keep-formation` — GoalType

[also support some kind of hook for specific AIs?]

`tasks` *tasks...* — PlanProperty

This property is the complete task agenda for the unit's plan. It is a list of tasks. Defaults to `()`.

`build` *u n n2 unit-id* — TaskType

`capture` *unit-id* — TaskType

`do-action` *action* — TaskType

`hit-position` *x y z* — TaskType

`hit-unit` *unit-id* — TaskType

`move-dir` *dir* — TaskType

`move-to` *x y z dist* — TaskType

`occupy` *unit* — TaskType

`pickup` *unit* — TaskType

`repair` *unit* — TaskType

`resupply` — TaskType

`sentry` *n* — TaskType

`asleep` *t/f* — PlanProperty

This property is true if the unit is asleep. Defaults to `false`.

`reserve` *t/f* — PlanProperty

This property is true if the unit is in reserve. Defaults to `false`.

`wait` *t.f* PlanProperty

This property is true if the unit is waiting for orders. Defaults to `false`.

`formation` *goal* PlanProperty

4.7 Agreements

`agreement` [*name/id*] *properties...* Form

This form defines an agreement among a set of sides. The name/id is a unique internal identifier.

`type-name` *str* AgreementProperty

This property is the name of the general type of agreement, such a trade. Defaults to `""`.

`title` *str* AgreementProperty

This property is the player-visible name of the agreement. Defaults to `""`.

`terms` *forms...* AgreementProperty

This property is the list of terms of the agreement. Defaults to `()`.

`drafters` *side-list* AgreementProperty

This property is the side that initially proposed the agreement.

`proposers` *side-list* AgreementProperty

This property is the side that initially proposed the agreement.

`signers` *side-list* AgreementProperty

Before the agreement is made, this property is the proposed list of participants. After the agreeement is made, this is the actual list of participants.

`willing-to-sign` *side-list* AgreementProperty

This property is all the sides that have already agreed to this agreement, on condition that all the other sides accept it.

`known-to` *side-list* AgreementProperty `enforcement` *form* AgreementProperty

[include values such as `enforced` and `publicity`?]

`state` *state* AgreementProperty

[add symbols for states]

4.8 Scorekeepers

Scorekeepers are the objects that manage scoring, winning, and losing. A game design need not define any scorekeepers, and none are created by default. A scorekeeper may either maintain a numeric score that is used at the end of the game to decide rankings, or simply declare a side to have won or lost.

`scorekeeper` *name properties...* Form

This form creates or modifies a scorekeeper with the given *name*, with the given *properties*.

`title` *str* ScorekeeperProperty

This property is a string that identifies the scorekeeper to the players. Defaults to `""`.

`when` (*type* [*exp*]) ScorekeeperProperty

This property is when the scorekeeper will be checked or updated. Defaults to `after-turn`.

`before-turn` *exp* ScorekeeperWhenType

This indicates that the scorekeeper will run at the start of each turn matching *exp*, or after every turn if *exp* is not given.

`after-turn` *exp* ScorekeeperWhenType

This indicates that the scorekeeper will run at the end of each turn matching *exp*, or after every turn if *exp* is not given.

`after-event` *exp* ScorekeeperWhenType

This indicates that the scorekeeper will run after every event matching *exp*, or after every event if *exp* is not given.

`after-action` *exp* ScorekeeperWhenType

This indicates that the scorekeeper will run at the end of each action matching *exp*, or after every action if *exp* is not given.

`applies-to` *side-list* ScorekeeperProperty

This property is the set of sides or side classes to which the scorekeeper applies. Scorekeepers apply only to sides that are in the game. Defaults to `side*`.

`known-to` *side-list* ScorekeeperProperty

This property is the list of sides that know about this scorekeeper, and can see the value of the score for each side that it applies to. Defaults to `side*`.

`trigger` *form* ScorekeeperProperty

This property is an expression that is true when it is time to start checking the scorekeeper's main test. Once a scorekeeper is triggered, it remains active. Defaults to `false`.

`triggered` *t/f* ScorekeeperProperty

This property is true if the scorekeeper is currently triggered. Defaults to `true`.

`do` *forms...* ScorekeeperProperty

This property is a list of forms to execute in order each time the scorekeeper runs. Defaults to `()`.

`messages` *forms...* ScorekeeperProperty

This property is a list of messages to be sent [???]. Defaults to `()`.

`initial` *value* ScorekeeperProperty

This property is the value of the score upon game startup. If this value is `-9999`, the scorekeeper does not maintain a numeric score. Defaults to `-9999`.

4.8.1 Bodies

The forms in the body (the `do` property) of the scorekeeper may be any of the forms listed here.

`last-side-wins` ScorekeeperForm

If supplied as the only symbol in the body, then the scorekeeper implements the usual "last side left in the game wins" behavior.

`if` *test action* ScorekeeperForm

If the *test* evaluates to `true` or any nonzero number, then the *action* will be done.

`cond` (*test actions...*) ... ScorekeeperForm

This is like Lisp's cond.

`stop` [*message*] ScorekeeperForm

This stops the game immediately, with a draw for all sides.

`win` [*sides*] [*own-message*] [*other-message*] ScorekeeperForm

`lose` [*sides*] [*own-message*] [*other-message*] ScorekeeperForm

`end` [*message*] ScorekeeperForm

This scorekeeper action ends the game immediately.

`add` *exp* [*side*] ScorekeeperForm

This adds the result of evaluating *exp* to the score of the given side. The value may be a negative number.

4.8.2 Scorekeeper Functions

`and` *exps* ScorekeeperFunction

`or` *exps* ScorekeeperFunction

`not` *exp* ScorekeeperFunction

`=` *exp1 exp2* ScorekeeperFunction

`/=` *exp1 exp2* ScorekeeperFunction

`>` *exp1 exp2* ScorekeeperFunction

`>=` *exp1 exp2* ScorekeeperFunction

`<` *exp1 exp2* ScorekeeperFunction

`<=` *exp1 exp2* ScorekeeperFunction

`sum` *types properties* [*test*] ScorekeeperFunction

4.8.3 Scorefile

`scorefile-name` *str* GlobalVariable

4.9 The History

All the important events in a game are logged into a history.

`evt` [*date*] *type* [*sides*] *data* Form

This form creates a single historical event. If *date* is omitted, then the date will be the same turn as for the last event read.

`exu` Form

`log-started` EventType

This event records when the recording of events began. Multiple instances of this may occur, for instance if logging were to be turned off and then on again.

`log-ended` EventType

`game-started` EventType

This event records the actual start of the game. There should only be one in a game's history.

`game-saved` EventType

`game-restarted` EventType

`game-ended` EventType

`side-joined` EventType

This event records when a side joined the game.

`side-lost` EventType

This event records when a side lost.

`side-withdrew` EventType

This event records when a side withdrew from the game.

`side-won` EventType
: This event records when a side won.

`unit-started-with` EventType
: [???]

`unit-created` *id* EventType
: This event records the creation of a unit.

`unit-completed` EventType
: This event records the completion of a unit.

`unit-acquired` EventType
: This event records the acquisition of a unit, for instance as a gift from another side.

`unit-captured` EventType
: This event records the capture of a unit, as an outcome of combat or from a direct attempt to capture.

`unit-moved` *id x1 y1 x2 y2* EventType
: This event records the movement of a unit.

`unit-name-changed` EventType

`unit-type-changed` EventType

`unit-assaulted` EventType

`unit-damaged` EventType

`unit-killed` EventType

`unit-vanished` EventType

`unit-wrecked` EventType

`unit-garrisoned` EventType

`unit-disbanded` EventType

`unit-starved` EventType

`unit-left-world` EventType

The following event types are the results of actions.

`action-ok` EventType

`action-error` EventType

`cannot-do` EventType

`insufficient-acp` EventType

`insufficient-material` EventType

`action-done` EventType

`insufficient-mp` EventType

`cannot-leave-world` EventType

`destination-too-far` EventType

`destination-full` EventType

`overrun-failed` EventType

`overrun-failed` EventType

`fire-into-outside-world` EventType

`fire-into-too-far` EventType

`fire-at-too-far` EventType

`fire-into-too-near` EventType

`fire-at-too-near` EventType

`too-far` EventType

`too-near` EventType

4.10 Battle States

Battles always have exactly two "sides", referred to as the attacker-list or A-list and the defender-list or D-list, so as not to confuse them with sides in the game.

`battle` *a-list d-list...* Form

Each list has the form

```
((<unit> <commitment>) ...)
```

4.11 Types in General

Types are the foundation of *Xconq* game designs. Nearly all the rules and game parameters are associated with the unit, material, and terrain types. There is no sort of type hierarchy; instead, most forms allow sets of types to be used in the place of single types.

Each type has an index associated with it, starting from 0. This index never appears directly, and cannot be set. This does mean that types have an order, so the order in which types are defined is sometimes significant. These cases will be noted. The order is always the order in which the types appear in the file, so it is always the same.

4.11.1 Naming

The names of types need not be distinct from each other, but you run the risk of player confusion if they share names.

`name` *string* TypeProperty

This property is the specific name of the type. This name will be displayed to players; the exact format is up to the interface, but will typically depend on the name's length and the space available in the display. If no type names have been defined, the internal type name (see below) will be used. Defaults to `""`.

`long-name` *string* TypeProperty

This property is a fully spelled-out name for the type. Defaults to `""`.

`short-name` *string* TypeProperty

This property is an abbreviated name of r Defaults to `""`.

`generic-name` *string* TypeProperty

This property is like `name`, but identifies the type less specifically, and several types may have the same generic name. If no generic names are defined, then the regular type names will be used. This is useful when making abbreviated lists, so that related types get counted together. Defaults to `()`.

As an example of the distinction between type names and generic type name, the names of a automobile type might be `"1965 Mustang"`, `"Mustang"`, and `"M"`, while the generic name is `"auto"`.

Then the interface could choose to display a parking lot as containing either `"4 auto"` or `"2 Mustang 1 Edsel 1 Jeep"`.

Note that names specified as properties are strings only, and are not defined as evaluable symbols.

4.11.2 Imaging

The interpretation of these properties is entirely up to each interface; see the appropriate interface documentation for details.

`image-name` *str* TypeProperty

This property is the name of the type's image. If undefined or unusable for some reason, the interface will display the type in some default manner, such as a solid-color square or a string.

For example, in X11, the name might be the name of a file in the usual bitmap format, as produced by the *bitmap* program. The actual file name is produced by appending `".b"`. (The situation in X is actually more complicated than this.) See the interface documentation for details on how the interface uses the image.

`color` *str* TypeProperty

This property is the name of the preferred color for this type. Both normal color names and the strings `"bg"` and `"fg"` (meaning "foreground color" and "background color") may be used. If the image is in color, then this property has no effect. Defaults to `"fg"`.

`char` *str* TypeProperty

This property supplies a single character for this type (all characters after the first one in *str* are ignored). Defaults to `""`.

4.11.3 Documentation

`description-format` *list...* TypeProperty

This property defines the different ways in which an instance or instances of this type may be described textually. This information may be used in narrative descriptions and by some interfaces. [describe syntax of the lists - are similar to name grammars] If `()`, then the instance will be described in some default fashion, such as (for units) `"the <side> <ordinal> <type>"`. Defaults to `()`.

`help` *string* TypeProperty

This property is a brief (preferably one-line) description of the type. Defaults to `""`.

`notes` *strings...* TypeProperty

This property is detailed documentation about the type. The formatting of the strings is up to the interface, but in general each string is a separate line, the string `""` indicates a line break, and two `""` in a row indicates a paragraph break. Defaults to `()`.

4.11.4 Availability

It may be that a set of types is larger than strictly necessary for a particular game. You can make any type unavailable, which means that irrespective of any other controls, that type cannot come into play during a game. You can also make it available only for particular turns.

`available` *n* TypeProperty

If the value of this property is greater than 0, then this type is available in the game on or after turn *n*. If the value is less than 0, then the type is available, but only until turn -*n*. If the value is 0, then the type is never available. Defaults to `1`, which means that the type is always available.

If a type becomes unavailable and there are units of that type in play, then they will vanish immediately.

4.11.5 Type Extension

It may occasionally be necessary to add new kinds of information to a type. For instance, new synthesis methods may require special data, or an interface may be able to use extra hints to improve its display. The `extensions` property can be used to store this kind of data.

`extensions` *properties...* TypeProperty

This property is a catch-all for nonstandard type properties. Anything may appear here, but it will only be interpreted as much as needed, and unrecognized extensions will not be warned about (so if you misspell one, you won't find out).

4.12 Unit Types

`unit-type` *symbol properties...* Form

This form defines a new type of unit. The *symbol* is required and must be previously undefined. The bindings in *properties* are then added to the type one by one. If no other name properties are defined, the *symbol* may be displayed to players (see above). You can define no more than 126 types of units.

The *symbol* here becomes the unit type's "internal type name" which is guaranteed unique. To make synonyms for the internal type name, use `define`.

`u*` GlobalVariable

This variable evaluates to a list of all unit types, listed in the order that they were defined. This list always reflects the list of types at the moment it is evaluated.

`non-unit` GlobalVariable

This variable [constant?] evaluates to a value that is NOT a unit type. This is needed in several places to enable/disable features. Use of this in any other way is an error, and may or may not be detected before it causes a crash.

4.12.1 Unit Naming

`namer` *namer-id* UnitTypeProperty

This property is the namer that will be used to generate names for units, if the unit's side does not have a namer, or the unit is independent and not in any country. Defaults to `0`, which leaves the unit unnamed.

`assign-number` *t/f* UnitTypeProperty

This property is true if the unit should have a serial number assigned to it by the side it belongs to. Serial numbers are maintained for each type on each side separately, start at 1 for the first unit of the type, and increase by one each time. Defaults to `true`.

4.12.2 Class-Restricted Unit Types

Sometimes the designer will want to make different sides have different types of units. Although this can be done by setting up scenarios appropriately, that won't close all the loopholes that might allow a side to get units that should only ever belong to another side.

The first step is to define a class for each side. For instance, a side named `"Rome"` might have a class `"Roman"`, while the sides named `"Aedui"` and `"Parisii"` could both be in the class `"barbarian"`.

`possible-sides` *exp* UnitTypeProperty

This property restricts the unit type to only be usable by a side meeting the conditions of *exp*. If *exp* is a string, it restricts the unit type to only be usable by a side whose class includes a matching string. This can also be a boolean combination. Independent units belong to a side whose class is `"independent"`. The default of `""` allows the unit to belong to any side.

4.12.3 Self-Units

The self-unit can be any type, including one that cannot act; for instance, a capital city could be the self-unit, thus making its defense all-important for a player.

`self-required` *t/f* GlobalVariable

This variable is true if each side is required to have a self-unit at all times. However, if no unit of a suitable type is available when the game begins, then none will be required. Defaults to `false`. [this should also have a related side property?] [rounding-down advantage should not eliminate one needed as self-unit?]

`can-be-self` *t/f* UnitTypeProperty

This property says that the type of unit can represent the side directly. Defaults to `false`.

`self-changeable` *t/f* UnitTypeProperty

This property is true if the player can choose to change a self-unit of this type at any time. Otherwise the self-unit can be changed only if the current one dies. Defaults to `false`.

`self-resurrects` *t/f* UnitTypeProperty

This property is true if when the self-unit dies, another unit of an allowable type becomes the self-unit automatically. Defaults to `false`.

Observe that these parameters can be used to develop various forms of backup, so that a player can start out as a capital city, resurrect as a town, change self to one of several towns, then lose when all the towns are lost.

`direct-control` *t/f* UnitTypeProperty

This property is true if a unit of this type can be controlled by its side automatically. If false, then it must be within range of a unit that can control it, and is itself under control by the side. Defaults to `true`.

`control-chance-at` *u1 u2* -> *n%* Table

`control-chance-adjacent` *u1 u2* -> *n%* Table

`control-chance` *u1 u2* -> *n%* Table

`control-range` *u1 u2* -> *dist* Table

This table gives the maximum distance from self-unit *u1* at which units of type *u2* can be controlled directly. Units further away always act on their own (as if the doctrine said so[?]). If this value is `<` 0, then *u1* can never directly control any other *u2* on the side. Defaults to `infinity`.

4.12.4 Limiting Unit Quantities

The effect of these is to prevent any extra units from being created or from going over to a side, regardless of the reason. This happens by either preventing player actions that would result in exceeding a limit (such as when building units), or by making the unit vanish instantly (such as when capturing a unit).

`units-in-game-max` *n* GlobalVariable

This variable is the maximum number of all types of units, on all sides, including independents, that may exist at any time, including initially. Defaults to `-1`, which means that there is no limit.

`units-per-side-max` *n* GlobalVariable

This variable is the maximum number of units (of all types together) that any side may have, at any time. Events that would cause the limit to be exceeded, such as capturing a unit, result in either the unit vanishing or becoming independent. Defaults to `-1`, which means that there is no limit.

There is no limit on the number of units that may be independent.

`type-in-game-max` *n* UnitTypeProperty

This property is the maximum total of the given type, for all sides together. Defaults to `-1`, which means that there is no limit.

`type-per-side-max` *n* UnitTypeProperty

This property is the maximum number of units of the given type allowed to each side. Defaults to `-1`, which means that there is no limit.

4.12.5 Hit Points

A unit's hit points determine how healthy it is. If a unit's hp goes below 1, it is either *wrecked*, meaning that it changes to a new type `wrecked-type` or else it *vanishes*, meaning that it is completely cleared from the world.

`hp-max` *n* UnitTypeProperty

This property is the maximum number of hit points for (each part of) a unit. Completed units start with this many hit points. Defaults to `1`.

`parts-max` *n* UnitTypeProperty

This property declares that a unit is to be treated as an aggregate of *n* smaller identical units. Defaults to `1`.

`wrecked-type` *unit-type* UnitTypeProperty

This property is the type of unit that a unit with 0 hp will become. For instance, a destroyed "fort" might become a "rubble pile" unit. If its value is `non-unit`, then the destroyed unit just vanishes. The `wrecked-type` of a type must be a different type. Defaults to `non-unit`.

The transformation to the wrecked type does not change position or name. The transformed unit has full hp, supplies are conserved as much as possible, tooling is preserved, and any unit plan is erased. It has the same number of parts, or as many as possible if that is fewer. It may be that the wrecked type is on terrain that it cannot survive on; in that case, it will be wrecked again, repeating until the unit either vanishes or is in a viable position, or this process has been repeated more times than the number of unit types (prevents infinite loops). Any excess occupants will be removed and either placed in another nearby unit or in the open, or will vanish if there is no other option.

`hp-recovery` *n* UnitTypeProperty

This property is the number of 1/100 hp recovered per turn. Recovery happens automatically, as opposed to repair, which requires explicit action. The amount *n* / 100 is recovered automatically each turn, while *n* mod 100 is the percent chance of recovering 1 hit point in addition. Defaults to `0`.

4.12.6 Experience

`cxp-max` *cxp* UnitTypeProperty

This property is the maximum combat experience this type of unit can have. Defaults to `0`.

4.12.7 Tech Levels

Before it can do anything with a type of unit, the side must have the appropriate tech level for that type, which is just a number ranging from 0 up to `tech-level-max`. Each type has a distinct tech level.

Tech levels always increase (since they represent abstract knowledge rather than physical plant). Tech can be transferred freely to any other side via the message `tech` [xref to messages].

For each unit type, the following parameters define the minimum tech levels at which sides can do various things.

`tech-to-see` *tl* — UnitTypeProperty

This property is the minimum tech level that a side must have before it can see a unit of this type. Defaults to `0`.

`tech-to-own` *tl* — UnitTypeProperty

This property is the minimum tech level that a side must have in order to have a unit of this type. Defaults to `0`.

`tech-from-ownership` *tl* — UnitTypeProperty

This property is the tech level that may be reached by acquiring a unit of this type. Since this is expressed as a minimum, multiple acquisitions have no additional effect. Defaults to `0`.

`tech-to-use` *tl* — UnitTypeProperty

This property is the minimum tech level that a side must have in order to give actions to this type of unit. Defaults to `0`.

`tech-to-build` *tl* — UnitTypeProperty

This property is the minimum tech level that a side must have in order to build this type of unit. Defaults to `0`.

`tech-max` *tl* — UnitTypeProperty

This property is the absolute maximum tech level possible for this type. Defaults to `0`.

`tech-crossover` *u1 u2* -> *n%* — Table

This table is the minimum tech level for *u2* that is guaranteed by a particular tech level for *u1*, expressed as a percentage of the `tech-max` for the types. For instance, if `tech-crossover` is 80, and the tech level for *u1* is 10 out of a max of 20, and the max for *u2* is also 20, then the side has a tech for *u2* at least 8. Defaults to `0`.

It is possible to gain some tech level just by being in the same game with a side that is more advanced.

`tech-leakage` *.01tl* UnitTypeProperty

This property is the amount of tech level gain per turn that can happen to any side's tech level that is less than the max of all sides in the game. This only happens if at least one unit on the side has nonzero coverage of a unit on a more advanced side. Defaults to `0`.

4.12.8 Opinions

`has-opinions` *t/f* UnitTypeProperty

This property is true if the unit has opinions about sides, both other sides and its own. Defaults to `false`.

4.12.9 Point Value

Point values provide an abstract way to characterize the overall importance of a unit type. Point values figure into some scorekeepers, and are used by AIs.

`point-value` *n* UnitTypeProperty

This property is the "value" of a unit. Defaults to `1`.

4.13 Terrain Types

Terrain types are associated with the cells, borders, connections, and coatings in a world.

`terrain-type` *name properties...* Form

This form defines a new type of terrain, named by *name*. Details are similar to those for unit types.

`t*` GlobalVariable

This variable evaluates to a list of all terrain types, listed in the order that they were defined.

`non-terrain` GlobalVariable

This variable has a value that is guaranteed not to be a terrain type.

4.13.1 Terrain Subtypes

Terrain can appear in four different roles: as the interior of a cell, as a border between cells, as a connection between cells, or as a coating overlaying the normal terrain. The terrain subtype says which role a type can play.

`subtype` *subtype* — TerrainTypeProperty

This property is the role that the terrain type can appear in. Defaults to `cell`.

`cell` — GlobalConstant

This constant indicates that terrain can fill a cell. All units in the open and with an altitude of 0 are assumed to be surrounded by the cell terrain.

`border` — GlobalConstant

This constant indicates that the terrain can be a border.

`connection` — GlobalConstant

This constant indicates that the terrain can be a connection.

`coating` — GlobalConstant

This constant indicates that the terrain can be a coating. A *coating* is a temporary terrain modification. The classic example is snow, which effectively changes some kinds of terrain, but not completely and usually not permanently. Cells can have varying heaviness of each type of coating.

`coating-depth-min` *t1 t2* -> *n* — Table

In order for a coating *t1* to "stick", this table says much must be added all at once to terrain *t2*. A coating depth that drops below this will disappear immediately. Defaults to `0`.

`coating-depth-max` *t1 t2* -> *n* — Table

This table is the upper limit on coating depth. Defaults to `0`.

Terrain types may have additional subtype attributes that are used only during synthesis, to select appropriate subtypes for special purposes.

`subtype-x` *n* TerrainTypeProperty

This property is extra subtype information, used in synthesis. Defaults to `no-x`.

`no-x` GlobalConstant

`river-x` GlobalConstant

This constant indicates that synthesis methods should treat this type as a river. The terrain type may be either a border or a connection.

`valley-x` GlobalConstant

This constant indicates that synthesis methods should treat this type as a valley.

`road-x` GlobalConstant

This constant indicates that synthesis methods should treat this type as a road.

`liquid` *t/f* TerrainTypeProperty

This property is true if the terrain type represents a liquid, which means that adjacent cells of liquid must have the same elevation. Defaults to `false`.

4.13.2 Terrain Compatibility

Terrain types are not always mutually compatible. Incompatible types may not be juxtaposed, either at game setup time or by unit action during a game.

`adjacent-terrain-effect` *t1 t2 -> t3* Table

This table specifies what will happen to a cell of type *t1* adjacent to a cell of type *t2*. If *t3* is `non-terrain`, nothing will happen, otherwise it will become a cell of type *t3*.

If *t1* is a border type adjacent to a cell of type *t2*. If *t3* is `non-terrain`, nothing will happen. Otherwise, the border of type *t1* will be removed, and if *t3* is a border type, a border of that type will be added. The effect on connection types is analogous. Defaults to `non-terrain`.

4.13.3 Other Terrain Properties

`elevation-min` *dist* TerrainTypeProperty `elevation-max` *dist* TerrainTypeProperty

These properties define the minimum and maximum possible values for the elevation in a cell of given terrain type. Both default to `0`.

`temperature-min` *n* TerrainTypeProperty

`temperature-max` *n* TerrainTypeProperty

These properties define the minimum and maximum possible values for the temperature in a cell of given terrain type. Both default to `0`.

`wind-force-min` *n* TerrainTypeProperty

`wind-force-max` *n* TerrainTypeProperty

These properties define limits on wind force. Both default to `0`.

`clouds-min` *n* TerrainTypeProperty

`clouds-max` *n* TerrainTypeProperty

These properties define limits on cloud density. Both default to `0`.

4.14 Material Types

Materials are materials that are manipulated in mass quantities. In general, material types just index vectors of values attached to other objects, such as unit supplies.

No more than 126 types of material may be defined.

`material-type` *symbol properties...* Form

This form defines a new type of material, named by *symbol*. Details are similar to those for unit types.

`m*` GlobalVariable

This variable evaluates to a list of all material types, listed in the same order as they were defined.

`non-material` GlobalVariable

This variable has a value that is never a material type.

4.14.1 People

A material type can be designated as representing people.

`people` *n* MaterialTypeProperty

This property is the actual number of individuals represented by 1 of a material. If 0, then the material type does not have people associated with it at all. Defaults to `0`.

Multiple types of materials can represent different types of people, so for example there could be one type `nomad` with 10 people/material, and another type `urbanite` with 10,000 people/material.

The basic cell capacities for materials also constrain people materials. There can be an additional limit on the number of individuals.

`people-max` *n* TerrainTypeProperty

This property is the maximum number of individuals allowed in a cell of this type of terrain. This is checked at the end of each turn; any excess will be moved into adjacent cells or disappear entirely. Defaults to `-1`, which allows any number of people in a cell.

4.15 Static Relationships Between Types

In general, static relationships are those that must always hold during a turn. *Xconq* will usually only test these when necessary, but this is up to the implementation. From the players' and designers' point of view, these relationships can never be violated, even temporarily.

4.15.1 Occupants and Transports

A unit inside another unit is an "occupant" in a "transport", even if the "transport" can never move. There are two kinds of capacity. Generic capacity is shared by all different types, while guaranteed capacity is for a particular type only.

`capacity` *n* UnitTypeProperty

This property is the limit on the sum of sizes of units that may occupy this type of unit, not counting the exclusive capacities. Defaults to `0`.

`unit-size-as-occupant` *u1 u2* -> *n* Table

This table is the "size" of a (full-sized) unit *u1* when it is in a transport *u2*. Defaults to `1`.

`unit-capacity-x` *u1 u2* -> *n* Table

This table is the number of units of type *u2* that are guaranteed a place in a unit of type *u1*. Defaults to `0`.

`occupant-max` *u1 u2* -> *n* Table

This table is the upper limit on the number of occupants of this type (not counting `unit-capacity-x`). Defaults to `0`.

`occupant-total-max` *n* UnitTypeProperty

This property is the upper limit on occupants of all types together. Defaults to `-1`, which allows unlimited occupancy.

A unit that is an occupant may not always have the same capabilities as when it is out in the open. Its vision, combat, construction, and capacity may be affected.

`occupant-vision` *u1 u2* -> *t/f* Table

Defaults to `true`.

`occupant-combat` *u1 u2* -> *n%* Table

This table defines the effect on the combat abilities of a unit of type *u1* when an occupant in a unit of type *u2*. If `0`, then the occupant cannot attack or fire. Defaults to `100`.

`occupant-can-construct` *u1 u2* -> *t/f* Table

This table is `true` if *u1* can create or complete units while an occupant of *u2*. Defaults to `false`.

`occupant-can-have-occupants` *u1 u2* -> *t/f* Table

This table is `true` if *u1* can have occupants of its own while an occupant of *u2*. Defaults to `false`.

4.15.2 Units and Terrain

This section describes relationships between units and terrain. Units can be set to disappear or be wrecked on particular types of terrain. If the terrain can be occupied safely, there may be a limit on the numbers of units that can be in the same cell.

`vanishes-on` *u t* -> *t/f* — Table

This table is `true` if a unit *u* will disappear instantly if it somehow ends up on terrain of type *t*. Defaults to `false`.

`wrecks-on` *u t* -> *t/f* — Table

This table is `true` if a unit *u* will wreck instantly if it somehow ends up on terrain of type *t*. Defaults to `false`.

`capacity` *n* — TerrainTypeProperty

This property is the limit on the sum of unit sizes that may share this cell. Defaults to `1`.

`unit-size-in-terrain` *u t* -> *n* — Table

This table is the "size" of a (full-sized) unit *u* when it is in/on the terrain *t*. Defaults to `1`.

`terrain-capacity-x` *u t* -> *n* — Table

This table is the number of (full-sized) units of type *u* that are guaranteed to have a place in the cell. Defaults to `0`.

Note that the units' sides are irrelevant; the sizes of units of all sides are added together. Limits are calculated separately for the connection and open terrain in a cell.

`stack-order` *n* — UnitTypeProperty

This property is the relative position of this type of unit within a stack of different units. Larger values put units higher in the stack. The exact values are unimportant, they are just used as sort keys. The use of this value is to ensure that particular types are "seen first" when looking at a cell, so for instance if a truck and a city are stacked on the same cell, everybody will see the city and not the truck. The owner of these units can still see them. If the stack-order of two units is the same, then the higher-numbered type will be higher in the stack. Defaults to `0`.

There is a possible bizarrity with stacking limits and units that can't see each other when in the same hex, namely that a player could be prevented from moving a unit into a cell that looks like it has enough room.

4.15.3 Units and Materials

Units can carry materials. As with occupants, there is both a generic storage space and spaces specialized for each material type.

`unit-storage-x` *u m* -> *n* — Table

This table is the space reserved specifically for each type of material. Defaults to `0`.

Materials that represent people may surrender to a unit in their cell.

`people-surrender-chance` *u t* -> *n*% — Table

This table is the base chance that people in terrain of type *t* will change sides if a unit of type *u* is in their cell. Defaults to `0`.

`people-surrender-effect` *u m* -> *n* — Table

This is a multiplier that takes the people type into account. Defaults to `100`.

4.15.4 Terrain and Materials

`terrain-storage-x` *t m* -> *n* — Table

This table is the amount of a material *m* that can be accumulated in a cell with terrain *t*. Defaults to `0`.

4.16 Vision

`see-all` *t/f* — GlobalVariable

This variable is `true` if everything in the world, units, terrain, etc, is always visible at all times, including initially. It takes precedence over *all* other visibility and spying parameters. Defaults to `false`.

`see-terrain-always` *t/f* GlobalVariable

If this variable is `true`, then any side that has seen the terrain of a cell will be informed if that terrain ever changes. Defaults to `true`.

`see-always` *t/f* UnitTypeProperty

This property is `true` when a unit is always visible after it has been seen once, so that side changes, movements, etc will be seen forever afterwards. If the unit moves into terrain that has not been seen, then that terrain also becomes seen as well. Defaults to `false`.

`see-occupants` *t/f* UnitTypeProperty

This property is `true` when a unit's occupants are also seen whenever the unit itself is under observation. Defaults to `false`.

`spot-action` *t/f* UnitTypeProperty

If this property is `true`, then the unit's chance to be seen by other sides will be tested each time the unit acts in any way. This property is in addition to the check at the beginning of each turn. Defaults to `true`.

The people in a cell effectively view (for their side) all units in that cell. Some units can hide from the people.

`people-see-chance` *u m* -> *n*% Table

This table is the chance that the people of the given type *m* will see a unit of type *u*. This will be evaluated for each people type individually, once at the beginning of each turn, and once for each populated cell that the unit enters during the turn. Defaults to `100`.

`vision-range` *dist* UnitTypeProperty

This property is the maximum range of vision coverage by the unit. A value of `-1` disables all vision, `0` means only units in the same cell may be seen, and `1` means units in adjacent cells may be seen. Defaults to `1`.

`see-chance-at` *u1 u2* -> *n*% Table

`see-chance-adjacent` *u1 u2* -> *n*% Table

`see-chance` *u1 u2* -> *n*% Table

All default to `100`.

`visibility` *u t* -> *n* Table

Defaults to `100`.

`vision-night-effect` *u t* -> *n* Table

This table is the multiplier for unit *u*'s vision at night in each type of terrain *t*. Effect is to multiply with both vision range and see-chance. Defaults to `100`.

4.16.1 Weather Vision

`see-weather-always` *t/f* GlobalVariable

If true, then weather changes (in cells that have been seen) will always be reported. Defaults to `true`.

4.16.2 Line of Sight

`vision-bend` *n* UnitTypeProperty

This property is the amount by which a unit can see "around corners". 0 means that vision is strictly line-of-sight, while 100 means that elevations never obstruct vision. Defaults to `100`.

`eye-height` *u t* -> *dist* Table

This propety is the additional elevation above the unit's position that a unit can see with, when in the given terrain. Defaults to `0`.

`thickness` *dist* TerrainTypeProperty

This property is the thickness of the terrain, which is the difference between the "ground" of the terrain and its top. Defaults to `0`.

4.16.3 Spying

A unit type can also be specified to do spying automatically. The outcome of spying is calculated once/unit/turn, at the beginning of the turn (after move calculation but before any players can do anything). Spying can happen to any unit not on the spying unit's side.

`spy-chance` *.01n%* UnitTypeProperty

This property is the chance that the unit's spies will find out something. Defaults to `0`.

`spy-range` *dist* UnitTypeProperty

This property is the maximum distance at which the unit's spies will find out something. Defaults to `0`.

`spy-quality` *u1 u2 -> n%* Table

This table gives the chance that *u1*'s spies will return information about a unit of type *u2*. Defaults to `100`.

4.17 Game Initialization and Naming

Game initialization always starts by resetting all the game-defining data structures to an empty state. This means no types, no world, etc. Then *Xconq* reads and interprets all of the game modules that have been requested. These modules may overwrite each other arbitrarily. Then any command line or startup options are processed (this may involve an interactive dialog), and the random number generator is initialized. and players are matched with sides (any sides needed for players will be created and named at this time). *Xconq* then executes a number of *synthesis methods* to do various kinds of setup.

(Some interfaces might allow for confirmation of the setup before launching into the game proper, but this cannot be assumed.)

Since the details of good game synthesis can be complicated, synthesis methods are simply wired-in pieces of code. Each method is self-contained; it assumes the game state to be valid, it will determine its own applicability and produce a valid result. It will also acquire any data that it needs, so does not require any special setup; however, a method may fail to run if it cannot find that data. For instance, the usual fractal terrain generator needs percentiles for each terrain type, and will not function without them. It may be that all the requested synthesis methods fail; this is OK if *Xconq*'s data is present and consistent, but otherwise *Xconq* will shut itself down, since it has no remaining alternatives (think of this as a serious programming error and fix the game design).

4.18 The Synthesis Method List

The synthesis method list specifies which methods will be run, and in what order. After they have all been run, *Xconq* runs a consistency and completeness check. For instance, there should be a world with terrain everywhere. Failure at this point is fatal; *Xconq* will either exit or return to a game setup dialog.

`synthesis-methods` *method-list* GlobalVariable

This variable is a list of synthesis methods. If the list is empty, no synthesis methods will be run.

The list of synthesis methods is ordered, and many contain duplicates, so that a method can be run multiple times during setup. Note that most of the existing methods will simply return if they detect that their work has already been done, so multiple runs will have no effect.

The default synthesis method list is

```
(make-fractal-percentile-terrain
 make-countries
 make-independent-units
 make-roads
 make-rivers
 init-supplies
 name-geographical-features
)
```

The synthesis method list may also contain items of the form

```
("program" forms...)
```

For each of these items, *Xconq* will attempt to find and run an external program named `"program"`, giving it as input the result of evaluating the `forms`, and then reading the output of the program, which must be a valid game module. The program must be capable of interpreting two arguments; the first is the name of the input file it is to read from, and the second is the name of the output file it must write to. If successful, it should return with a result code of 0; otherwise, *Xconq* will issue a warning to players.

Any further details will depend on your system, since each will use different conventions. Note that this is NOT a portable construct; you cannot assume that everybody will have built and installed the program you're using.

4.18.1 Fractal World

The fractal world synthesizer can make a variety of natural-looking terrain. It relies on a number of parameters to govern a single algorithm.

`make-fractal-percentile-terrain` SynthesisMethod

This method generates the terrain layer of a world. It works by generating two distinct layers of random blobs, known as the "alt" and "wet" layers, then decides on a terrain type for each cell. If elevations are defined, then this method will use the "alt" layer to produce elevations.

`alt-blob-density` *n* GlobalVariable

`wet-blob-density` *n* GlobalVariable

These variables are the number of blobs to put down, expressed as number per 10,000 cells. Defaults to `500`.

`alt-blob-size` *n.f%* GlobalVariable

`wet-blob-size` *n.f%* GlobalVariable

These variables are the average number of cells in a blob, expressed as number per 10,000 cells. Defaults to `100`.

`alt-blob-height` *n* GlobalVariable

`wet-blob-height` *n* GlobalVariable

These variables are the amounts by which to increment or decrement within a blob. Defaults to `1000`.

`alt-smoothing` *n* GlobalVariable

`wet-smoothing` *n* GlobalVariable

These variables specify the number of averaging steps to perform after the blobs have been generated. Defaults to `2`.

`alt-percentile-min` *n%* TerrainTypeProperty

`alt-percentile-max` *n%* TerrainTypeProperty

`wet-percentile-min` *n%* TerrainTypeProperty `wet-percentile-max` *n%* TerrainTypeProperty

These properties are the percentiles of elevations and moistures that result in the given terrain type. Percentile ranges may overlap, in which case the earlier-defined terrain type will be used. If a cell has a alt and wet that does not fall in any of the ranges, then terrain type 0 will be used there and players will be warned. Mins defaults to `0`, maxes to `100`.

4.18.2 Maze World

A maze consists of a set of randomly placed "rooms" connected by random passages.

`make-maze-terrain` SynthesisMethod

This method creates terrain that looks like a maze. It starts by randomly assigning terrain according to its `occurrence`, similarly to `make-random-terrain` below, then carves out rooms and passages, filling each of those with terrain types according to their respective occurrences.

`maze-room-occurrence` *n* TerrainTypeProperty

This property is the weighted amount of this terrain type in rooms in the maze. Defaults to `0`.

`maze-passage-occurrence` *n* TerrainTypeProperty

This property is the weighted amount of this terrain type in passageways in the maze. Defaults to `0`.

`maze-room-density` *n* GlobalVariable

This variable is the fraction of the maze that is room, expressed as the number of cells per 10,000 cells in the area. Defaults to `1000`.

`maze-passage-density` *n* GlobalVariable

This variable is the fraction of the area that is passageway, expressed as the number of cells per 10,000 cells in the area. Defaults to `3000`.

4.18.3 Random World

The random world generator just assigns terrain and elevations randomly.

`make-random-terrain` SynthesisMethod

This method generates completely random terrain. It uses a simple weighting to govern how much of each terrain type appears, and makes random elevations as well.

`occurrence` *n* TerrainTypeProperty

This property is the percentage of the world that will be of this type. Defaults to `1`.

4.18.4 Earthlike World

Earthlike generation uses algorithms that more closely approximate realistic terrain.

`make-earthlike-terrain` SynthesisMethod

This method generates terrain that approximates what actually appears on Earth.

4.18.5 River Generation

Rivers are borders or connections consisting of "watery terrain" that run downhill to regions of water.

`make-rivers` SynthesisMethod

This method looks for a border or connection terrain type with a `subtype-x` of `river-x`. then uses the world's elevation data to run rivers downhill (always choosing the lowest of possible adjacent locations) until they reach cell terrain with a `subtype` > 0. This method will not run if there are no appropriate terrain types, nor if there is no elevation data.

`river-chance` *n*% TerrainTypeProperty

This property is the chance that a river will start in or around a cell of this terrain type. Defaults to `0`.

`river-sink-terrain` *t* GlobalVariable

If the value of this variable is a terrain type, then a cell completely surrounded by river will be changed to be this type. Defaults to `non-terrain`.

Note that the algorithm computes rivers in a deterministic way, so high values of `river-chance` do not result in tangled rivers.

4.18.6 Road Generation

The road generation method makes networks of connection terrain between particular unit types, usually those resembling cities.

`make-roads` SynthesisMethod

This methods synthesizes roads for an area. For any connection type of terrain, if no layer has been created for it already, and the type has a `subtype-x` of 3, put down roads between any pair of units whose `road-chance` is nonzero. The method will attempt to share road routes whenever possible, and choose terrain according to `road-into-chance`.

`road-chance` *u1 u2* -> *n%* Table

This table is the chance that a road will be laid, running from a unit of type *u1* to one of type *u2*. This is not a symmmetrical relationship. Defaults to `0`.

`road-into-chance` *t1 t2* -> *n%* Table

This table is the chance that a road will be chosen to pass from terrain of type *t1* into terrain of type *t2*. Defaults to `100`.

4.18.7 Making Countries

The `make-countries` method sets up the starting units for each side, placing them in a confined area, separated from the starting units of other sides and taking terrain preferences into account. If requested, this method will also expand the country outwards by a specified amount, possibly placing additional units in the process.

`make-countries` SynthesisMethod

This method works by looking for a likely place for the country, randomly places a basic set of starting units within that area, then expands the country outwards. The parameters give you control over the mix of terrain types in the country, as well as the size and relative positions of the different countries. This method runs on any side with fewer units than it is supposed to start with, as given by the parameters below. It places groups of units at locations separated from each other by specified distances.

`country-radius-min` *dist* GlobalVariable

This variable is the radius of the country's initial area. Defaults to `-1`, which allows the algorithm to calculate a "reasonable" country size appropriate to the given number of units.

`country-separation-min` *dist* GlobalVariable

`country-separation-max` *dist* GlobalVariable

These variables are the minimum and maximum distances of country centers from each other, in cells. If small, countries will mostly overlap; if very large, then attempts to use small worlds will fail; if the max and min are too close to each other, placements can also fail. For both of these, a value of `-1` disables their effect. Both default to `-1`.

The max separation bound needs to be satisfied for a country with respect to only *one* other country, so for instance the final layout may involve a long "string" of countries where the first and last countries are very far apart from each other. The minimum bound must be satisfied for all pairs of countries.

`country-terrain-min` *n* TerrainTypeProperty

This property is the minimum amount of terrain that must be within the country's initial radius. Defaults to `0`.

`country-terrain-max` *n* TerrainTypeProperty

This property is the most terrain of the given type that may appear. If `-1`, then any amount may be present. Defaults to `-1`.

`start-with` *n* UnitTypeProperty

`independent-near-start` *n* UnitTypeProperty

These properties set the number of units of the given type in a player's country. These units are randomly scattered within the initial radius, and the `favored` table (see below) decides which terrains will be used. Units may be placed inside each other; in fact, units with no favored terrain will be made into occupants if possible.

The independent units will be placed after the ones belonging to the side, so on the average they will get the less desirable locations in the country. Both independent and the side's units will be named using the side's namers.

Both default to `0`.

`favored-terrain` *u t* `->` *n%* Table
This table sets the probability of the unit type being on the given type of terrain at the outset. A value of `0` is an absolute prohibition against placing the unit on that type of terrain, thus every game must specify at least one non-zero value for some terrain type and some initial unit type. Defaults to `100`.

Once the initial country area has been set up, then you can allow the countries to expand outwards. Expansion occurs at the same rate for all countries. Countries may expand into and through each other.

`country-growth-chance` *n%* TerrainTypeProperty
This property is the chance that a country will expand onto an unclaimed cell of the given terrain type. Defaults to `100`.

`country-takeover-chance` *n%* TerrainTypeProperty
This property is the chance that a country will expand onto another country's cell of the given terrain type. Defaults to `0`.

`unit-growth-chance` *n.f%* UnitTypeProperty
This property is the chance that a unit of the given type will be placed when the country expands onto a cell. The unit will only be placed if the `favored` chance is also true. Defaults to `0`.

`independent-growth-chance` *n.f%* UnitTypeProperty
This property is the chance that an independent unit of the given type will be placed when the country expands onto a cell. The `favored` chance is also evaluated. Defaults to `0`.

`unit-takeover-chance` *n.f%* UnitTypeProperty
This property is the chance that a unit of the given type in another country and belonging to another side will be given to the growing side. Defaults to `0`.

`independent-takeover-chance` *n.f%* UnitTypeProperty
: This property is the chance that an independent unit of the given type in another country will be given to the growing side. Defaults to `0`.

`country-radius-max` *dist* GlobalVariable
: This variable is a cap on the country growth process. Values between `0` and `country-radius-min` prevent country growth entirely, while a value of `-1` allows growth to encompass the entire world. Defaults to `0`.

`country-units-max` *n* UnitTypeProperty
: This property is a cap on the number of units given to the side's country. Defaults to `-1`, which disables any limit.

`growth-stop-chance` *n%* GlobalVariable
: This variable is the chance that a country's growth will stop, if during the current [ring or round] no new cells were added to the country. Defaults to `0`.

`country-people-chance` *n%* TerrainTypeProperty
: This property is the chance that the people's side will be changed to match that for the country they are in. Defaults to `100`.

4.18.8 Making Independent Units

`make-independent-units` SynthesisMethod
: This method scatters independent units randomly over the world. This method will not run if the specified density of independent units has already been achieved, for instance from a predefined world or from country placement. Independent units that should be inside other independents will be handled correctly.

`independent-density` *u t -> n* Table
: This table is the total number of independent units appearing throughout the world, at the rate of *n* per 10,000 cells of the given terrain type. Any independent units already placed are counted first, so this value represents final density. If the sum of values for a given unit type on all terrain types is nonzero, then at least one unit of that type will be placed, even if the world is very small (i.e. the calculation of numbers rounds up not down). Defaults to `0`.

This method uses the `favored-terrain` table as the chance that a given unit will be placed at a randomly-chosen position, and it will keep trying different positions until a suitable one is found.

`independent-people-chance` *.01n%* TerrainTypeProperty

This property is the chance that the people of a cell with this terrain type will be made independent. Deafults to `0`.

4.18.9 Initial Supply

By default, all units start out empty of materials. The supply initialization method gives each unit a starting supply, according to the stockpile tables.

`make-initial-materials` SynthesisMethod

This method fills unit and cell supplies to specified levels. It will fill all units in existence at the moment it runs, including reinforcements [and incomplete?] units. Similarly, all cells will be filled.

`unit-initial-supply` *u m* -> *n* Table

This table is the amount of each material that each unit will start out with. If the initial supply is greater than unit's capacity, then the unit will just be filled to capacity. Defaults to `0`.

`terrain-initial-supply` *t m* -> *n* Table

This table is the amount of material *m* that each cell with terrain *t* will start out with. This will be limited by the cell's capacity. Defaults to `0`.

4.18.10 Naming Geographical Features

Although named geographical features don't affect the outcome of a game in any way, they are useful for "color" and for identifying locations more readably.

`name-geographical-features` SynthesisMethod

This method identifies and names regions as geographical features, such as mountain ranges and islands.

`feature-namers` *feature-namer-list* GlobalVariable

This variable is a list of feature types and their associated namers. This is used for features not intersecting any country with a namer for the feature's type. Defaults to `()`.

`feature-types` *feature-expr-list* GlobalVariable

This variable is a list of feature types that may be identified. Examples: `("lake" (group (sea shallows) 1))`, `("peak" (high-point 1 1))`

Defaults to `()`.

4.18.11 Naming Units

`name-units-randomly` SynthesisMethod

This method gives names to previously-unnamed units, using their usual [?] naming methods.

4.18.12 Making a Random Date

`make-random-date` SynthesisMethod

[how is this controlled?]

4.19 Setup Postprocessing

Some initialization steps will be done after all synthesis methods have been run. *Xconq* will always do these.

4.19.1 Initial View

By default, each side starts out knowing only what its units can normally see at the beginning of the first turn. These parameters add to that initial view.

`terrain-seen` *t/f* GlobalVariable

This variable is `true` if all the terrain of the world is known initially. Defaults to `false`.

`initial-seen-radius` *dist* UnitTypeProperty

This property specifies the radius of the area seen around each of the starting units. It computes visibility of terrain (cells and borders) only. Defaults to `1` (which is a no-op if the unit's `vision-range` is greater than or equal to 1).

`already-seen` *n%* UnitTypeProperty

This property is the chance to see units of this type at the beginning of the game. This applies only to units belonging to another side, and on known terrain. The effect is one-time, so if an `already-seen` unit changes sides later on, other players will not see the change unless they have the unit under observation for themselves. Note that `see-always` implies `already-seen`. Defaults to `0`.

`already-seen-independent` *n%* UnitTypeProperty

This property is like `already-seen`, but applies to independent units specifically. Defaults to `0`.

4.20 Naming and Text Generation

Xconq can generate names for sides, units, and geographical features.

4.20.1 Naming Sides

Side naming is special, because several different but related names have to be produced.

`side-library` *side-info...* Variable

This variable is a weighted list of groups of side properties, each of which may be used to fill in a side.

The form of each side name entry is basically a subset of the side's properties:

```
([weight] ... (name "name") ... (color-scheme "colors") ...)
```

Each entry can include as many or as few of the attributes as desired; any missing will be filled in from the usual defaults. The optional *weight* is a number that adjusts the probability of selection of the given side name set; it defaults to 1, and the probability is scaled according to the sum of the weights for all the sides listed. If any property value is a namer, then the namer will be run. (Note that if multiple namers are specified, they cannot be guaranteed to coordinate with each other, so you can end up with a side noun that is inappropriate for its corresponding side name.)

4.20.2 Namers

Since one of the purposes of naming is to identify objects uniquely, any name generator should be able to maintain some memory as to what has been generated already. The objects that do this are *namers*.

`namer` [*symbol/id*] *method rejects...* Form

This form defines an instance of a namer, with either the symbolic name or numeric id. If either matches the name or id of an existing namer, then the old namer will be overwritten, otherwise a new one will be created. The *method* must be one of the naming methods listed below, and *rejects* defines what names may not be produced (its exact interpretation depends on the method).

4.20.3 Naming Methods

As with general synthesis, *Xconq* has a number of *naming methods* available.

An implementation is free to define additional naming methods.

`random` *names......* NamingMethod

This method picks a name from the given list of names and removes that name from the list

`junky` NamingMethod

This method produces a gobbledy-gook name, very techy-looking.

`grammar` *root max-length rules...* NamingMethod

This method defines a grammar, where *root* is the root symbol, *max-length* is a limit on the length of the generated names (in characters), and *rules* is a list of rules of the form

(*symbol* (`[sym]` `[weight]` *symbol/string/list* `[n]` `...`))

The generation process works by substituting one of the rule's alternatives for the symbol, starting with the root symbol. The probability of an alternative being selected is arrived at by adding up the optional weights *weight* (assuming missing weights to be `1`), and choosing with a probability of the weight divided by the total sum of weights. Thus the weights need not add up to any particular value.

Strings get used directly. If a symbol in the rule's chosen expansion does not appear as the lefthand side in any rule, then it will be handled as a string, otherwise it will be expanded in turn. If the symbol matches a namer's name, then that namer will be run (passing the same object??) and its result incorporated. A list should be a list of strings and symbols, and the expansion of each will be concatenated.

`any` GlobalConstant

[???]

`or` GlobalConstant

`reject` GlobalConstant

A special rule headed by `reject` is a list of substrings that should not appear in a generated name; this is a convenient way to filter out particularly unlovely results.

`capitalize` GlobalConstant

Directs capitalization of a nonterminal.

[text is not actually different from a namer?]

`text` [*symbol/id*] *method rejects...* Form

[elsewhere?]

`action-messages` *patterns* GlobalVariable

`event-messages` *patterns* GlobalVariable

4.21 Other Initialization Controls

`edge-terrain` GlobalVariable

This variable is the type of terrain to fill in on all the edges of a world. The edges of a world have little or no effect on the game, but the terrain type should be something distinctive, so that players can recognize the edges easily. (For instance, ice is usually a good choice for edges, but probably not on a map of Antarctica!)

4.22 Actions in General

The parameters in this chapter define and regulate the various actions that are available to units during a game.

Actions are always started and completed (including all of their effects) within the same turn, and a unit can only do one of them at a time.

All actions are in theory available to all units, but the parameters can be set so as to deny any action type to any unit type. See the descriptions with each action type.

All action is limited by action points. Each unit gets a certain number at the beginning of each turn and expends them in the course of doing things. The usual expenditure is one point per action, but may be more, as defined for each type of action. A unit action must always consume at least one action point.

Units can accumulate acp from turn to turn, and they can also reduce acp below zero.

`acp-per-turn` *acp* UnitTypeProperty

This property is the basic allowance of action points that a unit gets each turn. Defaults to `1`.

`acp-min` *acp* UnitTypeProperty

This property specifies how far into "action debt" a unit can go during a turn before it is prevented entirely from acting. A unit with acp `<` 1 at the beginning of a turn cannot do anything at all. Defaults to `0`.

`acp-max` *acp* — UnitTypeProperty

This property is the maximum number of action points that a unit can save up. The value `-1` means that `acp-max` is equal to `acp`. Extra acp is silently lost. Defaults to `-1`.

`free-acp` *acp* — UnitTypeProperty

This property is the value is the amount by which the action points for some action can exceed the unit's currently available acp and still allow that action. Defaults to `-1`, which means enough free acp to allow any action.

Note that a unit with an acp of 0 is completely unintelligent, about like a cow patty. Cow patties can be useful for blocking paths, hiding behind, and suchlike, and have the advantage that once they're in place, you don't have to manage them. Other units will have to pick them up and put them down, of course.

`material-to-act` *u m* -> *n* — Table

This table is a minimum amount of *m* needed for *u* to be able to act. The material is not consumed. Defaults to `0`.

`acp-damage-effect` *xxx* — UnitTypeProperty

`acp-night-effect` *u t* -> *n* — Table

This table is the multiplier for unit's acp at night in each type of terrain. Defaults to `100`.

`acp-occupant-effect` *u1 u2* -> *n* — Table

Defaults to `100`.

`acp-per-turn-min` *acp* — UnitTypeProperty

`acp-per-turn-max` *acp* — UnitTypeProperty

This property limits on effect of occupants, damage, etc. Defaults to `1`.

4.22.1 Action Ordering

`use-side-priority` *t/f* — GlobalVariable

This variable is `true` if the sides may only act one at a time; otherwise, all sides and units may move simultaneously during a turn. Defaults to `false`.

`action-priority` *n* UnitTypeProperty

This property is the order in which units of this type will act. Higher numbers act earlier. If the difference between the priority of one type and another is greater than 100, then the earlier-acting units must finish acting before the later-acting units, otherwise a player can rearrange the actual acting order as desired. Defaults to `0`.

4.22.2 Movement

Movement is the most common sort of action. This section covers movement over open terrain; the next section discusses interaction with transports.

The general theory of movement is that a unit not in a transport crosses its current cell terrain to the edge of the cell, crosses any border terrain, and then moves into the destination cell, OR it moves onto connection terrain, travels along connection terrain to the new cell, and maybe moves off the connection. If the unit starts in a transport, then the transport may ferry the unit over some of the intervening terrain, possibly as far as the unit's destination.

A unit's basic movement rate is defined by its *speed*, which is a ratio of the the unit's acp. A speed of 100% means that the unit can potentially enter as many cells as it has acp, while a speed of 20% means that the unit uses at least 5 acp to enter a cell.

Movement can only succeed if several conditions are met: the unit must be able to cross the border terrain, the destination must be inside the world (but see below), it must be able to exist on the terrain of the destination.

`move` *x y z* ActionType

This is the action that a unit performs to go from one location to another. The destination must be within the `move-range` of the unit.

`acp-to-move` *acp* UnitTypeProperty

This property is the number of acp a unit uses to do one move action. Defaults to `1`.

`speed` *n* UnitTypeProperty

This property is the basic multiplier relating acp to the number of cells that may be entered during a turn. Defaults to `100`.

`speed-damage-effect` *list...* UnitTypeProperty

Defaults to `()`.

`speed-occupant-effect` *u1 u2 -> n%* Table

This table is the percent change in the speed of type *u1* for each occupant of type *u2*. If the basic speed of *u1* is `0`, then the multiplication is performed as if the speed were `1` instead. Defaults to `100`.

`speed-wind-effect` *xxx* UnitTypeProperty

`speed-wind-angle-effect` *xxx* UnitTypeProperty

`speed-min` *mp* UnitTypeProperty

This property is the worst-case speed of a unit. Defaults to `0`.

`speed-max` *mp* UnitTypeProperty

This property is the upper bound on a unit's movement in one turn. Defaults to `0`.

`move-range` *n* UnitTypeProperty

This property is the maximum distance allowed to the destination cell. Defaults to `1`.

The product of a unit's acp and its speed is its available *movement points*. Any move between cells will cost at least one movement point. Some mp costs may be negative, but the total mp for a move will always be at least 1.

`mp-to-leave-terrain` *u t -> mp* Table

This table is the mp cost to leave a cell of type *t*. If *t* is a border type, this cost is never used. If *t* is a connection type, this cost is the cost of leaving the connection terrain for the open terrain of the cell. If *t* is a coating type, then this value adds to the cost of leaving the cell. Defaults to `0`.

`mp-to-enter-terrain` *u t -> mp* Table

This table is the mp cost to enter a cell of type *t*. If *t* is a border type, this cost is the cost of crossing the border. If *t* is a connection type, this cost is the cost of entering the connection terrain from the open terrain of the cell. If *t* is a coating type, then this value adds to the cost of entering the cell. Defaults to `1`.

`mp-to-traverse` *u t* -> *mp* — Table

This table gives the cost to travel along a connection or border of the given type. (note that the other costs are irrelevant if unit starts and ends its movement on the connection).

A special type of move known as a *border slide* can occur when the endpoints of a border touch on the start and destination cells. Sliding works like normal movement that happens to end up on a nonadjacent cell. Same rules for permissibility apply. If the value is negative, then border sliding is not possible.

Defaults to `1`.

If both enter/traverse/leave and enter/leave movement is possible, then *Xconq* will automatically choose the cheapest alternative.

Each unit type has a range of altitudes within which it normally operates.

`altitude-min` *u t* -> *n* — Table

This table is the minimum altitude possible for each type of unit on each type of terrain. Defaults to `0`.

`altitude-max` *u t* -> *n* — Table

This table is the maximum altitude possible for each type of unit on each type of terrain. Defaults to `0`.

`mp-to-leave-world` *mp* — UnitTypeProperty

This property is an additional move cost to leave the world entirely. To leave, the unit must be within its `move-range` of an edge, and have sufficient mp to move into the terrain in the edge cell designated as the destination of the move. If the value is `-1`, then the unit may never leave. Defaults to `-1`.

`free-mp` *mp* — UnitTypeProperty

This property is the amount by which the move points can "go into the red" and still allow one more move. Defaults to `0`.

ZOC is exerted only over units out in the open, has no effect on occupants, unless they leave their transport. Occupants can themselves exert a ZOC, if `occupant-can-fight` is true. ZOC applies to all units on a hostile side.

`zoc-range` *u1 u2* -> *dist* Table
: This table is the maximum distance at which type *u1* exerts a ZOC over type *u2*. A value of `0` means that the unit controls only its own cell, and a value of `-1` means that the unit does not exert a ZOC at all. Defaults to `0`.

`zoc-into-terrain` *u t* -> *t/f* Table
: This table is `true` if the unit exerts its ZOC into terrain *t*. Defaults to `true`.

`zoc-from-terrain-effect` *u t* -> *n* Table
: Defaults to `100`.

`mp-to-enter-zoc` *u1 u2* -> *mp* Table
: This table specifies extra movement points needed to enter the ZOC. `-1` prevents entry entirely. Defaults to `-1`.

`mp-to-leave-zoc` *u1 u2* -> *mp* Table
: This table specifies extra movement points needed to leave the ZOC. `-1` prevents departure entirely. Defaults to `0`.

`mp-to-traverse-zoc` *u1 u2* -> *mp* Table
: This table specifies extra movement points needed to move within the ZOC. `-1` prevents traversing entirely. Defaults to `0`.

If multiple units exert a ZOC into the same cell, then the mp cost is the maximum of the different ZOC costs.

Units may use up some of their materials when they move. Consumption happens after the move action, and only for successful moves.

`material-to-move` *u m* -> *n* Table
: This table is the amount of each material that a unit of type *u* must have in order to be able to move. Defaults to `0`.

`consumption-per-move` *u m* -> *n* Table

This table is the amount of each material used by a unit to do one move action. The amount taken is independent of terrain. If the unit has less than the required amount of any of these materials, it is immobilized until it gets more (this is tested before each move action; note that this does not affect any other action, including entering and leaving transports). Defaults to `0`.

4.22.3 Entering and Leaving Transports

Units can be inside other units, and have units inside them, in a tree-like fashion. There is no limit on the depth of the tree, but most occupant-transport relationships have other limits.

`enter` *unit* ActionType

This is the action to enter the given *unit*.

`acp-to-enter-unit` *acp* UnitTypeProperty

This property is the number of acp a unit uses to do one entry action. Defaults to `1`.

`can-enter-independent` *u1 u2* -> *t/f* Table

This table is true if a unit *u1* can enter an independent unit *u2*. Defaults to `false`.

Entering and leaving incur mp costs as does movment, but units with a speed of 0 may enter and leave transports.

`mp-to-enter-unit` *u1 u2* -> *n* Table

This table is the extra movement points required for *u1* to enter the transport *u2*, and vice versa (i.e. how much of transport's time is consumed by the process). Defaults to `0`.

`mp-to-leave-unit` *u1 u2* -> *n* Table

Similar to entry cost. Defaults to `0`.

Note that these mp consumptions need not be symmetrical between occupant and transport, so for instance a passenger can use 2 of its mp to get on a transport, while costing the transport only one of its mp.

`ferry-on-entry` *u1 u2* -> *ferry-type* Table

`ferry-on-departure` *u1 u2* -> *ferry-type* Table

This table specifies how much intervening terrain the unit *u2* entering or leaving transport *u1* will have to cross on its own (and thus incur the terrain's mp costs and limitations). Defaults to `over-border`.

`over-nothing` GlobalConstant

This constant indicates no ferrying, occupant must pay all costs to go to destination cell.

`over-own` GlobalConstant

This constant indicates that the transport ferries over terrain of its own cell.

`over-border` GlobalConstant

This constant indicates that the transport ferries over any border terrain also.

`over-all` GlobalConstant

This constant indicates that the transport ferries to destination cell, effectively putting occupant on middle of cell, on connection terrain if necessary.

4.22.4 Research

Research is an action performed by a unit with the sole effect of increasing its side's tech level. Research cannot be performed by independent units.

`research` *u* ActionType

This is the action of researching the unit type *u*. If the action is valid, then the tech level of the side will increase. Unit types with any tech crossover will also have their tech levels adjusted.

`acp-to-research` *acp* UnitTypeProperty

This property is the number of action points used up by one research action. Defaults to `0`, which disallows research.

`tech-per-research` *u1 u2 -> .01n* Table

This table is the gain in tech level resulting from a research action, expressed as 1/100 of a level. Gains of less than 100 are probabilistic [should describe this concept in general, used by several parms] Defaults to `0`.

`tech-per-turn-max` *tl* UnitTypeProperty

This property is a ceiling on the total gain of tech level possible in one turn for each side and this unit type. Defaults to `9999`.

4.22.5 Tooling Up

There are several stages in the construction of a unit: tooling up, creation, and completion. Tooling up is where the building unit prepares to build, creation is the step where the new unit comes into existence, and completion is where the new unit is brought up to being operational.

For the player, this is mostly automatic; if tooling must be done first, a user command to build will generate the appropriate actions.

Once the technology has been achieved, a unit that intends to construct other units may need to tool up. This is expressed as *tool points* or *tp*. Tool points start at zero, can be increased by tooling actions, and may gradually decline (representing wear and tear on the equipment).

`toolup` *u* ActionType

This is the action of tooling up to build a unit of type `u`. The result is an increase in the tp for the acting unit.

`acp-to-toolup` *acp* UnitTypeProperty

This property gives the number of acp needed to do a toolup action. Defaults to `0`, which disallows tooling up.

`tp-per-toolup` *u1 u2 -> tp* Table

This table is the number of tp gained by one tooling action. Defaults to `0`.

`tp-to-build` *u1 u2 -> tp* Table

This table is the number of toolup points needed before a unit of type *u1* can create or build a unit of type *u2*. Defaults to `0`.

`tp-max` *u1 u2* -> *tp* Table

This table is the maximum possible tooling. Defaults to `0`.

`tp-attrition` *u1 u2* -> *tp* Table

This table is the number of .01 tool points automatically lost at the end of each turn. Defaults to `0`.

`tp-crossover` *u1 u2* -> *n%* Table

This table is the effective number of tool points for *u2* that is guaranteed to exist, expressed as a percentage of the tool points for *u1*. [copy tech-crossover description here] Defaults to `0`.

4.22.6 Creating a Unit

When a constructing unit is tooled up, the build action creates a unit immediately and puts it in its designated location, whether inside the unit doing the building or somewhere nearby. This new unit, however, is incomplete, representing the keel of the ship or the surveyor's lines for an airstrip. Incomplete units are thus basically skeletons, with some unit characteristics, but unable to move or act in any way. They also cannot have any occupants, unless the occupants are of a type that can complete the unit. Those occupants do not derive any protection or other advantages from occupying the incomplete unit, and they are not affected by the `occupant-can-build` limitation.

`create-in` *u unit* ActionType

This action creates a new unit of type *u* occupying the given unit *unit*. The unit *unit* must have room for the new unit.

`create-at` *u x y z* ActionType

This action creates a new unit of type *u* in the open at *x,y,z*. The cell must have room for this new unit.

`acp-to-create` *u1 u2* -> *acp* Table

This table is the acp used by a unit of type *u1* to create a a unit of type *u2*. If zero, then *u1* cannot create a *u2*. Defaults to `0`.

`create-range` *u1 u2* -> *dist* Table

This table is the maximum distance at which a unit of type *u1* can create a unit of type *u2*. Defaults to `0`.

`cp-on-creation` *u1 u2* -> *cp* Table

This table is the completeness of a unit of type *u2* when created by a unit of type *u1*. Defaults to `1`.

`material-to-create` *u m* -> *n* Table

This table is the total amount of a material type *m* needed to create a unit of type *u*. Defaults to `0`.

`consumption-on-creation` *u m* -> *n* Table

This table is the amount of a material type *m* consumed to create a unit of type *u*. Defaults to `0`.

`supply-on-creation` *u m* -> *n* Table

This table is the amount of supply of each material type *m* to give a newly created unit of type *u*. This supply is newly generated, does not come from anywhere else. (Note that players could cheat by creating units, taking their supply, and never completing them.) Defaults to `0`.

4.22.7 Building a Unit

Once an incomplete unit has been created, other units can help to complete it.

`build` *unit* ActionType

This action adds to the completeness of *unit*. If the unit becomes complete, it will be given its initial supply, acp, name, etc.

`acp-to-build` *u1 u2* -> *acp* Table

This table is the acp used up by one build action by a unit of type *u1* when buiding a unit of type *u2*. Defaults to `0`.

`cp-per-build` *u1 u2* -> *cp* Table

This table is the amount of completeness of a unit of type *u2* added by each completion action performed by a unit of type *u1*. If `0`, then *u1* cannot contribute to completing *u2*. Defaults to `1`.

`material-to-build` *u m* -> *n* Table

This table is the amount of each material type *m* that *u* must have in order to build anything at all. Defaults to `0`.

`consumption-per-build` *u m* -> *n* Table

This table is the amount of each material type *m* consumed by *u* when doing a build action. Defaults to `0`.

`build-range` *u1 u2* -> *dist* Table

This table is the maximum distance allowed between a unit of type *u1* and the incomplete unit of type *u2* it is working on. Defaults to `0`, which requires the two units to be in the same cell.

At a given point, incomplete units can make progress towards completion on their own. This is automatic because incomplete units are unable to act, and occurs at a constant specified rate.

`cp-to-self-build` *cp* UnitTypeProperty

This property is the minimum completeness of the unit necessary before it can work on itself. Defaults to `0`.

`cp-per-self-build` *cp* UnitTypeProperty

This property is the completeness added each turn when a unit works on itself. Defaults to `0`.

`supply-on-completion` *u m* -> *n* Table

This table is the minimum amount of supply of each material type *m* guaranteed to a newly completed unit of type *u*. If not already available to the unit, it will be newly generated. Defaults to `0`.

4.22.8 Repair

Units can restore their own and each other's hp by doing repairs. Repair requires a repair action. The action points for this action are taken from both the unit being repaired and the repairer (using the same table `acp-to-repair`). When a unit repairs itself, the action cost is counted once only.

`repair` *unit* ActionType

This is the action of repairing the given *unit*.

`acp-to-repair` *u1 u2 -> acp* Table

This table is the number of action points used up by a unit of type *u1* doing one repair action on a unit of type *u2*. Defaults to `0`, which disallows the action.

`hp-per-repair` *u1 u2 -> .01hp* Table

This table is the hundredths of a hp that a single repair action by a unit of type *u1* restores to a unit of type *u2*. The fraction of this over 100 is added to hp directly, while the `<` 100 fraction is added probabilistically. (For example, a value of 160 means that 1 hp will be added for each action, and there is a 60% chance that a second hp will be added also.) Defaults to `0`.

Materials may be needed and/or consumed during repair. The materials will be taken from the unit being repaired, then from the repairer.

`material-to-repair` *u m -> .01n* Table

This table is the amount of each material type *m* needed for one repair action. As with `hp-per-repair`, the `<` 100 part is average, and `>` 100 is guaranteed. Defaults to `0`.

`consumption-per-repair` *u m -> .01n* Table

This table is the amount of each material type *m* used up by a repair action.

The repairing unit must also not be too damaged itself to do repairs.

`hp-to-repair` *u1 u2 -> hp* Table

This table is the minimum hp level required of a unit of type *u1* to repair a unit of type *u2*. If less, then *u1* is too damaged to do any repairing. Defaults to `1`, which allows repair always.

4.22.9 Producing Materials

Units can produce materials by explicit action.

`produce` *m* ActionType

This action results in a quantity of material *m* coming into existence.

`acp-to-produce` *u m* -> *acp* Table

This table is the acp used up by one `produce` action. Defaults to `0`.

`material-per-production` *u m* -> *n* Table

This table is the amount of material produced by *u* when acting to produce type *m*. Defaults to `0`.

`material-to-produce` *u m* -> *.01n* Table

4.22.10 Transferring Materials

Although most movement of materials between units happens automatically (see backdrop economy description, section xxx), players can also do it explicitly. Players can both take materials from other units, and give a unit's materials to others.

`transfer` *unit m n* ActionType

This is the action of transferring supply to the given unit *unit*. The desired amount is *n*; if *m* is a valid material type, then only that type will be transferred, otherwise the action will transfer all types of materials possible. The actual transfer amounts may be less than *n*. [If *unit* is NULL, then is equiv to discarding material?]

`acp-to-unload` *u1 m* -> *acp* Table

`acp-to-load` *u1 m* -> *acp* Table

These tables are the number of action points used up by one material transfer action from *u1* to *u2*. The amount is independent of the material type being transferred. If either value is `0`, then the material cannot be transferred. Defaults to `0`.

`unload-max` *u1 m* -> *n* Table

`load-max` *u2 m -> n* Table

These two tables determine how much of material *m* can be transferred out of a unit of type *u1* and into one of type *u2* in one transfer action. The actual quantity transferred by the action is the minimum of these two values. A value of `0` disallows manual transfer. Both default to `-1`, which allows any amount to be transferred.

4.22.11 Changing Sides

`change-side` *side* ActionType

This is the action of changing the actee's side to *side*. The *side* can be any allowable side, and the actee may be any unit controlled by the actor's side.

`acp-to-change-side` *acp* UnitTypeProperty

If the value of this property is greater than 0, then this type of unit can be ordered to change to another given side. The type must also be allowed to be on the new side. Defaults to `0`.

4.22.12 Disbanding

Disbanding is the voluntary and controlled destruction of a unit, performed by the unit itself or another unit. A disbanded unit always vanishes, rather than changing to its `wrecked-type`.

`disband` *unit* ActionType

This is the action of removing hp from *unit*. The unit will vanish if all its hit points are gone.

`acp-to-disband` *u1 u2 -> acp* Table

This table is the number of action points used by the unit *u1* to do a disband action on unit *u2*. Defaults to `0`.

`hp-per-disband` *u1 u2 -> hp* Table

This table is the number of hp lost in a disband action performed by *u2*. Defaults to `0`, which disallows disbanding.

A disbanded unit can be scavenged for materials.

`supply-per-disband` *u m* -> *n*% Table

This table is the percentage of the unit's supply that is recovered from a single disband action. If the value is zero, then the unit's supply will not be recovered by the disbanding process, and be lost permanently. If any supply remains when the unit's hp is 0, then that supply will be lost also. Defaults to `100`, which means that the entire supply will be recovered on the first disband action.

Note that if an essential supply is 100% recovered before the unit is completely disbanded, then it may die from starvation first. A partly-disbanded unit may still acquire supply from nearby units, via the backdrop economy.

`recycleable-material` *u m* -> *n* Table

This table is the quantity of each type of material that becomes available when a unit is completely disbanded. The materials go to transports, occupants, and nearby units, in that order. Any materials exceeding capacities of these units will be discarded. These materials become available only when the unit vanishes. Defaults to `0`.

4.22.13 Transferring Parts

Units of variable size can transfer parts of themselves to other units, or create a new unit.

`transfer-part` *n unit* ActionType

This action moves *n* parts of the actee to *unit*, or creates a new unit if *unit* is omitted. If *n* is negative, this takes from *unit* instead. If the action takes all the parts of any involved unit, then it vanishes.

`acp-to-transfer-part` *acp* UnitTypeProperty

Defaults to `0`.

4.22.14 Changing Type

`change-type` *u* ActionType

`acp-to-change-type` *u1 u2* -> *acp* Table

Defaults to `0`.

`material-to-change-type` *u m* -> *n* Table

Defaults to `0`.

4.22.15 Combat

Xconq combat is somewhat abstract; the attacking player decides what sort of attack to mount and perhaps when to retreat, but all else happens automatically.

Combat may last longer than a single action; it is then called a *battle* and divided into *rounds*. The battle exists until one participant has a commitment of zero. Units in a battle need not attack, and no damage will occur if none do so, but they cannot move away until no longer committed.

The attacker/defender distinction applies only to a single action.

`attack` *unit* [*commitment*] ActionType

This action is a direct attack on the given *unit*. The *unit* must be known to the attacking unit's side.

`overrun` *x y z* [*commitment*] ActionType

Overruns are a sort of combined attack/capture/move action. The basic theory of an overrun is that the actor will attack, capture, or co-occupy the given destination. The exact effects depend on the types and sides of units in the destination.

`acp-to-attack` *u1 u2* -> *acp* Table

This table is the number of action points used up by the attacker. Defaults to `1`.

`acp-to-defend` *u1 u2* -> *acp* Table

This table is the number of action points used up by the defender. Defaults to `1`.

`attack-range-min` *u1 u2* -> *dist* Table

This table is the minimum distance at which a unit can attack another. Defaults to `0`.

`attack-range` *u1 u2* -> *dist* Table

This table is the maximum distance at which a unit can attack another. Defaults to `1`.

One round of combat consists of an attack, a reaction, and a calculation of effects.

The defender's reaction is completely automatic, and occurs as part of the attack action. The defender's side does not get a chance to decide what to do until the next round, although doctrine can constrain the randomness somewhat.

`surrender-chance-per-attack` *u1 u2* -> *n%* Table
: This table is the chance that u2 will surrender to u1 immediately upon being attacked. Defaults to `0`.

`withdraw-chance-per-attack` *u1 u2* -> *n%* Table
: This table is the chance that u2 will retreat from u1 immediately upon being attacked. Defaults to `0`.

`acp-for-retreat` *u1 u2* -> *acp* Table

In an overrun action, if all the defending units are destroyed, the attacker has sufficient acp and mp, and the destination is safe to enter, then the attacker can move into the defenders' cell.

Firing is a kind of attack that can take place at a distance, involves no commitment or counter-attack, and for which the type of ammo may be selected.

`fire-at` *unit* [*m*] ActionType
: This is the action of firing at a given *unit*. If *m* is given, then that type will be used as ammo, otherwise all available types will be used together.

`fire-into` *x y* [*z*] [*m*] ActionType
: This is the action of firing into the cell at *x,y*. If *z* is given, then the fire will be concentrated on units at that elevation. If *m* is given, then that type will be used as ammo, otherwise all available types will be used together.

`acp-to-fire` *acp* UnitTypeProperty
: If this property is greater than 0, this type may attack by firing. Defaults to `0`.

`acp-to-be-fired-on` *u1 u2* -> *acp* Table
: This table is the acp lost when a unit is being fired upon. Defaults to `1`.

`range` *dist* UnitTypeProperty

This property is the maximum distance to which a unit can fire. Defaults to `1`.

`range-min` *dist* UnitTypeProperty

This property is the minimum distance to which a unit can fire. Defaults to `0`.

`elevation-at-max-range` *dist* UnitTypeProperty

[elaborate calc to interpolate while rising and falling, basically approximating a parabola]

Both attack and fire combat calculate hits and damage in the same way.

`hit-chance` *u1 u2* -> *n%* Table

This table is the basic chance that a unit of type *u1* will actually hit a unit of type *u2*. Defaults to `0`.

`attack-terrain-effect` *u1 t* -> *n%* Table

`defend-terrain-effect` *u2 t* -> *n%* Table

These tables specify the effect of attacker's and defender's respective terrains on `hit-chance`. These chances are multiplied with the basic hit chance. Default to `100`.

`hit-cxp-effect` *u1 u2* -> *n* Table

This table is the effect of combat experience on hit chance. Its value is interpolated according to actual experience (so that *n* is the effect when *u1* is at its maximum experience), then multiplied with the hit chance. Defaults to `100`.

`hit-falloff-range` *u1 u2* -> *n* Table

This table is the maximum range at which the effectiveness of combat is *not* affected by distance. Defaults to `1`.

`hit-at-max-range-effect` *u1 u2* -> *n%* Table

This is the multiplier for the effectiveness of combat at the maximum range possible. Defaults to `100`.

`damage` *u1 u2* -> *hp* Table

This table is the basic amount of damage caused by a successful attack. The value is a "dice spec" [explain somewhere] Defaults to `1`.

The damage in an attack is always prorated by commitment; the table value is for attacks at full commitment.

`damage-cxp-effect` *u1 u2* -> *n* Table

This table is the effect of combat experience on damage. Its value is interpolated according to actual experience (so that *n* is the effect when *u1* is at its maximum experience), then multiplied with both the dice size and the addend of the damage spec. Defaults to `100`.

`hp-min` *u1 u2* -> *hp* Table

This table is the lowest hp possible for *u1* from attacks by *u2*. Further attacks by *u2* are still valid, but have no effect. Defaults to `0`.

You can set a unit to use a material as ammo.

`consumption-per-attack` *u1 m* -> *n* Table

`hit-by` *u2 m* -> *n* Table

These tables specify material consumption in combat. For each material `m`, the min of these two values is the amount of u1's supply used up in an attack on u2. Both default to `0`.

`material-to-fight` *u m* -> *n* Table

This table is a minimum of each material that is necessary to either attack or defend. Defaults to `0`.

Transports can protect their occupants, and vice versa.

`protection` *u1 u2* -> *n%* Table

Transport's destruction may leave occupants stranded on hex, will do some sort of auto-escape or die if terrain is hostile. [use ferry-on-leave to decide]

`stack-protection` *u1 u2* -> *n%* Table

Several other side-effects of combat may also be defined.

`retreat-chance` *u1 u2* -> *n%* Table

This table is the chance that *u2* will retreat if hit by *u1*. Defaults to `0`.

`cxp-per-combat` *u1 u2* -> *cxp* Table

This table is the number of combat experience points gained by *u1* by surviving a combat round with *u2*. This applies equally to attackers and defenders. Defaults to `0`.

4.22.16 Capture

Finally, a unit can attempt to capture another unit directly. This means that the unit's side changes to that of the capturing unit.

`capture` *unit* ActionType

This is the action of capturing the given *unit*.

`acp-to-capture` *u1 u2* -> *acp* Table

This table is the number of acp used up by a `capture` action. Defaults to `0`, which disallows capture.

`capture-chance` *u1 u2* -> *n%* Table

This table is the basic chance for *u1* to capture *u2*. Defaults to `0`.

`independent-capture-chance` *u1 u2* -> *n%* Table

This table is the basic chance for *u1* to capture an independent unit of type *u2*. If the value is `-1`, then the chance of capture is given by the `capture-chance`. Defaults to `-1`.

`scuttle-chance` *u t* -> *n%* Table

This table is the chance that a unit whose capture is guaranteed will destroy itself instead. Scuttling is destructive, so unit changes to `wrecked-type`. Occupants of an about-to-be-captured unit will also attempt to scuttle. Defaults to `0`.

`occupant-escape-chance` *u1 u2* -> *n%* Table

This table is the chance that an occupant *u1* will escape during the capture of a unit of type *u2*. Occupants that do not escape are either captured themselves or destroyed, depending on their type and the capturing unit's side. Defaults to `0`.

`hp-to-garrison` *u1 u2* -> *n* Table

This table is the number of hp that will be taken from the capturing unit *u1* in order to guard a captured *u2*. If the amount is the unit's full hp, then the unit will vanish and any occupants will be distributed to the captured unit, to open terrain, or will vanish themselves if there is no other option. Defaults to `0`.

`cxp-per-capture` *u1 u2* -> *ep* Table

This table is the number of combat experience points gained by *u1* by capturing *u2*. Defaults to `0`.

`cxp-on-capture-effect` *n* UnitTypeProperty

This property gives the change in a unit's cxp due to being captured, expressed as a multiplier. Defaults to `100`.

4.22.17 Detonation

Detonation is an action and/or behavior that causes damage indiscriminately. The action specifies the location of the detonation, which may be in the unit's cell or an adjacent one. A unit that detonates loses hp, changing to its `wrecked-type` if it loses all of its hp. It also hits every unit within a specified radius. Detonation may also affect terrain within a specified radius.

`detonate` *x y z* ActionType

This action detonates the actee at the given location *x,y,z*.

`acp-to-detonate` *acp* UnitTypeProperty

This property is the number of action points used by one detonate action. Defaults to `0`, which disallows detonation.

`hp-per-detonation` *hp* UnitTypeProperty

This property is the number of hp lost in each detonation. Defaults to `0`.

`detonation-unit-range` *u1 u2* -> *dist* Table

This table gives the range of effect from detonation of *u1*. The severity falls off according to the inverse square law extrapolated from the adjacent cell damage. (1/4 severity at range 2, 1/9 at 3, etc.) Defaults to `0`.

`detonation-damage-at` *u1 u2* -> *hp* Table

This table is the severity of *u1*'s hit on a unit *u2* in the same cell. Defaults to `0`.

`detonation-damage-adjacent` *u1 u2* -> *hp* Table

This table is the severity of *u1*'s hit on a unit *u2* in an adjacent cell. Defaults to `0`.

`detonation-terrain-range` *u t* -> *dist* Table

Defaults to `0`.

`detonation-terrain-damage-chance` *u t* -> *n%* Table

Defaults to `0`.

`terrain-damaged-type` *t1 t2* -> *n* Table

Relative chance that terrain of type *t1* damaged by a detonation will change into another type *t2*. Defaults to `0`.

The following tables and properties can be used for units that cannot detonate deliberately by doing a detonate action.

`detonate-on-hit` *u1 u2* -> *n%* Table

This table is the chance that a hit on *u1* by a unit of type *u2* will cause it to detonate (once). Noncombat reductions in hp, such as attrition, have no effect. Defaults to `0`.

`detonate-on-death` *n%* UnitTypeProperty

This property is the chance that if this type is about to die from a combat hit, it will detonate first. Defaults to `0`.

`detonate-on-capture` *u1 u2* -> *n%* Table

This table is the chance that a unit of type *u1* will detonate if a capture by a unit of type *u2* is about to succeed. Defaults to `0`.

`detonate-on-approach-range` *u1 u2* -> *dist* Table

When a unit of type *u2* on a non-trusted [?] side appears at a distance of *dist* or less, then *u1* will detonate. If `-1`, then unit will not detonate upon approach. Defaults to `-1`.

`detonation-accident-chance` *u t* -> *n.f%* Table

This table is the chance that the unit will detonate spontaneously. This is checked once/turn, at the beginning of the turn, and also upon each entry to a cell, if moving. Defaults to `0`.

4.22.18 Altering Terrain

`alter-terrain` *x y t* ActionType

This action changes the type of the cell at *x,y* to *t*.

`add-terrain` *x y dir t* ActionType

This action adds a connection or border of type *t* to the cell at *x,y*, in direction *dir*.

`remove-terrain` *x y dir t* ActionType

This action removes a connection or border of type *t* to the cell at *x,y*, in direction *dir*.

`acp-to-add-terrain` *u t* -> *n* Table

`acp-to-remove-terrain` *u t* -> *n* Table

For auxiliary terrain types, these tables are the costs to add or remove. For cell terrain, the costs of removing the old type and adding the new type are added together.

`alter-terrain-range` *u t* -> *n* Table

This table is the maximum distance at which a unit can alter terrain *t*. Defaults to `0`, which means that the unit can change only the terrain in its own cell.

At present, all sides that have seen the terrain once will be informed about any changes.

4.23 Environmental Computation

This section describes how to set up backdrop computations.

4.23.1 Random Parameters

Environmental conditions may be computed randomly.

`temperature-average` *n* TerrainTypeProperty

This property is the average temperature for each type of terrain. Defaults to `0`.

`temperature-variability` *n* TerrainTypeProperty

This property is the amount of totally random variation in the temperature in each cell. Defaults to `0`.

`wind-force-average` TerrainTypeProperty

`wind-force-variability` TerrainTypeProperty

`wind-variability` TerrainTypeProperty

`wind-mix-range` GlobalVariable

This variable is the radius out to which winds interact. If 0, then winds in adjacent cells can vary independently of each other, and do not interact in any way. Defaults to `0`.

4.23.2 Season Parameters

`year-length` *n* WorldProperty

This property is the number of turns in an annual cycle. If less than `2`, then no seasonal effects will be calculated. Defaults to `0`.

`day-length` *n* WorldProperty

This property is the number of turns in a single day. If less than `2`, then day and night will not be calculated. Defaults to `0`.

Note that `year-length` and `day-length` are completely independent of each other, and it is possible to have days that are longer than years.

`initial-year-part` *n* AreaProperty

This property is the season of the first turn in the game. Defaults to `0`.

`initial-day-part` *n* AreaProperty

This property is the hour of the first turn in the game. Defaults to `0`.

[need amount of daylight, twilight, etc]

4.23.3 Varying Activity with the Season

`acp-season-effect` *xxx* UnitTypeProperty

This property is the effect of the seasons on acp. This property is added to the basic `acp-per-turn`. Defaults to `()`.

4.23.4 Varying Temperature with the Season

`temperature-year-cycle` GlobalVariable

`temperature-moderation-range` *distance* TerrainTypeProperty

This property is the radius of the area whose raw temperatures will be averaged to get the actual temperature. This can be very time-consuming to calculate, so only values of 0 (no averaging) and 1 (average with adjacent cells) are recommended. Defaults to `0`.

4.23.5 Weather Parameters

While the seasons change relatively slowly and predictably, weather can change drastically from turn to turn. *Xconq* weather is based on a daily cycle of heating and cooling plus the movement of water vapor.

Weather and seasons can be defined completely independently of each other. The weather model assumes a constant basic temperature, set from summer-equator if the season model is not being used.

Atmospheric vapor is modelled by having a vapor quantity in each cell [define a layer for this]. Vapor originates with evaporation from terrain, moves around with changing winds and air pressure, and high levels result in clouds, rain, and snow.

4.24 Environmental Effects

The environmental conditions include temperature, coatings such as snow, and atmospheric conditions.

[specify these]

The current environmental conditions in each cell [or in world as a whole? or calc by regions?] derive from a combination of three calculations: random, seasons, and weather.

4.24.1 Coating Effects

[effects of coating should be increased attrition, decreased productivity, decreased activity and mobility]

4.24.2 Effects of Temperature on Units

Transports can protect their occupants from temperature extremes.

`temperature-protection` *u1 u2* -> *t/f* Table

4.25 Economy

The following parameters control the automatic production, distribution, and consumption of materials by units and by cells.

4.25.1 Unit Production and Consumption

Units can be set to always produce some amount of material without taking explicit action.

`base-production` *u m* -> *n* Table

This table is the basic amount of each material *m* produced by a unit of type *u* in each turn. Defaults to `0`.

`occupant-base-production` *u m* -> *n* Table

This table is the base production of each material *m* when a unit of type *u* is an occupant. Defaults to `0`.

`productivity` *u t* -> *n%* Table

This base is the percentage productivity of a unit of type *u* when on terrain of type *t*. This is multiplied with the basic production rate to get actual material production, so productivity of `0` completely disables production on that terrain type, and productivity of `100` yields the rate specified by `base-production`. Defaults to `100`.

`productivity-min` *u m* -> *n* Table

`productivity-max` *u m* -> *n* Table

These tables are the lower and upper bounds on actual production after multiplying by productivity. Default to `0` and `9999`, respectively.

`base-consumption` *u m* -> *n* Table

This table sets the amount of materials consumed by the unit in a turn, even if it doesn't move or do anything else. Defaults to `0`.

`hp-per-starve` *u m* -> *hp* Table

If the unit runs out of a material that it must consume, this table specifies how many hp it will lose each turn that it is starving. If starving for several reasons, loss is max of starvation losses, not the sum. Defaults to `0`.

`consumption-as-occupant` *u m* -> *n%* Table

This table is the consumption by a unit of type *u1* when it is an occupant of *u2*, expressed as a percentage of its `base-consumption`. This is useful for units such as planes which always consume fuel in the air but not on the ground. Defaults to `100`.

4.25.2 Terrain Production and Consumption

Materials may be produced by cells, redistributed, and also taken up by units. Some amount of material may need to stay in the cell's storage, or the type of terrain might change. Exhaustion is tested after all consumption has been accounted for.

`terrain-production` *t m* -> *n* Table
: This table is the amount of each material *m* produced by a cell of the given type *t* in each turn. Defaults to `0`.

`terrain-consumption` *t m* -> *n* Table
: This table is the amount of material *m* consumed by a cell of type *t* each turn. If insufficient material is available, then the terrain may change type. Defaults to `0`.

`change-on-exhaustion-chance` *t m* -> *n%* Table
: This table is the chance that a cell of type *t*, with no supply of material of type *m*, will become exhausted and change to its exhausted type.

`terrain-exhaustion-type` *t1 m* -> *t2* Table
: If *t2* is not non-terrain, then this table says that any cell with terrain *t1* that is exhausted will change to *t2*. If several materials are exhausted in the same turn, then the lowest-numbered material type will determine the new terrain type. Defaults to `non-terrain`.

`people-consumption` *m1 m2* -> *n* Table
: This table is the base consumption per turn by people of type *m1* of each other material type *m2*. Defaults to `0`.

`people-production` *m1 m2* -> *n* Table
: This table is the people of type *m1* base production per turn of each other material type *m2*. Defaults to `0`.

4.25.3 Supply Lines

In real life, material production and consumption rarely occur in the same place at the same time. For some games, the player must transfer materials manually, by loading and unloading from units. However, this can be time-consuming and difficult, and is best reserved for scarce and/or valuable materials. For more common materials, *Xconq* provides *supply lines*.

`in-length` *u1 m -> dist* Table

`out-length` *u2 m -> dist* Table

These two tables together determine the length of supply lines between units. The given type of material can only be transferred from unit type *u1* to unit type *u2* if the distance is less than the minimum of the `in-length` of *u1* and the `out-length` of *u2*. For instance, the `in-length` for a fighter's fuel might be 3 cells, while the `out-length` of fuel from a city is 4 cells. Then the fighter will be constantly supplied with fuel when within 3 cells of a city. If the fighter's out-length is -1, it will never transfer any fuel to the city. An in- or out-length of `0` means that the two units must be in the same cell, while a negative length disables the automatic transfer completely. Long `out-length` lines should be used sparingly, since the algorithm uses the `out-length` to define a radius of search for units to be resupplied. Both default to `0`.

4.25.4 Trade

To move materials automatically between cells, you must define the demand and supply relationships, as well as the rate and distance that materials can move.

Demand for a material originates with consumption and construction needs, issuing either from a side or from some other part of the economy.

4.25.5 Taxation

A side can set a taxation rate, which is the amount of material that will be taken from the cell-based economy and given to units on that side.

Taxes may be negative, which will have the effect of returning materials from units back to cells.

Taxation is the last step in economic calculations.

4.25.6 Material Conversion

Some types of materials can be converted or combined into other types of materials.

[do by letting production vary according to consumption?]

[in general, should distinguish productive from consumptive units, specify as limits on in/out for each rtype]

4.26 Random Events

`random-events` *method-list* GlobalVariable

This variable is a list of random event methods that will be run at the end of each turn. The list is not ordered.

4.26.1 Terrain Attrition

Attrition is the automatic loss of hit points due to being in certain types of terrain.

`attrition-in-terrain` Method

For every unit not in a transport, this method computes the chance to lose hit points, then damages the unit accordingly. This method runs once per turn.

`attrition` *u t* -> *.01hp* Table

This table is the rate of loss of hp per turn. The terrain used is cell or connection terrain as appropriate for the unit's position. Defaults to `0`.

4.26.2 Terrain Accident

Accidents result in the damage or disappearance of a unit in the open in some kinds of terrain.

`accidents-in-terrain` Method

For every unit not in a transport, this method computes the chance to be hit or to vanish completely. This method runs once per turn.

`accident-hit-chance` *u t* -> *.01n%* Table

This table is the chance of the unit being hit while in the given terrain. Defaults to `0`.

`accident-damage` *u t* -> *hp* Table

This table is the hp that will be lost in an accident. Defaults to `0`.

`accident-vanish-chance` *u t* -> *.01n%* Table

This table is the chance of the unit simply vanishing while in the given terrain. Defaults to `0`.

4.26.3 Revolt

Revolt is a spontaneous change of side, occurring in place of a side-given unit action. The new side may be none (independence) or another side. [only if other side wants it?] [50/50 chance?]

`units-revolt` Method

For each completed unit, this method decides whether the unit revolts, then changes its side.

`revolt-chance` *.01n%* UnitTypeProperty

This property is the chance for the unit to revolt spontaneously. Defaults to `0`.

4.26.4 Surrender

`units-surrender` Method

For each completed unit, this method checks whether the unit will surrender to a nearby unfriendly unit.

`surrender-chance` *u1 u2* -> *.01n%* Table

This table is the chance that a unit of type *u1* will change its side to match the side of a unit *u2* that is within the `surrender-range` for the two types. Defaults to `0`.

`surrender-range` *u1 u2* -> *dist* Table

This table is the distance out to which a unit of type *u1* will surrender to a unit of type *u2*. Defaults to `1`.

4.27 The Random State

It is useful to be able to restart the random number generator consistently.

`random-state` *n* GlobalVariable

This variable is the state of the random number generator. If this is not used, then the initial state of the random number generator will be set in a system-dependent way.

4.28 Images and Image Families

The `imf` form defines graphical images in a platform-independent way. An *image family* is a named collection of images of varying sizes and depths.

`imf` *name* [*properties*] [*images*] Form

This form declares an image family to exist, with the name *name* and *properties*, and consisting of the specified *images*. Each image has the form `((`*w h* `[tile])` [*properties*] `(`*type data*`...) ...)`, where *w* and *h* are its width and height, respectively, the *type* may be one of `color`, `mono`, or `mask`, and the *data* consists of strings of hexadecimal digits. The data strings may include slashes, which have no effect on interpretation, but are useful to indicate each row of an image. Color images may also have additional properties, which come between the *type* and the *data*.

Multiple forms with the same name may occur, and each adds to the family, overwriting individual image parts that are of the same size and depth.

`tile` Symbol

If this symbol appears following the dimensions of an image, it indicates that the image is a pattern tile rather than a single image.

`actual` *w h* ImageProperty

This property is the actual size of the image data. [Ever really used?]

`embed` *name* ImageProperty

This property specifies that another image, similar to the image family named by *name*, is already embedded within the image, and so *Xconq* need not superimpose such an image itself. This may occur when an image has a "builtin" side emblem, or is readily identifiable as belonging to a particular side, and it would be redundant for *Xconq* to add an emblem when displaying a unit.

`embed-at` *x y w h* ImageProperty

`mono` *data...* ImageProperty

This property indicates that the data represents a monochrome image.

`mask` *data...* ImageProperty

This property indicates that the data represents a mask.

`color` [*properties*] *data...* ImageProperty

This property indicates that the data represents a color image.

`pixel-size` *n* ColorImageProperty

This property is the number of bits used to encode each pixel.

`row-bytes` *n* ColorImageProperty

This property is the number of bytes in each row of the image.

`palette` [*name* `|` (*index r g b*) ...] ColorImageProperty

This property is the color palette that should be used with the image.

`palette` *name* (*index r g b*) ... Form

This form defines a palette with the given *name*.

`color` *name r g b* Form

This form names the color.

Note that for the purposes of stability and change tracking, tools that generate image families use a more restricted format. This format requires a separate imf form for each size of image, the size is on the same line as the imf name, and each image/mask is on a separate line, indented by 2. (See the existing `lib/*.imf` files for further detail.)

4.29 Default Display Style

The exact style of display depends on the user interface and on user preferences, but for some games, you may want to encourage a particular style by making it be the default.

`unseen-char` *str* GlobalVariable

This variable is a string whose first character will be used to represent unexplored terrain. If the string consists of two characters, the second char will be scattered throughout a field of the first char. Defaults to `""`.

`unseen-color` *str* GlobalVariable

This variable is the name of a color that will be used to represent unexplored terrain. Defaults to `""`.

`unseen-image-name` *str* GlobalVariable

This variable is the name of an image that will be used to represent unexplored terrain. Defaults to `""`.

`grid-color` *str* GlobalVariable

This variable is the name of a color to use to draw the cell-separating grid. Defaults to `""`.

4.30 Dates and Time

You can make *Xconq* display game time as a calendar date, rather than as a simple turn number.

`calendar` *type data...* GlobalVariable

This variable is the description of the calendar type that will be used. If `none`, then turns will be reported numerically starting from `1`. If `usual`, then the standard Gregorian calendar will be used. (Other calendars may be supported in the future.) Defaults to `()`, which is equivalent to `(number "turn")`.

For the `usual` calendar, the *data* defines how long a turn is, in terms of the calendar. For instance, a time measure of `"day"` (and a base date of `"1 Jan 1900"`) will result in turns `"1 Jan 1900"`, `"2 Jan 1900"`, etc, while a date unit of `"year"` will yield just `"1900"`, `"1901"`, and so forth.

If the numeric or `number` calendar is in use, then a *data* of `"day"` will yield `"day 1"`, `"day 2"`, etc.

The rest of this variable lists the name of each season and the turns within a year for which it is appropriate. A twelve-turn year with four seasons could be

```
((0 2 "winter") (3 5 "spring") (6 8 "summer") (9 11 "autumn"))
```

If any number ranges overlap, then the first match will be used, while if a particular turn has no named season, then it will go unnamed in the display.

`none` Symbol

`usual` Symbol

`initial-date` *str* GlobalVariable

This variable is the date, in the specified calendar system, of the first turn. Defaults to `""`, which has the effect of setting the initial date to be whatever the calendar does with turn number 1.

`turn` *n* GlobalVariable

This variable is the number of the current turn. Defaults to `0`.

`last-turn` *n* GlobalVariable

This variable is the number of the last turn. Defaults to `-1`, which means that there is no limit on the number of turns.

`extra-turn-chance` *n%* GlobalVariable

This variable is the chance that the game will go one more turn after the `last-turn`.

Xconq is currently limited to games of 32,767 turns.

4.30.1 Real Time

A game may also be limited in real time.

`real-time-for-game` *seconds* GlobalVariable `real-time-per-turn` *seconds* GlobalVariable

`real-time-per-side` *seconds* GlobalVariable

`elapsed-real-time` *seconds* GlobalVariable

This is the difference in real time between the start of the game and its current state.

4.31 Miscellany

GDL forms in this section are those that do not seem to fit anywhere else.

`name-internal` *str* UnitTypeProperty

Internally used type name.

4.31.1 Debugging

`print` *value* Form

This form prints to a console (or whatever the interface provides) the object *value*, in GDL syntax.

4.31.2 Internal AI Data

These are normally computed and used internally by AIs. They can be filled in by a game design, but the effects are undocumented and will depend on the working of the AI using these forms.

`zz-fr` XXX

`zz-b` XXX

`zz-bb` XXX

`zz-transport` XXX

`zz-c` XXX

`zz-cm` XXX

`zz-cc` XXX

`zz-bw` XXX

`zz-basic-hit-worth` Table

`zz-basic-capture-worth` Table

`zz-basic-transport-worth` Table

5 Hacking Xconq

Although *Xconq* and its GDL have considerable power and flexibility already built in, you may decide that you want to modify the *Xconq* program itself. You should know what you are doing; *Xconq* is designed to be modifiable, but it is not simple code. In the past, people have found it easy to make changes, but much harder to make them correctly!

Xconq is designed to be portable to different types of user interfaces. It is based on a kernel-interface architecture, where the semantics of the game, as documented in the preceding chapters, is part of the kernel, while the main program and player interaction are specific to each system.

Xconq is also designed to allow the addition of new AIs. The default `"mplayer"` AI, while it is flexible and will attempt to play any side in any game, does not have the depth that is often important to success in a game. Its position is that of a generic AI program that can learn to play any game, given only the rules; while such a program might figure out how to win at tic-tac-toe or checkers, it is not going to be particularly good at the subtleties of go or chess.

The *Xconq* GDL is also extensible. This is useful when the basic GDL does not provide some feature that is essential to a game.

5.1 Kernel

The kernel is the part of *Xconq* shared by all interfaces. It does no I/O except to files or for debugging.

Specifically, the kernel supplies the following functionality:

Data structure initialization. (`init_data_structures`)
Game module loading and interpretation. (`load_game_module`)
Initial player/side setup. (`make_trial_assignments`)
Synthesis methods. (`run_synthesis_methods`)
Final player/side setup. (`make_assignments`)
Game execution. (`run_game`)
Implementations of unit actions. (`prep_*_action`)
AI players.

Help Info (`get_help_text`)
Game saving and scorekeeping.

5.1.1 Configuration Options

There are a small number of options available to alter aspects of the kernel. These are defined in `kernel/config.h`.

[eventually describe all of them?]

5.1.2 Porting the Kernel

The kernel should be restricted to ANSI C, and should avoid or optionalize features not in "traditional" C, such like prototypes. Although the kernel uses stdio, it does not assume the presence of a console (stdin, stdout, stderr). For instance, a graphical interface can arrange to disable stdin entirely and direct stdout/stderr into a file (see the Mac interface sources for an example).

You should be careful about memory consumption. In general, the kernel takes the attitude that if it was worth allocating, it's worth hanging onto; and so the program does not free much storage. Also, nearly all of the allocation happens during startup. Since a game may run for a very long time (thousands of turns perhaps), it is important not to run the risk of exhausting memory at a climactic moment in the game!

Also, the kernel should not exit on its own. The only permissible times are when the internal state is so damaged that interface error-handling routines (see below) cannot be called safely. Such situations are rare. If you add something to the kernel and need to handle error situations, then you should call one of the interface's error-handling routines. There are distinct routines for problems during initializations vs problems while running, and both error and warning routines. Warning routines may return, so kernel code should be prepared to continue on, while error routines will never return.

5.1.3 Writing New Synthesis Methods

You can add new synthesis methods to *Xconq*. This may be necessary if an external program does not exist, is unsuitable, or the external program interface is not available. Synthesis methods should start out by testing whether or not to run, and should never assume that any other method has been run before or after, nor that any particular game module has been loaded. However, "tricks" are usually OK, such as setting a particular global variable in a particular module only, then having the synthesis method test whether that global is set. See the file `init.c` for further details.

Synthesis methods that take longer than a second or two to execute should generate percent-done info for the interface to use, via the function `announce_progress`. Be aware that most methods will be O(n) or O(n*n) on the size of the world or the number of units, so they can take much longer to set up a large game than a small one. Players will often go overboard and start up giant games, so this happens frequently. Also, *Xconq* may be running on a much smaller and slower machine than what you're using now.

5.1.4 Writing New Namers

[describe hook and interface]

5.1.5 Writing New AIs

You can add new types of AIs to *Xconq*. You would do this to add different strategies as well as to add AIs that are programmed specifically for a single game or class of games. (This is useful because the generic AI does not always understand the appropriate strategy for each game.)

You have to design the object that is the AI's "mental state". If your AI need only react to the immediate situation, then this object can be very simple, but in general you will need to design a fairly elaborate structure with a number of substructures. Since there may be several AIs in a single game, you should be careful about using globals, and since *Xconq* games may often run for a long time, you should be careful not to consume memory recklessly.

Name. This is a string, such as `"mplayer"`. It may be displayed to players, so it should not be too cryptic.

Validity function. This runs after modules are loaded, and during player/side setup, and decides whether it can be in the given game on the given side. [have a chain of fallback AIs, or blow off the game?]

Game init function. This runs before displays are set up, just in case a display examines the AI's state.

Turn init function. This runs after all the units get their acp and mp for the turn, but before anybody actually gets to move.

Unit order function. This gets run to decide what the unit should do. Usually it should be allowed to follow its plan. [do separate fns for before and after plan execution?]

Event reaction functions. [how many?]

Note that these functions have very few constraints, so you can write them to work together in various ways. For instance, an AI can decide whether to resign once/turn, once/action, or once for each 4 units it moves, every other turn.

[describe default AI as illustrative example]

5.1.6 Extending GDL

GDL has been designed so as to be relatively easily extensible. I say "relatively" because although it is quite easy to define a new keyword or table, it is not always so easy to integrate the implementation code into the kernel correctly.

Instead of actually changing GDL, you can experiment with an addition by using the `extensions` property of unit, material, and terrain types. In the code, you call `get_u_extension`, pass it the type, name of the property, and a default to return if the value was not given. In the game definition, the designer would say `(unit town (extensions (my-ext xxx)))`.

[show examples for global, property, table, event, task]

The file `gvar.def` defines all the global variables.

The file `utype.def` defines all the unit type properties.

From time to time, it may be worthwhile to extend unit objects. This should be rare, because games may have thousands of units, and each unit requires at least 100 bytes of storage already, so you should avoid making them any larger. Properties of an individual unit are scattered through `keyword.def`. Once the structure slot is added, you just need to add reading and writing of the value, using the `K_`*xxx* enum that was defined with the keyword. You should attempt to make a reasonable default and use it to avoid writing out the value, so as to save time when *Xconq* reads a game in.

GDL symbols beginning with `zz-` should be reserved for the use of AI code. You may want to add some of these, either to serve as a convenient place for AIs to cache the results of their analyis of a game, or else as a way for game designers to add "hints" for AIs that know to look at them.

5.2 Interface

The player interface is how actual players interact with the game. It need not be graphical or even particularly interactive, in fact it could even be a network server-style interface! However, this section will concentrate on the construction of interactive graphical interfaces.

An interface is always compiled in, so it has complete access to the game state. However, if your version of *Xconq* has any networking support, the interface should not modify kernel structures directly, but should instead use kernel routines. The kernel routines will forward any state modifications to all other programs participating in a game, so that everybody's state remains consistent.

A working interface must provide some level of capability in each of these areas:

Main program. The interface includes the main application and any system-specific infrastructure, such as event handling.

Interpretation of startup options. This includes choice of games, variants, and players.

Display of game state. This includes both textual and graphical displays, both static and dynamic.

Commands/gestures for unit tasks and actions, and for general state modifications.

Display update in response to state changes.

Realtime progress. Some game designs require the interface to support realtime.

Error handling.

The file `skelconq.c` in the `kernel` directory is a good example of a minimum working interface.

Don't let interfaces ever set kernel object values directly, always go through calls that can be "siphoned" for networking.

5.2.1 Main Program

The interface provides `main()` for *Xconq*; this allows maximum flexibility in adapting to different environments. In a sense, the kernel is a large library that the interface calls to do game-related operations.

There is a standard set of calls that need to be made during initialization. The set changes from time to time, so the following extract from '`skelconq`' should not be taken as definitive:

```
init_library_path(NULL);
clear_game_modules();
init_data_structures();
parse_command_line(argc, argv, general_options);
load_all_modules();
check_game_validity();
parse_command_line(argc, argv, variant_options);
set_variants_from_options();
parse_command_line(argc, argv, player_options);
set_players_from_options();
parse_command_line(argc, argv, leftover_options);
make_trial_assignments();
calculate_globals();
run_synth_methods();
final_init();
assign_players_to_sides();
init_displays();
init_signal_handlers();
run_game(0);
```

Note that this sequence is only straight-through for a simple command line option program; if you have one or more game setup dialogs, then you choose which to call based on how the players have progressed through the dialogs. The decision points more-or-less correspond to the different `parse_command_line` calls in the example. You may also need to interleave some interface-specific calls; for instance, if you want to display side emblems in a player/side selection dialog, then you will need to arrange for the emblem images to be loaded and displayable, rather than doing it as part of opening displays.

Once a game is underway, the interface is basically self-contained, needing only to call `run_game` periodically to keep the game moving along. `run_game` takes one argument which can be -1, 0, or 1. If 1, then one unit gets to do one action, then the routine returns. If 0, the calculations are gone through, but no units can act. If -1, then all possible units will move before `run_game` returns. This last case is not recommended for interactive programs, since moving all units in a large game may take a very long time; several minutes sometimes, and `run_game` may not necessarily call back to the interface very often.

5.2.2 Startup Options

Although there are many different ways to get a game started, you have three main categories of functionality to support: 1) selection of the game to play, 2) setting of variants, and 3) selection of players. For command-line-using programs, the file `cmdline.c` need only be linked in to provide all of this functionality. For graphical interfaces, you will need to design appropriate dialogs. This can be a lot of work, exacerbated by the fact that these dialogs will be the first things that new *Xconq* players see, and will therefore shape their opinions about the quality of the interface and of the game.

[more detail about what has to be in dialogs?]

Interface code should check all player specs, not proceed with initialization until these are all valid.

Both standard and nonstandard variants should vanish from or be grayed out in dialog boxes if irrelevant to a selected game.

5.2.3 Progress Indication

Some synthesis methods are very slow, and become even slower when creating large games, so the kernel will announce a slow process, provide regular updates, and signal when the process is done. The interface should display this in some useful way. In general, progress should always be displayed, although one could postpone displaying anything until after the first progress update, calculate an estimated time to completion, and not display anything if that estimate is for less than a few seconds. However, this is probably unnecessary.

`void announce_read_progress()`
: The kernel calls this regularly while reading game definitions. Interfaces running on slow machines should use this to indicate that everything is still working; for instance, the Mac interface animates a special cursor that indicates reading is taking place.

`void announce_lengthy_process(char *msg)`
: The kernel calls this at the beginning of each synthesis. The argument is a readable string that the interface can show to players.

`void announce_progress(int pctdone)`

The kernel may call this at milestones within a synthesis. The number ranges from 0 to 100.

`void finish_lengthy_process()`

The kernel calls this at the end of a synthesis.

5.2.4 Feedback and Control

The interface should provide visible feedback for every successful unit action initiated directly by the player, but it need not do so for failures, unless they are serious. It is better to prevent nonsensical input, for instance by disabling menus and control panel items. Simple interfaces such as for character terminals will have to relax these rules somewhat.

Interfaces should enable/disable display of lighting conditions.

5.2.5 Commands

There is no single correct way to support direct player control over units. Although keyboard commands and mouse clicks are obvious choices, it would be very cool to allow a pen or mouse to sketch a movement plan, or to be able to give verbal orders...

There is a common set of ASCII keyboard commands that are recommended for all *Xconq* interfaces that use a keyboard. These are defined in `kernel/cmd.def`. If you use these, *Xconq* players will be able to switch platforms and still use familiar commands. `cmd.def` defines a single character, a command name, a help string, and a function name, always in the form `do_*`. However, `cmd.def` does not specify arguments, return types, or behavior of those functions, so each interface must still define its own command lookup and calling conventions.

Prefixed number args should almost always be repetitions.

If already fully fueled, refuel commands should come back immediately.

A quit cmd can always take a player out of the game, but player may have to agree to resign. Player can also declare willingness to quit or draw without actually doing so, then resolution requires that everybody agree. If quitting but others continuing on, also have option of being a spectator. Could have notion of "leaving game without declaring entire game a draw" for some players. Allow for a timeout and default vote in case some voters have disappeared mysteriously. Must never force a player to stay in. Add a notion of login/logout so a side can be inactive but untouchable, possibly freezes entire game if a side is inactive. 1. if one player or no scoring confirm, then shut player down if one player, then shut game down 2. if side is considered a sure win (how to tell? is effectively a win condition then) or all sides willing to draw confirm, take side out, declare a draw, shut player down 3. if all sides willing to quit take entire game down 4. ask about resigning - if yes, resign, close display, keep game running if no, ask if willing to quit and/or draw, send msg to other sides Kernel support limited to must_resign_to_quit(side), similar tests.

5.2.6 Error Handling

The interface must provide implementations of these error-handling functions:

`void low_init_warning(str)`

This is for undesirable but not necessarily wrong things that happen while setting up a game. For instance, if players start out too close or too far from each other, it will often affect the play of the game adversely, so the kernel issues a warning, therby giving the prospective players a chance to cancel the game and start over. The kernel's warning message should indicate any likely results of continuing on, so the players can decide whether or not to chance it.

`low_init_error(str)`

This function should indicate a serious and unrecoverable error during initialization. It should not return to its caller.

`low_run_warning(str)`

Warnings during the game are rare but not unknown. They are very often due to bugs in `Xconq`, so any occurrence should be investigated further. It is possible for some game designs to have latent flaws that may result in a warning. In any case, the interface should allow the players to continue on, to save their game and quit, by calling `save_the_game`, or else quit without saving anything.

`low_run_error(str)`

In the worst case, *Xconq* can get into a situation, such as memory exhaustion, where there is no way to continue. The kernel will then call `run_error`, which should inform players that *Xconq* must shut itself down. They do get the option of saving the game, and the routine should call `save_game_state??` to do this safely. This routine should also not return to its caller.

`printlisp(obj)`

This is needed to print GDL objects to "stdout" or its equivalent.

5.2.7 Textual Displays

Text can take a long time to read, and can be difficult to provide in multiple human languages. (What, you thought only English speakers played *Xconq*? Think again!) Therefore, text displays in the interfaces should be as minimal as possible, and derive from strings supplied in the game design, since they can be altered without rebuilding the entire program.

(*Xconq* is not, at the moment, completely localizable, but that is a design goal.)

5.2.8 Display Update

Usually the interface's display is controlled by the player, but when `run_game` is executing, it will frequently change the state of an object in a way that needs to be reflected in the display immediately. Examples include units leaving or entering a cell, sides losing or winning, and so forth. The interface must define a set of callbacks that will be invoked by the kernel.

`update_cell_display(side, x, y, rightnow)`

[introduce area (radius or rect) update routines?]

`update_side_display(side, side2, rightnow)`

`update_unit_display(side, unit, rightnow)`

`update_unit_acp_display(side, unit, rightnow)`

`update_turn_display(side, rightnow)`

`update_action_display(side, rightnow)`

`update_action_result_display(side, unit, rslt, rightnow)`

`update_fire_at_display(side, unit, unit2, m, rightnow)`

`update_fire_at_display(side, unit, x, y, z, m, rightnow)`

`update_event_display(side, hevt, rightnow)`

`update_all_progress_displays(str, s)`

`update_clock_display(side, rightnow)`

`update_message_display(side, sender, str, rightnow)`

`update_everything()`

Each of these routines has a flag indicating whether the change may be buffered or not. To ensure that buffered data is actually onscreen, the kernel may call `flush_display_buffers()`, which the interface must define.

```
flush_display_buffers()
```

These may or may not be called on reasonable sides, so the interface should always check first that `side` actually exists and has an active display. [If side has a "remote" display, then interface has to forward?? No, because remote copy of game is synchronized and does own update_xxx calls more-or-less simultaneously]

Note that this is as much as the kernel interests itself in displays. Map, list, etc drawing and redrawing are under the direct control of the interface code.

5.2.9 Types of Windows and Panels

Xconq is best with a window-style interface, either tiled or overlapping. Overlapping is more flexible, but also more complicated for players. In the following discussion, "window" will refer to a logically unified part of the display, which can be either a distinct window or merely a panel embedded in some larger window.

The centerpiece window should be a map display. This will be the most-used window, since it will typically display more useful information than any other window. This means that it must also exhibit very good performance.

When a game starts up, the map display should be centered on one of the player's units, preferably one close to the center of all the player's units.

Another recommended window is a list of all the sides and where they stand in both the current turn and in the game as a whole. Each side's entry should include its name, a progress bar or other doneness indicator, and room for all the scores and scorekeepers that apply to that side.

If possible, you should also implement some kind of "face" or group of faces/expressions for a side, so get a barbarian's face to repn a side instead of generic. Could have interface generate remarks/balloons if face clicked on, perhaps a reason for feelings, slogan, citation of agreement or broken agreement, etc. Need 5 faces for hostile, unfavorable, neutral, favorable, friendly/trusting.

Overall status of side rules:

all grayed: out of game

grayed and x-ed out: lost

???: won

Progress bar rules:

missing: no units or no ai/no display

grayed frame: no acting units

empty solid frame: all acted

part full, black: partly acted

part full, gray: finished turn

5.2.10 Imaging

Imaging is the process of drawing pictorial representations. Not every interface needs it. For instance the curses interface is limited to drawing two ASCII characters for each cell, and its imaging code just has to choose which two to draw. However, full-color bitmapped displays need more attention to the process of getting an image onscreen.

No graphical icon should be drawn smaller than about 8x8, unless it's a text character drawn in two contrasting colors.

Interfaces should cache optimal displays for each mag, not search for best image each time.

Could allow 1-n "display variants" for all images, and for each orientation of border and connection.

Imaging variations can be randomly selected by UI, but must be maintained so redraws are consistent.

Allow the 64 bord/conn combos as single images, also advantage that all will be drawn at once.

Draw partial cells around edges of a window, to indicate that the world continues on in that direction.

Interface needs to draw only the terrain (but including connections and borders) in edge cells.

Could draw grid by blitting large light pattern over world, do by inverting so is easy to turn on/off. Do grids by changing hex size only in unpatterned color?

Draw large hexagon or rect in unseen-color after clearing window to bg stipple (if unseen-color different). Polygon should be inside area covered by edge hexes, so unseen area more obvious. Make large unseen-pattern that includes question marks?

If picture not defined for a game, use some sort of nondescript image instead of leaving blank. (small "no picture available" for instance, like in yearbooks)

To display night, could invert everything (b/w) or do 25/50% black (color) (let game set, so some games could be all-black at night, nothing visible) (have day/night coverage for each utype?)

To display elevation, use deep blue -> light gray -> dark brown progression, maybe also contour lines? To draw contour lines, for each hex, look at each adj hex. If on other side of contour's elev, compute interpolated point (in pixels) and save or draw a line to (one or both of the two) adj hex borders if they also have the contour line pass through. Guaranteed that line is part of overall contour line. Cheaper approach doesn't interpolate, just draws to midpoint of hex border (probably OK for small mags). Could maybe save contour lines once calculated (at each mag, lots of mem).

5.2.11 Animation

In addition to basic imaging, you can also support requests for the playing of animations or *movies*.

The kernel just calls `schedule_movie` to create one, and then `play_movies` when it is time to run all the movies that have been scheduled. It is up to the interface to do something useful. Note that the kernel is not aware of the movies' timing, so it is better not to call `run_game` until all the movies have finished playing. (Yes, this would be a good future enhancement!)

```
schedule_movie(side, movie_type, args...)
```

`play_movies(sidemask)`

Run all of the animations, sounds, etc that were scheduled previously, for the sides enabled in the side mask. It is allowable for the interface not to act on any user input while these are playing.

Several types of movies are predefined, so your interface can recognize them specially. These include `movie_miss`, `movie_hit`, `movie_death`, which are scheduled for the appropriate outcomes of combat.

5.2.12 Game Designer's Tools

An interface is not required to provide any sort of online designing tools, or even to provide a way to enable the special design privileges. Nevertheless, minimal tools can be very helpful, and you will often find that they are helpful in debugging the rest of the interface, since you can use them to construct test cases at any time.

A basic set of design tools should include a way to enable and disable designing for at least one side, a command to create units of a given type, and some sort of tool to set the terrain type at a given location. A full set would include "painting" tools for all area layers, including geographical features, materials, weather, side views, and so forth - about a dozen in all.

A least one level of undo for designer actions is very desirable, although it may be hard to implement. A useful rule for layers is to save a layer's previous state at the beginning of each painting or other modification action, when the mouse button first goes down.

The designer will often want to save only the part of the game being worked on, for instance only the units or only the terrain. The "save game" action should give designers a choice about what to save. For units particularly, the designer should be able to save only some properties of units. The most basic properties are type, location, side, and name/number. The unit id should not be saved by default, but should have its own option (not clear why).

Note that because game modules are textual and can be moved easily from one system to another, it is entirely possible to use one *Xconq* (perhaps on a Mac) to design games to be played on a Unix box under X11, or vice versa. Transferring the imagery is more difficult, although there is some support for the process.

5.2.13 Porting and Multiple Interfaces

In theory, it is possible to compile multiple interfaces into a single *Xconq* program, but this would be hard at best. They would have to be multiplexed appropriately and not conflict anywhere in the address space. Sometimes this is intrinsically impossible; how could you compile the Mac and X interfaces into the same program, and would the result be a Mac application, a Unix program, or what?

5.2.14 Useful Displays

This is a collection of minor but useful displays that might be worth adding to an interface.

A "mouse over" is a line or two of text that describes what the mouse/pointer is currently pointing at, and which updates automatically as the player moves the pointer around. This is better for high-bandwidth interfaces, since there may be a lot of updating involved. The volume can be reduced slightly by only redisplaying when the mouse moves, or, better, when what is being looked at changes. This is probably best done by recalculating the line of text and then comparing it to what has been drawn already, although if the display is very fast, you may not save much in drawing time. One approx 40-char lines covers basic info, such as terrain type and unit type; more detail may require multiple long lines.

5.2.15 Useful Options

A "follow action" option scrolls the screen to where the last event happened, such as combat. [etc]

5.2.16 Debugging Aids

Xconq is complicated enough that you can't expect to throw together a complete working interface over the weekend. Therefore, you should build some debugging aids into the interface. You can ifdef with the flag `DEBUGGING` so as to ensure the code won't be in final versions.

Display unit id if closeups, toplines, etc, if debugging is on.

5.2.17 Guidelines and Suggestions

Although as the interface builder, you are free to make it work in any way you like, there are a number of basic things you should do. Some of these are general user interface principles, others are specific to *Xconq*, usually based on experiences with the existing interfaces. Applying some of these guidelines will require judicious balancing between consistency with the different version of *Xconq* and consistency with the system you're porting to.

[following items should be better organized, moved in with relevant sections]

Draw single selected unit in a stack larger.

Draw single selected occupant in UR corner next to transport, when at mags that show both transport and occs.

There should always be some sort of "what's happening now" display so player doesn't wonder about apparently dead machine.

Image tool should report which type of resource is generating a given image, so can find which to hack on (report for selected image only).

Interfaces should ensure stability of display choices if random possibilities, so need to cache local decisions about appearance of units if multiple images to choose from, choice of text messages, etc.

Rules of Interaction: 1. Player can get to any unit in any mode. 2. Any player can prevent a turn from completing(/progressing?), unless a hard real limit is encountered. 3. All players see each others' general move/activity state, modes, etc. 4. Players can "nudge" each other. 5. Real time limits can be set for sides, turns, and games, both by players and by scenarios.

Player should be able to click on a desired unit or image, and effectively say "take this", either grabs directly or else composes a task to approach and capture.

Unit closeups should be laid out individually for each type, too much variability to make a single format reasonable.

Add option where game design can specify use or avoidance of masks with unit icons.

Player could escape a loss by saving a game, then discarding save. Mplayers could register suspicion when player saves then quits - "You're not trying to cheat, are you?" - but can't prevent this.

All interfaces should be able to bring up an "Instructions" window that informs player(s) about the current game, includes xrefs to all game design info. Restrict help to generic and interface info only.

Graph display should graphing of various useful values, such as amounts of units and materials over time, attitudes of sides, combat, etc. Maximal is timeline for all sides and units, usually too elaborate but allow tracking movement for some "important" units. Note that move actions may be recorded anyway.

Make specialized dialog for agreements, put name on top, then scrolling list of terms, then signers, then random bits (public/secret, etc). Use for proposals also, so allow for "tentative" signers, desired signers who have not looked at agreement. Be able to display truth of each term, but need test to know when a side can know the truth of a term?

Interfaces should have a "wake up dummy" button that can be used by players who have finished their turn, to prod other players not yet done.

Commands that are irrelevant for a game ought to be grayed out in help displays, and error messages should identify as completely invalid (or just not do anything, a la grayed Mac menu shortcuts).

Should be able to drag out a route and have unit follow it (user input of a complete task sequence).

Hack formatting so that variable-width fonts usually work reasonably.

Add xref buttons to various windows to go to other relevant windows and focus in.

The current turn or date should be displayed prominently and be visible somewhere by default.

Add some high-level verbs as commands ("assault Berlin", "bomb London until destroyed").

Don't draw outline boxes at mags that would let them get outside the hex.

If dating view data, allow it to gray out rather than disappear entirely. Could even have a "fade time" for unit images...

Even if display is textual, use red text (and other colors) to indicate dangerous conditions.

Next/prev unit controls should change map focus, even if screen unaffected.

In general, ability to "select" a unit implies ability to examine, but not control. Control implies ability to select, however.

Use a builtin color matching a color name if possible, otherwise use the imc definition.

Connections may need to be drawn differently in each of the two hexes they involve, such as straits connecting to a sea. (what is this supposed to mean?)

If cell cramped for space, show only one material type at a time, require redraw to show amounts of a different type.

Draw time remaining both digitally and as hourglass, for all time limits in effect.

Could tie map to follow a specified unit (or to flip there quickly a la SimAnt).

Have a separate message window from notices, allow broadcasting w/o specific msg command? (a "talk" window)

Redraw hexes exposed when a unit with a legend moves. Truncate or move legend if would overlap some other unit/legend.

Put limits on the number of windows of each type, set up so will reuse windows, except for ones that are "staked down".

Fix border removal so inter-hex boundary pixels are cleaned up also.

Need a specialized window or display to check on current scores (showing actual situation vs what's still needed). (Show both scorekeepers actually in force, as well as the others.) Side display could also display scores relevant to that side.

Every unit plan display should have a place to record notes and general info about the unit, add a slot to units also. Use in scenarios.

Need a command for when a player can explicitly change the self-unit.

Players should be able to rename any named object. The interface should also provide a button or control to run any namer that might be available to the unit.

Be able to select unit number display indep of unit name display, and feature name display indep of unit names.

Don't draw things that xform to 0 pixel areas, only draw the most important things if 1-4 pixels or so.

If time/effort to do action is > length of game, then interface can disable that action permanently.

Use moving bar or gray under black to indicate reserve/asleep units.

5.3 Networking

Xconq has been also been designed to allow for different kinds of networking strategies.

The kernel/interface architecture can be exploited to build a true server/client *Xconq*, by building an "interface" that manages IPC connections and calling this the server, and then writing separate interface programs that translate data at the other end of the IPC connection into something that a display could use. My previous attempt at this (ca 1989) was very slow and buggy, though, so this is not necessarily an easy thing to write. The chief problem is in keeping the client's view of thousands of interlinked objects (units, sides, cells, and so forth) consistent with the server. Most existing server/client games work by either restricting the state to a handful of objects, or by only handing the client display-prepared data rather than abstract data, or by reducing the update interval to minutes or hours.

[When networking, all kernels must call with same values...]

5.4 Miscellany

5.4.1 Versioning Standards

In version *7.x.y*, *x* should change only when some documented user-visible aspect of *Xconq* changes, whether in the interface or kernel. In particular, any additions to GDL, such as a new table or property, require a new *x* version. *y* is reserved for bug-fix releases, which can include the implementation of features that were documented but not actually made to work.

5.5 Pitfalls

This chapter would not be complete without some discussion of the traps awaiting the unwary hacker. The Absolute Number One Hazard in hacking *Xconq* is to introduce code that does not work for *all* game designs. It is all too easy to assume that, for instance, unit speeds are always less than 20, or airbases can only be built by infantry, or that worlds are always randomly-generated. These sorts of assumptions have caused no end of problems. Code should test preconditions, especially for dynamically-allocated game-specified objects, and it should be tested using the various test scripts in the test directory.

The number two pitfall is to not account for all the possible interfaces. Not all interfaces have a single "current unit" or map window, and some communicate with multiple players or over a network connection.

You should not assume that your hack is generally valid until you have tested it against everything in the library and test directories. The `test` directory contains scripts that will be useful for this, at least to Unix hackers. See the `README` in that directory for more information.

Another pitfall is to be sloppy about performance. An algorithm that works fine in a small world with two sides and 50 units may be painfully slow in a large game. Or, the algorithm may allocate too much working space and wind up exhausting memory (this has often happened). You should familiarize yourself with the algorithms already used in *Xconq*, since they have already been debugged and tuned, and many have been written as generically useful code (see the area-scanning functions in `world.c` for instance).

If your new feature is expensive, then define a global and compute its value only once, either at the start of the game or when it becomes relevant. Such a global should be named `any_<feature>`.

Similarly, complicated tests on unit types or sides should be calculated once and cached in a dynamically-allocated array.

5.6 Rationale and Future Directions

This is where I justify what I've done, and not done.

Please note that although *Xconq* has considerable power, its design was expressly limited to a particular class of two-dimensional board-like strategy games, and that playability is emphasized over generality. For instance, I avoided the temptation to include a general-purpose language, since it opens up many difficult issues and makes it much harder for game designers to produce a desired game (after all, if game designers wanted to use a general-purpose programming language, they could just write C code!). Similarly, full 3D, realtime maneuvering, continuous terrain, and other such goodies must await the truly ultimate game system.

The real problem with a general-purpose language is that although everything is possible, nothing is easy. Many "adventure game writing systems" have fallen into this trap; they end up being poor reimplementations of standard programming languages, and the sole support for adventure gaming amounts to a small program skeleton and a few library functions. It would have been easier just to start with a pre-existing language and just write the skeleton and libraries!

Xconq, on the other hand, provides extensive optimized support for random game setup, large numbers of units, game save/restore, computer opponents, and many other facets of a game. Game designers don't have to deal with the subleties of fractal terrain synthesis, or the ordering of terrain effects on units, or how to tell the computer opponents that airbases are sometimes good for refueling but never any good for transportation, or the myriad of other details that are wired into *Xconq*. In fact, a complete working game can be set up with less than a half-page of GDL.

Even so, the current *Xconq* design allows for several layers of extensibility, as was described earlier in this chapter.

There are also several major areas in which *Xconq* could be improved.

Tables should be supplemented with general formulae, although such formulae will complicate AIs' analyses considerably, since tables are much easier to scan. Formula-based game definition would work much better with AIs that are coded specifically for the game and compiled in; this is more-or-less possible now, but there is not yet a good way to keep AIs from being used in games where they would be inappropriate (it might be amusing to have a panzer general AI attempting to play Gettysburg, but the coding would have to be careful not to try to index nonexistent unit types).

Currently everything is based on a single area of a single world. This could be extended to multiple areas in the world, perhaps at different scales, as well as to multiple worlds.

However, even with its limitations, *Xconq* has provided, and will continue to provide, many years of enjoyable playing, designing, and hacking. Go to it!

Appendix A Glossary

Some of the concepts below are only relevant to certain interfaces, but are sufficiently frequent to warrant inclusion; they are identified as interface-specific.

accident: A random event that damages or destroys a unit.

acp: See action points.

action: A single thing that a unit can do. Examples include movement to an adjacent cell, detonation, and repair.

action points (acp): The basic number of actions available to a unit during the turn.

agreement: A treaty or deal made between players, consisting of a number of terms defining each player's part in the agreeement.

AI: See Artificial Intelligence.

Artificial Intelligence (AI): A player that is run by code internal to *Xconq*.

altitude: the z-coordinate of a unit, relative to its cell's elevation.

area: A section of the world that you play on. An area may be a polygon, or a cylinder if it is large enough to go all the way around the world.

attrition: A gradual loss of a unit's hp, usually due to a harmful environment.

backdrop: The set of activities that proceeds independently of units and sides.

border: A special type of terrain that occurs between two cells.

border slide: A special kind of unit move that traverses a border rather than crossing it.

cell: A single location in an area, typically a hex or square shape.

closeup: A part of an interface that displays detailed information about a side or unit.

coating: A layer that temporarily modifies terrain, such as snow.

combat experience points (cxp):

completeness points (cp): The degree to which a unit is ready to do things. Newly-created units usually fall short of completeness.

completion: A type of action that brings a given unit closer to being complete.

connection: A special type of terrain that negates the effects of cell and border terrain between two adjacent cells. Examples would be roads and canals.

construction: A general term referring to the combination of creation and completion actions that result in a usable unit.

consumption: The process by which units and terrain use up materials.

country: The initial region of a side's units.

coverage: A set of numbers representing the quality of a side's vision at each point in the world.

cp: See completeness points.

cxp: See combat experience points.

creation: An action by which a unit creates another unit. Creation actions can specify that the newly-created unit is to be at a given location or inside a given unit.

designer: A special kind of side/player that is permitted to examine and alter the entire state of the game directly.

detonation: An action that results in hits on every unit near the detonation.

disband: An action that results in the orderly destruction of a unit.

doctrine: A set of flags and parameters that individual units of a side use to help decide what to do, in the absence of explicit orders from the player.

elevation: The height of a cell above some arbitrary point.

emblem: An iconic image used by interfaces to display a side.

event: a historical occurrence.

feature: see geographical feature.

game design: The set of type definitions and rules of a game, usually composed from several game modules.

Game Design Language (GDL): The language used to define *Xconq* games.

game module: A group of game-related definitions and information, not necessarily a complete game design.

geographical feature: A named region in the world.

goal: A state or situation that units and/or sides can plan to achieve.

grammar: A set of rules that define how a name or phrase will be constructed from letters and/or syllables.

hex: A cell in a world where each cell is adjacent to exactly six others.

history: The record of events in a game.

hit points (hp): The amount of damage that a unit can sustain before it dies or is otherwise destroyed.

hp: See hit points.

image: A visual icon or pattern used by interfaces to display units and terrain.

image family: A collection of images, of various sizes, orientations, colors, etc, all of which represent a single visual concept.

independent unit: A unit that does not belong to any side.

interface: The software that manages interaction between a player and the kernel.

kernel: The part of *Xconq* that manages the action of the game itself.

list: A linear list of units or sides (interface).

map: A visual display of part or all of a world (interface).

material: A quantity of a material type.

material type: A type of mass stuff.

movement: A type of action in which a unit changes its location. The destination of movement may be either a cell or another unit.

movement points (mp): The basic amount of mobility available to a unit during a turn.

mp: See movement points.

namer: An object that generates names using a naming method.

naming method: An algorithm for generating names.

occupant: A unit that is contained in another unit. See transport.

pattern: A special kind of image that can be repeated many times in a regular fashion. Usually used to display terrain.

people: Special "material types" that are actually considered to represent individuals.

plan: The information that a unit uses to decide what to do next. Both human-run and AI-run units have plans.

player: A participant in a game. Can be a human or a computer.

production: The process by which units and terrain make materials appear.

property: An attribute or value associated with a type or object, such as a unit type's speed or a side's name.

region: A set of cells in the world. It can be any size or shape.

savefile: A special game module that contains an exact replica of a game in progress.

scorekeeper: An object that manages (part of) the standings of players in a game, and also handles recording of final scores.

self-unit: A unit that represents the whole side.

side: The representation of a single player within the game.

speed: The ratio of movement points to action points.

spying: A backdrop process where units collect information about units on other sides.

stack: A group of units at the same location, none of which are inside another.

supply: The materials being carried by a unit.

supply line: A path by which a unit can get supplies automatically from another unit.

synthesis method: An algorithm that can build part of the initial game setup, usually randomly.

table: A two-dimensional array of numbers that define some interaction between pairs of types. Tables may be indexed by unit types, material types, and terrain types, in various combinations.

task: A single element of a unit's plan. A task usually results in one or a few unit actions.

task agenda: A list of tasks that a unit plans to do.

tech level: An abstract number representing a side's ability to use and/or construct a type of unit.

terrain type: One of a set of possible terrains.

terrain subtype: The specific role played by a terrain type. Subtypes currently include open terrain, borders, connections, and coatings.

tooling points (tp): A unit's amount of preparation towards the construction of a particular unit type.

tp: See tooling points.

transfer: The process by which supplies/materials are moved from one unit or cell to another.

transport: A unit that may contain other units. See occupant.

turn: A single cycle of unit actions and backdrop operation.

unit: A single distinct object, like a playing piece.

unit type: One of a set of possible types for the units in a game.

vanish: The process whereby a unit is removed entirely from the game.

variant: A predefined option by which players can alter a game before starting it.

vision: The mechanism by which a side's units collect and report information about the world and about other units.

world: The entire space within which units move around.

wreck: The process whereby an about-to-die unit changes into another type of unit instead of dieing.

zone of control (zoc): A region around which a unit can affect the behavior of other units.

Appendix B Summary of GDL Syntax

Whitespace is never significant, except to separate two symbols or within a string or escaped symbols.

[describe rest of lexical stuff?]

```
form ::= module-form
           | ( include [ if-needed ] module-name [ variant-set ] * )
           | ( if test-form [ symbol ] )
    | ( else [ symbol ] )
    | ( end-if [ symbol ] )
    | world-form
    | area-form
    | side-form
    | side-defaults-form
    | independent-units-form
    | doctrine-form
    | player-form
    | agreement-form
    | unit-form
    | unit-defaults-form
    | scorekeeper-form
    | exu-form
    | evt-form
    | battle-form
    | unit-type-form
    | terrain-type-form
    | material-type-form
    | namer-form
        | table-form
        | add-form
        | ( define symbol value )
        | ( set symbol value )
        | ( undefine symbol value )
        | value

module-form ::= ( game-module [ module-name ]
                   [ game-module-property-binding ] * )

module-name ::= string

game-module-property-binding ::= ( game-module-property-name value )

game-module-property-name ::= title | blurb | picture-name | base-game
| instructions | notes | design-notes | version | program-version
| base-module | default-base-module
```

```
variant-definition ::=
    ( [ string ] var-type [ var-default ] [ var-range ] [ var-clause ] * )

var-type ::= world-size | world-seen | see-all
                 | sequential | real-time | symbol

var-default ::= value

var-range ::= ( value value )

var-clause ::= ( [ string ] value [ form ] * )


variant-set ::= ( var-type value )


world-form ::= ( world [ circumference ] [ world-property-binding ] * )

world-property-binding ::= ( world-property-name value )

world-property-name ::= circumference | axial-tilt


area-form ::= ( area [ width [ height ] ] [ area-restriction ]
                 [ area-property-binding ] * )

area-restriction ::= ( restrict width height x y )

area-property-binding ::= ( area-property-name value )
| ( terrain [ layer-subform ] * [ string ] * )
| ( aux-terrain terrain-type [ layer-subform ] * [ string ] * )
| ( features feature-list [ layer-subform ] * [ string ] * )
| ( material material-type [ layer-subform ] * [ string ] * )
| ( people-sides [ layer-subform ] * [ string ] * )
| ( elevations [ layer-subform ] * [ string ] * )
| ( temperatures [ layer-subform ] * [ string ] * )
| ( winds [ layer-subform ] * [ string ] * )
| ( clouds [ layer-subform ] * [ string ] * )
| ( cloud-bottoms [ layer-subform ] * [ string ] * )
| ( cloud-heights [ layer-subform ] * [ string ] * )

area-property-name ::= width | height | latitude | longitude | cell-width

layer-subform ::= ( constant n )
        | ( subarea x y w h )
| ( xform mul add )
| ( by-bits )
| ( by-char string )
| ( by-name name-list )
```

```
side-form ::= ( side [ side-id ] [ side-property-binding ] * )

side-defaults-form ::= ( side-defaults [ side-property-binding ] * )

side-property-binding ::= ( side-property-name value )

side-property-name ::= name | long-name | short-name | noun | plural-noun
        | adjective | color-scheme | color | emblem-name | names-locked
| class | active | status | advantage | advantage-min | advantage-max
| controlled-by | trusts | trades | next-numbers | unit-namers
| feature-namers | tech | init-tech | terrain-view | unit-view
| unit-view-dates | turn-time-used | total-time-used | timeouts
| timeouts-used | finished-turn | willing-to-draw | respect-neutrality
| real-timeout | task-limit | doctrines | doctrines-locked
| self-unit | priority | scores | ui-data | ai-data | player

player-form ::= ( player [ player-id ] [ player-property-binding ] * )

player-id ::= number

player-property-binding ::= ( player-property-name value )

player-property-name ::= name | config-name | display-name | ai-type-name
| password | initial-advantage


agreement-form ::= ( agreement [ agreement-id ]
                        [ agreement-property-binding ] * )

agreement-property-binding ::= ( agreement-property-name value )

agreement-property-name ::= type-name | title | terms | drafters
| proposers | signers | willing-to-sign | known-to | enforcement
| state


unit-form ::= ( unit [ unit-id ] [ unit-property-binding ] * )

unit-defaults-form ::= ( unit-defaults [ reset ] [ unit-property-binding ] * )

unit-property-binding ::= ( unit-property-name value )

unit-property-name ::=  | z | s | # | n | nb | cp | hp | cxp | mo
| m | tp | in | opinions | x | act | plan


doctrine-form


exu-form

evt-form ::= ( evt turn type observers [ data ] * )
```

```
table-form ::= ( table table-name [ table-clause ] * )

table-clause ::= value | ( type-or-types type-or-types value )

add-form ::= ( add type-or-types property-name value )
| ( add table table-name [ table-clause ] * )

type-or-types ::= type | ( [ type ] * )

value ::= number
         | symbol
         | global-variable
         | ( [ value ] * )
         | ( operation-name [ value ] * )

operation-name ::= quote | list | append | remove

global-constant ::= true | false

global-variable ::=
| advantage-min | advantage-max | advantage-default
```

Appendix C GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DE-

FECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix D Index

B

C

D

E

S

T

U

Table of Contents