WWW
HTTP://WWW.AURAN.COM

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The Official
Tactics Engine
Configuration Guide

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Conditions of use

Permission to use the manual contained in this site ("Manual") is conditional upon you agreeing to the terms set out below. Do not proceed to download the manual until you have read and accepted all the conditions. If you do not wish to accept the conditions, do not download the Manual.

The user is granted a non-exclusive licence to download and use the Manual contained in this site on the following conditions;

1. The Manual download from this site remains the property of Auran;

2. The Manual downloaded from this site, in whole or in part may be used:

(a) to configure the Tactics software engine; or

(b) in conjunction with other software packages or programs,

provided that:

(c) such use is not for a commercial purpose or any financial gain; and

(d) these conditions are included without alteration wherever the Manual is reproduced;

3. Auran cannot warrant the performance or the results obtained from using the Manual contained in this site;

4. Auran makes no warranties, express or implied with respect to the Manual as to merchantability or fitness for purpose or nonconfringement of third parties rights but in the event that any legislation implies terms which cannot be lawfully excluded, such terms will apply except that the liability of Auran for breach of any such implied term will be limited to replacement of the Manual to which the breach relates or the supply of an equivalent manual;

5. Auran will not be liable for any damages whatsoever including;

(a) direct, indirect, incidental, consequential damages; or

(b) loss of business profits; or

(c) special damages,

arising from the use of the Manual contained in this site, even if Auran has been notified of the potential for such damages to arise;

6. The user acknowledges:

(a) they have not made known to Auran any particular purpose for which the Manual contained in this site is required; and

(b) they have not relied on Auran to provide the Manual as being suitable for any such purpose.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

CONTENTS

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Introduction

Auran's Tactics Engine was designed to allow the reconfiguration of any game developed with it, as well as the creation of new games. The engine houses many advanced features not available in other current strategy games and lends itself to amazing configuration possibilities. This allows game design to become a much easier process by not involving the programming step, which requires a great deal of resources.

The Tactics Engine was constructed as a basis for a real time strategy game engine with features far more advanced than current games. It was created in such a way to allow people to make changes to games made with it, or to make entirely new games. The engine was designed so that it could realise the dreams of game designers who perhaps did not have the coding advantages, but could also be used to design games at the leading edge of the gaming world.

An intense effort was made at Auran to always remember that it was an engine being coded, not a game in pure hardcoded form. Of course, another factor was to ensure that the availability of this power should not detract from the game's strengths in any way.

Section 1 - Engine Capabilities

This section lets you, the designer of a game utilising the Tactics Engine, in on a lot of the finer points of the engine, so that you can understand its general parameters. While there are limits to the configurability of the engine, you will soon discover that they are beyond anything you've seen before.

Given that the first aspect of the game a designer is likely to conceive is the unit structure of the sides, it is this part of the engine we will discuss initially.

Basic Unit Characteristics

There are a few basic characteristics units need to have to participate in a game using the Tactics Engine.

A standard unit has these characteristics:

- a movement type
- response and select sounds
- a side number designation
- cost (both in credits and time)
- hitpoints
- hitsize (radius that the unit's physical size should take up)
- seeing range
- armour class
- flag designating whether the unit is human or not
- a weapon

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

A unit doesn't *need* all of these characteristics, these are merely the attributes that a standard unit will have. There are also other special characteristics a unit can have.

Special Unit Characteristics

These are:

- morphing into units and/or overlays
- the ability to build buildings
- sabotaging ability
- phasing ability
- healing
- repairing
- RunDownInfo
- veteran ability
- self destruct ability
- camouflage ability
- the unit is attached to a building
- if the unit can transport other units
- does the unit have an alternate state, and if so, what is it?
- the ability to spy on other teams
- the ability to convert a human unit to a certain type
- the ability for the units weapon to "charge" (showing a bar representing the time of the weapon charging)

For simplicity we will refer to two different classes of units: simple units and complex units.

Simple Units

Simple Units are comprised of a single part, with the movement ability and the unit's weapon being attached to the same part. Examples of simple units are human units and larger units which have guns fixed to the same direction as the unit, like artillery. This means that units with this type of configuration cannot move and fire simultaneously. This is because any single part can only have one concurrent order. Orders include commands such as "move", "fire" and "sabotage".

Complex units

Complex units have more than one part. Tanks are an example of when this sort of configuration is necessary. Tanks need to have guns (turrets) that can fire in any direction, more or less irrespective of the facing direction of the unit. Complex units, because of the fact that the top part is independent of the bottom part, can fire at the same time as they move. The ability to fire upon units while moving is a huge advantage, both in defence, and while attacking, and it's a factor that shouldn't be overlooked during play balancing.

Basic Characteristic Definitions

Movement types.

The movement type of a unit determines the way in which the unit moves relative to the terrain. There are five different types of movement in the engine: Ground; Hover; Fly; Fixed; and Tunnel.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Ground Movement
The Ground movement makes the unit move by first turning to the direction in which it must head. This type of movement makes it effective for most ground based vehicle and unit types, such as human feet and tracked vehicles. The movement is fairly jerky, indicative of the units which use it. Units with this movement style can either move, or turn, but cannot do both simultaneously. The one exception is when the unit is already moving in one direction, and then must turn to a direction that is less than or equal to the rotation rate of the unit - it can complete the turn in less than one unit cycle. Ground units pay attention to all overlay effects, including building effects.

Hover Movement
The Hover movement is similar to the ground movement type, except for one small difference. The unit turns to face the direction that it is moving, but moves instantly in that direction regardless of initial movement direction. This effect gives the unit a much more smooth movement, allowing it to appear to glide over terrain, indicative of hover-type units. Hover units have their shadows slightly offset to indicate their separation from the ground. Only one ground unit or one hover unit may occupy any tile on a map. If a unit is phased however, a ground unit or hover unit may occupy the same tile. The phased unit is not able to surface while another unit is above it.

Fly Movement
The Fly movement is similar to Hover in the way that it moves; units with the fly movement type can move while turning. Flying units are at a different level to ground and hover units, and can therefore occupy the same tile as a ground or hover unit. Only one flying unit can occupy any tile at one time.

To hit a flying unit, a weapon used by another unit must be able to specifically hit a flying unit. Any weapon aimed at a flying unit does not do damage to units below or around it, either by way of direct attack damage, or area effect. Any weapon aimed at a non-flying unit, does not do damage to flying units. Flying units create shadows that are not like those coming from hover units. While flying units have shadows that are offset from the main body of the unit, the offset depends on the height of the terrain that the unit is flying over. This makes the unit seem to be of much superior height to ground units. Flying units also demonstrate their superior height with reference to their line of sight. Units with the fly movement type can see better into valleys, and do not lose much seeing range due to lower overlay effects like cliffs.
In the case of flying units especially, it can be ideal for the unit to animate through its movement frames constantly, even while idling (for example, to make a helicopter always fly). Any unit can be given the ability to constantly cycle through a section in it's sprite.

Tunnel Movement
Units with the movement mode tunnel, are special in that they are transparent in appearance. These units travel in an underground layer, and may travel onto the same tile as any flying or hover/ground unit, but may not travel onto the same tile as a phased unit.

Fixed Movement
Units that are to be placed on buildings (through the use of the ActivePart() line) need the movement type Fixed. This movement type should not be given to any other type of unit.

Response and Select Sounds
There are two types of sounds that can be bound to a specific unit. *Response sounds* are played when the unit is given an order: to fire, to move, to build a building. The second type, *s*, are played when the unit is selected on the playing board. The engine can handle up to four sounds for each type, per unit. Of those four sounds, one is chosen at random whenever applicable.

Side Number Designation
Using the Tactics Engine, it is possible to create more sides, or teams, than the standard two that most other real time strategy games allow. Theoretically, it is possible to make dozens of sides, each with their own unit hierarchy. However, because the maximum number of units allowed is restricted due to memory requirements (ie it must run on a machine with 16 meg.) sides can only have a maximum of 200 units between them, making the practicality of having too many sides a key consideration. This includes definitions of veteran units, and alternate units. The engine's shell interface only allows for 4 sides. Side 2 (they number from 0) is always assigned to team 8 (the non-player team).

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Cost
Building units require two factors: money, and time. Units have both a credit cost, and a build time. These two costs are used to limit the game play so that certain units do not end up dominating the game.

Hitpoints
The hitpoints of a unit are its health value, represented as a numerical figure. As a general rule, hitpoints will be balanced against how much the unit costs, and its special abilities.
Units can be healed and repaired by using the healing and repairing functions which can be assigned to units and buildings with the engine. The hitpoints of a unit can be a good indication of how long the unit will last in battle, and should be used accordingly.

Armour Class
The armour classes, using the Tactics Engine, were designed to give maximum flexibility to play balancing, and allow specific weapons to have different attack values depending on the kind of armour a unit has. Armour classes can be defined to have different defences against different offences. For example, it is possible to make a chemical cloud extremely destructive against infantry, but to do no damage against a cyborg; or to increase a tank shell's damage capability to all units with tank armour (heavily plated steel), but to not damage infantry units. There is a maximum of 40 armour classes possible. Each armour class can have a different attack from any of up to 260 different offences. Any given unit need not have a unique armour class, but can have a value that determines a percentage to which any attack on that unit will count towards the overall damage the unit receives. For example, a Large Tank can be given the same armour class as a smaller tank, but if for some reason the smaller tank needed its armour scaled down by a percentage, so that it lasted a little longer, this could be done. Note that the same effect could be given by choosing a higher number of hitpoints for the smaller tank.

Hitsize
Every unit is given a certain hitsize, which is a radius in pixels that describes the physical size of the unit, specifically for the weapon systems.

Seeing Range
The seeing range of a unit is the distance the unit can see in any direction, in tiles. This value is indicative of the unit's seeing range on flat ground only, and the unit's Line of Sight depends on what other obstructions, such as terrain, overlays and buildings, are in the way. The maximum line of sight for a unit or building is 19 tiles.

Human vs Non Human
There are a considerable number of issues relating to the distinction between human and non-human units. Most of these relate to special abilities, which are listed later in this section. Where the ability distinguishes between human or nonhuman a note has been made.. Particularly important issues are repairing and healing. Healing is relevant only to human units, whereas repairing is only for non-human entities.

Prerequisite System
In addition to having a unique identifying string, each unit and also has a unique identification number. Units and buildings are identified by this number within the prerequisite system. The prerequisite system was set up to handle the way units and buildings would act, to make players need certain buildings before they could produce the more expensive and essential units and buildings.

For example, to create a greedy humdinger tank, it might be desired that a greedy humdinger tank factory first be created. Or it might be decided that the greedy humdinger tank should require the greedy humdinger tank petrol facility, as well as the greedy humdinger tank factory.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Any unit can have up to 8 buildings/units as prerequisites. The same is true of buildings. Further, as well as the prerequisites to make sure certain buildings are required to produce a unit or building, it can also be fixed that the building/unit must have a maker as well. A unit can have up to 8 different makers, and as long as one of the makers exists, the unit can be built. The distinction between makers and prerequisites of a unit, is that while all the prerequisites must exist to be able to build the unit, only one of the makers need exist. A building must have units as its makers, and a unit must have buildings as its makers.

Weapons

Any unit may be given one or more weapons. A weapon may be attached to any part of a unit, however it will usually be attached to the top section. Any part may have none, one, or many weapons. Only one weapon can fire at a time. Multiple units can use the same weapon, irrespective of their other attributes. One factor to remember when designing units is that a weapon will always be drawn on top of all other sprites. It is not a good idea, therefore, to design a unit that shoots from underneath it (like a flying unit with a gun underneath and at the centre), because as soon as the projectile is fired, it will appear above the sprite.

Weapon Systems

Movement

Weapon movement has five possible values: Linear, Homing, Vortex, Wave and SelfDestruct.

Due to differences in the way that the attributes behave depending on the type of weapon movement chosen, the weapon types are split up at this point.

Linear, Homing

Linear and Homing movement types apply to all standard weapons. Projectiles fired by these weapons either move in a straight line when fired (Linear), or turn to try and hit the target if the target has moved (Homing). Almost all weapons will take one of these two movement types as they apply to all but a couple of special case weapons. Once the weapon's movement type has been selected, the rest of the weapon can be defined.

Set Offence

A weapon's offence is defined by three things, the offence class (which is used to calculate the final damage to the unit, explained later this section), the attack value (which is calculated in hitpoints), and an area effect (which causes the weapon to damage the area around the tile which is attacked). The area effect is a radius in pixels. The damage that the attack does to units around the target tile is scaled linearly to 0 at the edge of the area effect. For example, if a weapon is aimed at one tile, and another unit is 75% of the area effect away from that target tile, it receives 25% of the attack value.

Projectile Animation

The projectile's appearance is determined by a defined animation. This allows the projectile to change shape, colour and size as it moves, if necessary. The animation for a projectile should be in at least 24 rotations if it is to fly in a definite direction. The number of frames for the projectile's animation should be relative to the amount of time that the projectile will spend in the air when fired its maximum distance. There is no point to having enough frames in the projectile's animation if half of them are not played. Of course, it is completely acceptable to define the animation as continuously playing for simple bullets and laser blasts, but it is also possible to create effects that are more visually powerful..

Fire Explosion

It is possible to define an explosion to take place at the muzzle of the gun when it fires the weapon. This effect can be used to create "smoking gun" effects or gas clouds that follow the projectile out of the barrel.

Hit Explosion

The hit explosion for a weapon is the explosion played at the destination of the projectile's path. This explosion can be used to show the animation of a burning fire cloud. This explosion can also be used to designate the persistent damage effect that is left on ground by a weapon that induces flames or chemical clouds at its destination.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Attributes: range; maxammo; firedelay
A weapon has a set of attributes that relate to its range, the amount of ammunition, and how often the weapon fires. The range is measured in tiles from the firing point. The maximum ammunition is defined in absolute number of projectiles. The firedelay of the weapon determines the weapon's rate of fire. The fire delay is specified in unit cycles, allowing the weapon to fire once every x unit cycles, depending on its type.

Flyer; ground; ground unit; indirect; building; nonhuman; human
Certain weapons need to be able to fire at different types of units, dependant on whether the units are flying, whether they're human, and many other factors.

To allow this flexibility in the weapon types, there are special flags which set the targets for a weapon. A weapon type can have multiple target flags, and indeed, in most cases will. Some of the flags are mutually exclusive, and if used together defeat their purpose. There is a flag to specify that the weapon can shoot flying units, and a flag that allows the weapon to attack ground units.

There is a flag to allow the weapon to attack ground itself. This allows this weapon to destroy overlays. A weapon with this flag could possibly be used to create a successful block to stop an enemy unit travelling over that particular tile. Another flag specifies that the weapon can shoot and hit ground units (units with ground and hover movement types).

There is a flag to specify that the weapon can shoot beyond its seeing range, as long as another unit can see the tile/unit that is to be attacked. This flag is most useful for artillery-like weapons, where the artillery shouldn't be able to see tens of tiles in every direction, but might still want to shoot that far. A weapon with this flag can still only shoot where another unit can see, and this type of firing is called indirect fire. A flag specifies the ability to attack buildings with the weapon. Two more flags split human and non-human units apart, and are mainly for the purposes of separating healing and repairing among ground units.

Speed
The speed of the projectile fired by a weapon is defined to have an initial speed, an acceleration, a maximum speed, and a rotation rate. For a projectile to have a constant speed throughout its flight, the initial speed and maximum speed should both be set to the speed that the projectile should have, and the acceleration should be zero. The rotation rate of the projectile is not useful for linear weapons, but is used in homing weapons. For linear weapons, this attribute should be set to 1. This value specifies how many degrees the projectile can turn per unit cycle while it is homing on the target.

Persistent Damage
Using the persistent damage function, it is possible to create the effect of gas clouds and multiple burning flames to cause damage over time to units on tiles effected by the attack. Other possibilities for this function include chemical mortars that can be used to poison the ground or air other sides units may travel through. Using the damage tables it is also possible to make persistent damage hurt only certain units.

Special Unit Definitions
Morphing
The morphing ability allows units to disguise themselves as particular units and/or overlays. A unit can be given the ability to morph into overlays, to morph into units, or both. A unit which has been given this ability can only morph into units or overlays specified as being able to be morphed .

Building Ability
Any unit may be given the ability to build buildings, however, a unit with this ability cannot make every building on its side. Further, it can be specified that a building can be made by any number of 'makers', which refer to the identity of the construction units. In this way several different unit types could make guard towers, if so needed.

Sabotage
A unit with the sabotage ability can sabotage buildings. To sabotage a building, the unit walks next to an enemy building, at which point the sabotage begins. The sabotage period lasts for an amount

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

of time that is defined within the sabotage definition. After the period has elapsed, the building will have half of its defined hitpoints, and will be running at half efficiency.

Phasing

Phasing allows certain units to "hide" below the surface of the ground. The ability of a unit to phase is defined by the presence of the phasing flag in the unit definition and relies on the player having at least one building of the type specified. Units in the game do not display the ability to phase until the player owns one of the specified type of building. When the 'phase' occurs, the unit can play through a defined section in it's sprite. Units that have been phased are translucent in appearance to the owning player and allies, and are not visible to any other players. The section played while the unit is phasing is seen by all players.

Healing and Repairing

These two special characteristics can be given to units/buildings that have the task of repairing units. There is no way to repair a building with a unit/other building.

RunDown

The ability to run down small units with significantly larger ones was added to stop the conceptually incorrect problem of having small human infantry and the like blocking the path of enemy tanks. It seems unlikely that tanks would stop and try to shoot infantry out of their path, particularly when they should be moving straight to their objective (and this objective might have a time factor related to some strategic goal).

To overcome this situation, it became necessary to give the units an attack value and a defence value for run down. This allows values to determine whether certain units can be run over, and which units can run them over. Any unit can run over other units up to a defined value, but can be equally run over by larger units with a higher attack threshold. For example, a unit with defence value 50 and attack value 20 can run down any unit with a defence value lower than 20. However, this unit could be run down with any unit that had an attack value greater than 50.

Veteran Ability

The veteran ability is a very powerful ability which, due to its flexibility, can be used in a number of ways. A unit becomes a veteran when it has damaged enough enemy buildings through firing which equals a defined number of hitpoints worth of damage. The unit then becomes another unit, which is defined as part of the veteran characteristic. This new unit has hitpoints that are scaled as a percentage of the non-veteran unit's health percentage. For example, if the non-veteran unit is 45% healthy, then the veteran unit, upon creation, will be 45% healthy too, but as a percentage of the veteran unit's hitpoints, not the non-veteran's. Because veterans are, in effect, completely different units to their hosts, they can have a whole string of different abilities to the host unit. For example, a veteran unit could have a different health value, to reflect the fact that their long experience in battle should keep them alive long, or they could have a better weapon, larger line of sight, anything.

Self Destruct Ability

Units can be defined to self destruct when a particular button is pushed in the interface. When the unit self destructs it causes a weapon type to fire and explode at the tile location where the unit was when the button was pressed. This allows any explosion (group of animations) to be played when the unit self destructs. It is also possible to define weapon effects like persistent damage when the unit self destructs. The force of the weapon can be defined to take a percentage attack depending on whether the unit has full health or not. This percentage can be scaled so that the weapon does full force when the unit is 80% healthy, and then drops to 20% attack when the unit is almost dead. This feature has great flexibility in the way that it will damage other units.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Units Attached to Buildings
Sometimes it becomes necessary for a unit to be attached to a building, such as in the case of guard towers. Here, the unit would form the moving part of a guard tower, the part that fires the weapon. These units are typically one part and are simply attached to the building at a defined x y position in pixels.

Units that Transport Units
Units that transport other units can be defined to have a transport carrying facility determined by the number of units that can fit in the transport, and the maximum weight of each unit. The available spaces of a transport are represented by green boxes underneath, which display empty hold places in the transport unit. When a unit enters the transport unit, the hold place turns red to show that there is a unit in place. Therefore, at any given time, it is possible to look at a transport unit and tell how many units the transport is carrying, and how many hold places are still empty. When the facility to carry units is defined, the vehicle is defined to have a certain number of bays, with a max weight per bay. Only units that have a weight less than or equal to the bay weight are able to enter the transport unit.

Alternate State
This feature was introduced to allow a unit to *convert* from one state to another simply by pressing a button on the interface. This permits the conversion from a hover vehicle to a wheeled vehicle, for example, with both units having different graphics to reflect the change. Further, a unit could be changed to fire a different type of projectile.

Section 2 - General Art with the Tactics Engine

Files
All sprites are stored in pack files, or graphics libraries. The engine can use more than one graphics library at any time, so the original graphics libraries can remain unaltered when new sprites are added to the game. Graphics libraries can be used to add new sprites to the engine, such as when a new unit type is added, or to override the built-in sprites with customised sprites.

The tactics engine reads special configuration files to determine which graphics libraries are to be loaded. The configuration file is called packman.cfg, and has the following format:

```
[sprites]
pack=$GAME\graphics\sprites.ftg,$GAME\graphics\moresprites.ftg

[sounds]
pack=$GAME\sndfx\sounds.ftg

[setup]
exec=$GAME\addon\
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

[sprites]
pack=sprites_file[,sprites_file]

Lists all graphics libraries to be loaded. Multiple graphics libraries can be specified on a single line, and must be separated by commas, with no spaces between commas.

[sounds]
pack=sounds_file[,sounds_file]

Lists all sound libraries to be loaded. Sound libraries are treated in much the same way as sprite libraries. See the section on sounds for a further explanation.

[setup]
exec=cfg_file

Lists additional configuration files that are to be loaded. These files have the standard packman.cfg format, and may reside in any directory below the dark directory.

Additionally, the packman.cfg file in the game directory (i.e. dark) will have an additional section to define the location of tilesets.

```
[terrains]
barren=$GAME\graphics\barren
jungle=$GAME\graphics\jungle
```

[terrains]
terrain_name=directory

Specifies the directory that contains files for each tileset. Inside the specified directory must reside a packman.cfg (which has the standard format) file which will list the sprite and sound libraries that are to be loaded when this tileset is in use. This allows for terrain specific sprites and sounds.

The actual order of loading graphics libraries when a scenario loaded is as follows:

1. packman.cfg in the game directory (ie. dark) is loaded. This configuration file will load in all built-in sprites and sounds, as well as defining the directory that contains graphics for each tileset.
2. packman.cfg in the terrain directory is then loaded. Terrain specific sprites and sounds can be specified in this file.
3. packman.cfg in the scenario directory is then loaded. Scenario specific sprites and sounds can be specified in this file.

When a sprite is to be loaded into the engine, the graphics libraries are searched in the reverse order of being loaded. Scenario specific sprites will always override terrain specific sprites with the same name, which will in turn override built-in sprites of the same name.

Terrain Specific Overlays
There are many terrain specific art assets in the game which vary their art depending on the terrain to which they are specific. Overlays can be terrain specific. These overlays can use the palette of the game and the terrain so that they look much better than they would using just the game palette. This is because these overlays will never be used on any tileset except their own.

One of the features of the Tactics Engine is the ability to switch overlays depending on the tileset. A terrain with a volcanic theme, for example, could be converted to an interstellar one, and when the terrain changes, so can the overlays already placed on the terrain. So, by making overlays with the same names have similar effects, sizes, looks and so on, but with different themes, you can easily change the setting for an entire scenario, just by changing the tileset.

EMAIL
HELPDESK@AURAN.COM

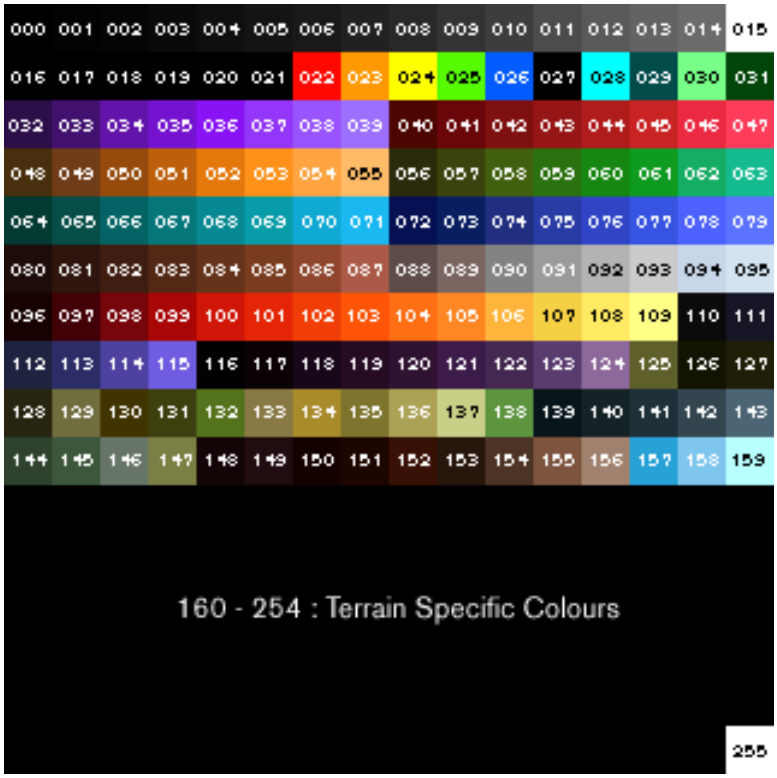
SUPPORT
SUPPORT@AURAN.COM

The Palette

The way that the team colours in the game work is that there are eight colours which are remapped to the team shades of whichever team the player is. The remapping colours are mentioned further in this document.

The palette used by the Tactics Engine is 256 colours. There are special sections in the palette which are used for different reasons and in different elements. Colour index 0 is used as the translucent colour. Any pixels in overlay and unit sprites that have colour index 0 are marked as see through. This is used so that units do not have to fill out rectangular blocks. The pixels around a unit body should all have colour index 0.

Colours 1-15 are greyscale colours that can be used in buildings to represent metallic finishes.



Colours 32 to 39 are the 8 shades of the team remapping colours. When you have the initial bitmaps of your units and you want to dither these source bitmaps to the game palette, you must remember that if colours similar to those in your unit sprites are used, then the program that you use to dither your source bitmaps to the palette used by the game might set the index of some pixels that shouldn't be remapped to team colours to be a close match to those that are the team colours, then the colours would be remapped to team colours when playing with those units. This could cause your units to have too much team colour. For this reason then, the colours chosen for the team remapping colours should be obscure colours that won't be wanted in your sprites anywhere else. If your units will be based around a mainly dark theme, then you should pick a light, fluorescent colour to use as the team remapping colours. An important factor to remember is that team 7 uses the straight team colour without remapping, so in maps with team 7 playing, the remapping colours will be used raw on units.

Colours 40 to 47 are the eight shades of team one that correspond to the eight shades of team colour. Colours 48 to 55 are for team 0, Colours 56 to 63 are for team three. Colours 64 to 71 are for team two, Colours 72 to 79 are for team four, Colours 80 to 87 are for team five, Colours 88 to 95 are for team six.

The colours 96 to 159 are the user definable portion of the palette. This allows the user to choose colours that will be necessary for the interface, units, buildings, and other assets that are not terrain specific.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The remaining 96 colours (160-255), are taken from the palette used by the tileset. These 96 colours are generated when the tileset is created to best fit the colours in the source files for the tileset.

When choosing colours for the game palette, remember that the colours best suited for the tileset are sometimes not all the colours that might be needed for the overlays placed on the terrain. For example, a desert tileset will be mostly sand, dunes and some water. Such a tileset is unlikely to have many green colours. So, if you wanted multiple shades of green, or some other colour which was necessary in the overlays for this tileset, it would be necessary to include those colours in the game palette, to make sure that they are available for the overlays when this tileset is in use.

Note: changing the palette for use by the tactics engine is not a task to take lightly!

Making Tilesets

Making your own tilesets for use by the Tactics Engine is quite a large venture, but if done correctly, will yield excellent results. Because of the nature of the tiling and blending system using the Tactics Engine, the artwork involved in making tilesets can be time consuming, but the engine also does a lot of work to make the tiles meld together properly. It is up to the artist working on the tileset, for example, to ensure that the tiles of the same type put together all join as seamlessly as needed and to make sure that the individual tiles do not seem separate from the rest of the play area. However, the engine blends tiles of different types together, allowing the change from one type of terrain to another to be much smoother. The blending of the tiles is done in real time by the engine to save memory. The blending technique relies on blend masks, which the artist also creates.

In addition to the standard blending of tiles to show standard changes of terrain, there is also special blend and tile joins for the water tile (tile 0). The water tileset needed to be special because the edges of water should not “blend” with dust terrain, but rather water should lie in distinctly separate areas of terrain. A separate set of art is used to show water and land joins. The water tile is the tile that usually is set to animate, to show ripples or bubbles in the water, rather than static, one colour art.

There are sixteen tiles in a tileset, and each has a designated artistic and practical function in the game. While realistically, due to the configurable nature of the speed performance changes to terrain, any terrain type can have any artistic attribute and any speed change, there is a convention set out for the terrain types that allows for the best application of the sixteen different tile types. There are some automated processes in the engine that use this convention, however all of them allow the user to manually override them. One of these is the automatic hill/mountain edging created by the scenario editor. It is an automatic process and can be put to good use, as long as the convention is adhered to. The other important tile to remember is the road tile, which is placed under any building. Any building that you make should appear to fit onto this tile easily. The tiling convention is as follows:

Tile Numbers:

0	Water Animation
1,2,3	Normal Ground
4,5	Mud
6,7`	Rough Ground
8,9	Road tiles. (tile 9 is placed under buildings)
10,11	Slopes
12,13,14,15	Cliff Faces

So, now that the convention typically used for games written with the Tactics Engine is presented, you’ll want to know exactly how the tiles are made, and how the blends and other special tricks are used.

For each of the fifteen standard tile types (fifteen excluding tile 0, which is special and will be referred to later), there are eight different variations. This means that when you lay a lot of the same

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

tiles together, it won't give the landscape a repetitive look. When tiles are placed onto terrain in the map editor, a random tile variation from the type is chosen and laid. Tiles that are placed next to each other of the same type, have no blending or modification to make them appear to fit together, so it is up to the artist to make sure that they do in fact 'tile'. The source bmps for the tiles are placed in a 3x3 grid together, with the bottom right tile in the grid being ignored by the engine (yes, the bottom right tile would be tile variation nine).

Knowing this, the source bmp for a tile's variation is 72 pixels wide and 72 pixels high (3x24). By convention, the tiles in the tile variation set are numbered like this.

There is no special order that these tiles are in, just remember that tile 9 will not be used.

Tile attributes are assigned to the vertices of the tile grid, rather than to each individual tile. For a tile to deduce its characteristics, it uses the four vertices to determine the way that the tile will blend with other tiles, or the behaviour of units that travel across them.

Blending

When two tiles of different types meet, it would be unreasonable and exceptionally limiting to expect them to look as though they melded perfectly together. To facilitate the joins of different tile types, the engine uses a blending technique, which is reliant on 'masks' - different shaped white/black bmps that allow one tile to show through in the black part, and another in the white part. The masks are arranged in 3x3 tile grids, in the same way as the standard tile groupings. However, the engine only uses two tiles out of a mask's grid, tile 2, and tile 4, to use in the blending routine. All other tiles in the mask grid are ignored. Where the tile is a blend of two different tile types horizontally, the engine uses the mask tile number 4. For example, in the case of a typical case where the four vertices of a tile are the tile types as follows . . .

This is outlining a horizontal blend. In this case, the engine uses all the tile 3 variation pixels where the mask at tile position 4 is black, and all the tile 5 variation pixels where the mask at tile position 4 is white. The tile will also blend according to greyscale if the image is not totally black or white.

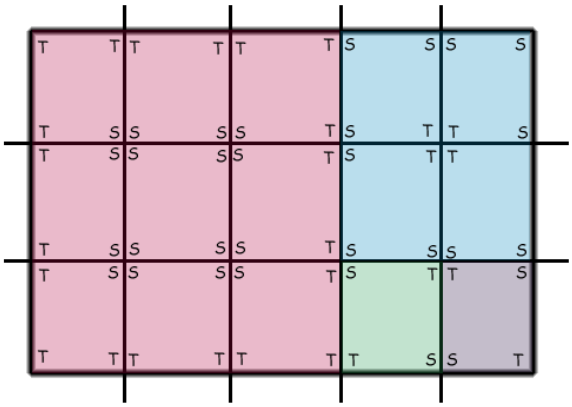
Where the two top vertices are the same, and the two bottom vertices are the same, but different from the top ones, the engine vertically blends, using the blend mask tile number 2 in the mask grid. The top tile is used where the mask is black, and the bottom tile is used where the mask is white.

For any other vertex combinations, where the tiles are blended for example, with three vertices of one type, and one vertex of another, these two mask grid tiles are used in combination to produce the required effect.

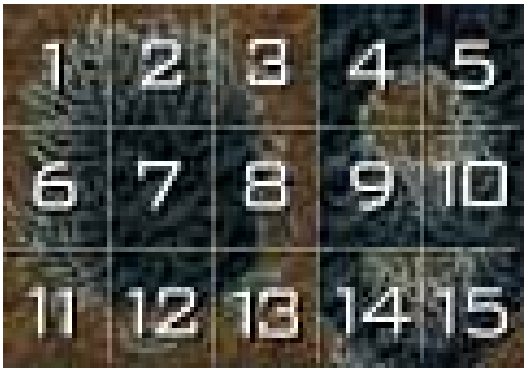
The Special Tile (tile 0)

The Special Tile is so called because it has many attributes that do not conform to the above specifications on blending and tiling. The major change to the special tile is its blending, which occurs in a vastly different way to the standard tiles. Basically, because of the specific nature of the special tile, and its unfriendliness to being blended with other tile types, the special tile has a set of tiles that specifically represent the joins between the special tiles and standard tiles. There are four variations of this tile, and each has its own mask, which allows the seven other terrains to blend to the edges of the change. The first textures to make for the tile types are the blend tiles. To allow for all of the possible other tile types to join to the water edge, the four

variations of the "water's edge" so to speak, also have an individual mask for each one. The masks are designed in such a way as to use the special tile for a bit of special texture, and the edging, and then the standard texture for the rest of the tile. The tile setup for the special to standard blends is a 5x3 tile grid, which is as follows:



The letter 'S' designates a special water tile vertex, and the letter 'T' a standard land tile type. To generate the masks for the special water tile make any area around T black and any area around S white. Ensure that all these sections can seamlessly tile onto another variation or adjacent tile. such as a corner piece. We use a grid numbering system as follows:



We can see that tile 1 shows the blend of one tile vertex special (in the bottom right corner), and all the other vertices standard. The tile would have the bottom right quadrant of the area being a special texture, as well as the joining piece through the centre. The other three quadrants of the tile would be any chosen standard texture, which could include the edging around the special texture, as well as some sort of texture that would be supposed to be shown around the edge of the special texture. For example, if the special texture was water, the edging would show the water joining onto some sort of grass. The mask can be used so that the other terrains blend anywhere into these tiles; for the special blend tiles there is a mask for each tile so that there are no problems in that area. The four tiles 4,5,9,10 are used when there are three out of the four vertices that are special, and only one is a standard tile type. The two other tiles, 14 and 15 are used when there are two special vertices in the 4 for a tile, and they are diagonally opposite. Tile 7 in this tile grouping (the special tile without blending) is not used by the engine.

To get the standard tiles for the special tile type, the engine uses the bmp format in the same way as it does for standard tile, that is, the eight tile 3x3 grid bmps.

Animating Tiles

There is another feature tied to the special tile type, and that is animation. The engine includes the ability to animate tile 0 between a series of frames. This is commonly used to animate water to show bubbling effects. The special tile may have any number of frames of animation for each tile variation. To animate the tile, the engine takes several .bmps as input, and then uses them as the different frames of animation. Each bmp still has the eight tiles - the different bmps are the different frames of the animation. When the tileset is made, the speed at which the tiles animate, and the

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

number of animations to play on screen at any one time are defined. For example, for tile variation number 1 of the 3x3 tile grid, the engine, where the random tile chosen to animate happened to be tile variation number 1, would cycle through the animation, using tile 1 out of each of the 3x3 tile grid bmps in order. The number of cycles that the game plays each frame is defined while making the tileset.

Making Unit Sprites

Units using the Tactics Engine can have many characteristics. Because their actions are not hard coded into their descriptions, as they might be in other real time strategy games, and because the flexibility for users to create their own units had to be included, the process of making your own unit graphics is explained below.

Unit graphics are split up into several sections, to reflect the number of different actions that the unit can perform. The first two most obvious sections are movement, and firing. The movement section is compulsory for all units and unit parts.

There are two distinct groups into which units should be split: units without turrets and units with turrets. For the purposes of this manual, we refer to a turret as another 'part' of a unit, that has separate rotational movement, regardless of where the bottom part turns or moves. An example of this sort of unit is the classic tank, where the turret, or cannon can turn in any direction, while the tracked bottom part drives the tank in whichever direction it needs to go. Units without turrets are the simpler of the two, so we will describe their functions and how to incorporate them graphically first.

By convention, the sections in a unit sprite are arranged in a certain order. The order is:

0. Movement (this section is played continuously as the unit is moving from tile to tile).
1. Firing Animation (this section is played once when the unit fires its weapon).
2. Second Firing Animation (this section is optional and need only be included if the unit has more than one weapon).
3. Special Action Sections (if the unit has special actions like phasing and alternating, then the unit needs a section to play for these actions).
4. Idle Animation Sections (these are the sections that are played when a unit has not been involved in any action for some time - it is essentially a bored animation).
5. Standing Animation (this animation is a single frame that the unit returns to after completing any of the other animations. In this way it can be used as a reference point to make sure the unit stops in a standing position, rather than halfway through a movement cycle. This animation is optional, but necessary to create the right effect.)

This order is for convention only. Because any action can be pointed to in any section in the sprite, the order in which these occur is completely up to the game designer. Each action at its definition takes a parameter that points to the section of the sprite from which the animation is to play. In this way, it is possible for two actions to use the same section of the sprite.

Further, any of the sections (except the movement section) are optional. For example, a unit without idle animations and a standing frame behaves the same way as a unit with them, both of these are just graphical attributes. Units don't need to fire, and don't need special activity sections (unless they are defined to have these attributes).

Units have a certain number of rotations for any section. Rotations are the images of a unit's action which face different ways. For example, if a unit section had four rotations, it would show the unit pointing four different ways evenly through the 360 degree circle. The engine uses the nearest available rotation to the direction that the unit is meant to be facing. The first rotation is always pointing directly right, with each concurrent rotation pointing counter-clockwise from the last rotation. So, in the four rotation example, we can see that the first rotation of the unit would be pointing to the right, the second would be pointing 90 degrees counter-clockwise from right, which would be straight up, the third would be pointing 90 degrees counter-clockwise from straight up, or directly left, and the fourth would be pointing 90 degrees counter-clockwise from left, or straight down.

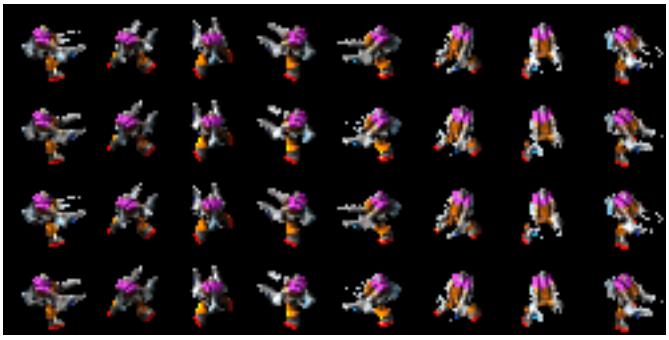
EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SECTION 0

Walk/Movement Animation, 8 rotations, 4 frames of Animation

SECTION 1



Firing Animation, 8 rotations, 4 frames of Animation

SECTION 2

Idle Animation, 8 rotations, 5 frames of animation

SECTION 3

Standing Animation, 8 rotations, 1 frame only, not animated

The above example follows the convention mentioned. There is however no second firing animation and no special action section so all following sections are effectively bumped up into those sections. The only core requirement is that Section 0 is the movement animation for the unit.

The sections of the sprites are broken into frames and rotations. Rotations determine which way the unit will face while the frames are playing. So, for example, to play the animation of a man walking to the right, the engine uses the first rotation out of all the frames in the movement section of the sprite and plays them in order to show the man walking. All the bmps used in a sprite section are the same size. The images are placed in a grid to represent rotations and frames. There is a utility that can do this if the images are rendered into individual bitmaps for each rotation and frame. The grid is set out so that each row is one frame of the animation, in however many rotations

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

needed. For example, if a section had three frames of animation, and eight rotations, then the grid would be three rows deep and eight columns wide. If the unit in question was 10 pixels wide, and 20 pixels high, then the final image section would be 80 pixels wide, and 60 pixels high. Each section in a unit's sprite can have individual sizes, rotations, and number of frames.

When the sprite is made it uses all the defined sections in the sprite. Each section has a different source image. The separate sections and their source images are compiled together by the makesprite win32 executable into the sprite files for use by the game.

Simple Units
For a unit without a turret, the bottom part will both move and fire.

Each unit part has one sprite, and because the bottom part of a simple unit does both the firing and moving for the unit, then the sprite for a simple unit will incorporate firing and moving, as well as any special and idle animations. There are occasions when it may not be necessary for a simple unit to have a weapon. All other sections except for the movement section of a part are optional.

All pixels with colour index 0 in the source bitmaps are tagged as transparent when making the sprite, so any such pixels in the source bitmaps will show up as whatever is underneath the sprite in the game. It is usually a good idea to make this index colour black, for the purposes of viewing the source bitmaps easily against a background. Other than that the source bitmaps should be done in the game palette, rather than using any colours from the terrain specific part of the palette.

Units, like humans and other units that are small in size, can have fewer rotations than the larger ones, and still look correct in the game. The only time where the unit will look any different is when it is firing. The projectile the unit fires has a large number of directions that it can move in.

Of course, this is a matter of balance between what is graphically suitable for the game, and what will and won't look good.

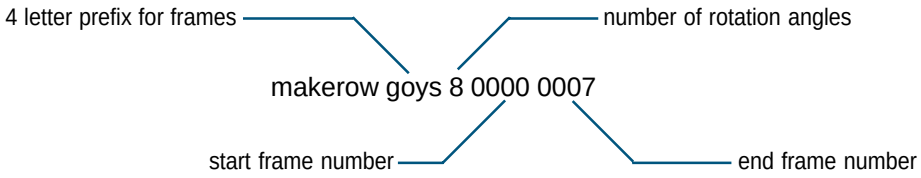
The factor limiting the number of rotations a unit should have is the memory on the destination hardware. Restrictions in making the game suitable for a machine with 16 meg of RAM means that it may not be possible to use 24 rotations for all units (of course, this also depends on how many units you're going to use). Units requiring many different animations can take up very large amounts of memory if the number of rotations for the unit is not limited. For example, making a unit move only in eight rotations looks fine if the unit turns very quickly, or is small in size, but does not look as effective if the unit is to turn slowly, or is relatively large. Again, the balance depends much upon the art in mind and other game play issues.

So, by knowing the number of rotations the separate sections relevant to the unit, you can create your source bitmaps. All bitmaps must be the same horizontal and vertical size. Each bitmap should be an image that is going to be the size of the sprite. Remember that any extra pixels than necessary will be using precious memory, but also remember that each source image must be the same size, so it's likely that some images will have extra space than they seem to need.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The images will then be compiled into their appropriate sections by the use of the makerow program. The makerow program takes a top number and a bottom number, and a parameter for the number of images per row. The top and bottom numbers are taken from the last four characters of the input files. The number of images per row is the number of rotations for the section, and the total number of source images divided by the number of images per row is the number of frames for that section. For example, for a simple 40x40 pixel unit with eight rotations, two frame sections, and the identifier “goys”, the makerow program could be executed in the following form.



Individual frames of the standing animation made into a 1 frame of animation row called goysrow.bmp
If you have more than one frame of animation you will have a file that had multiple 8 rotation rowss.

This would output a bitmap called goysrow for use as a section source in a sprite, with the size of 320x80 pixels. Remember that, because the makerow program outputs a file called identifierrow.bmp, care should be taken to choose different identifiers when working from the same directory, or the files should be renamed.

Once the rows have been made, then the rowed images are ready to be converted into unit sprites which can be used by the engine.

In addition to the graphics for the unit, there is also other information which needs to be in the sprite file, regarding the speed at which the section frames are played when their animation is called upon, and the remap colours for the sprite. The remap colours for a sprite are always 32 to 39, and cannot be changed. This is the remnant of an old system which is no longer supported. For now, these numbers are just padding. The speed of the animation playing is defined as a number of unit cycles, and can be fractional. Fractions of a unit cycle will cause some of the frames to not be played, so that the animation still takes the right amount of time to play. The sprite also needs to know the order of the sections, and which sections take which source bitmaps. The program takes this information from a text file, which can be called anything. This file is a concurrent list of all the sprites that are to be made. The file can contain information about only one sprite, or it can contain information for all the sprites in the game. The file can be created by a dos/windows text editor. The layout of the file is as follows:

```
sprite   = goys.spr
xsize    = 40
ysize    = 40
minremap = 32
maxremap = 39
section  = goysmove.bmp 1.0
section  = gostfire.bmp 1.0

sprite   = tank.spr
xsize    = 120
ysize    = 96
…
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The *sprite* line defines the sprite to be created using the source bmps in the following definition. This filename is the file to which the program outputs the finished sprite, and can be any filename using the standard dos format. The *xsize* and *ysize* defined afterwards are the width and height of the unit, in pixels. *Minremap* and *maxremap* are padding. After this, the sections are defined.

The order of the *sections* in the sprite follow the way that the sections are defined in the sprite definition. So then, for the goyt unit, there are two sections, a movement and a firing section, with the source names goytmove.bmp and goytfire.bmp respectively. The makesprite program reads through the file and then assigns the sections to the sprite in order. Each section is defined by its source bmp, and then a positive number which tells the engine how many sprite frames should be played per game frame. If the number is 1.0, then the game plays each frame of the section for 1 game cycle. If the number is 0.5, then 0.5 frames of the animation are played every game frame, or one sprite frame every two game frames. So, a 2 frame section would play for 4 game cycles. Conversely, if the number was 2.0, two sprite frames would play every game frame, meaning that the player would only see half the movement frames, because one would be skipped every frame. Any decimal in between will work also. Just remember that the if the each frame is sped up, the player might not see all frames in the section.

For every sprite you wish to make add another definition into the text file, with a line of white space to separate them (as in the example above).

After completing this step, save the file and exit the editor. The makesprite program is then run from the command prompt, in the following way:

makesprite textfile.ext

The makesprite program will read the file textfile.ext and make all the sprites that are referenced.

Once the sprite has been made in this way, it is ready for use in the game.

Shadow Sprites
The engine has the ability to offset the sprite of a unit in a greyed out form to make it look like the unit has a shadow, however, often this shadow is not satisfactory, and for this reason it is possible to create shadow sprites to be used for the shadow of the unit. This facility is also provided for making building and overlay shadows. The shadow sprite needs only a single hotspot (hotspot 0), and this hotspot is matched against hotspot 0 on the unit/building/overlay that the shadow is for.

The format for a shadow sprite is a bitmap that has been made in exactly the same way as the objects sprite bitmap (same number of frames, rotations, sections). The bitmap should be 256 colour, and should have at least one colour with RGB values 255 255 255, and one colour with rgb

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

values 0 0 0 (white and black). The makesprite program is run in exactly the same way as for a normal sprite, however, the line sprite = goyt.spr reads, shadowsprite = goyt.spr. Example;

```
shadowsprite   = goyt.spr
xsize    = 40
ysize    = 40
minremap = 32
maxremap = 39
section  = goytmove.bmp 1.0
```

These shadow sprite sections can be in same text file as the sprite sections, so when the makesprite program is run (makesprite textfile.ext), the makesprite program will make both the sprites and the shadow sprites.

Complex Units

Complex units are made in the same way as simple units, but there are just a few factors that need to be remembered. Firstly, the complex unit has two or more parts, and therefore needs more than one sprite. The unit will need one sprite for each part. The sections needed in each sprite depend on which functions the different parts will use. Assuming a typical two part complex unit, the bottom part of the unit will serve as the movement part, and will not need a firing section. However, the top part will be doing the firing, and so will probably want a complex firing sequence. The movement frame for this part is only used when the turret is being turned, and therefore only needs to be one frame. If the section is made with more than one frame, the rest of the frames are ignored.

Special Animations and Complex Units

For complex units, corresponding sprite sections from all parts of the unit are played when a special action is undertaken.

This means that care must be taken when deciding on the actions to ensure it will look correct when both parts play the specified sections of their sprite. By making one part of the sprite play a translucent section, it could be possible to have a more detailed animation which would coordinate the unit sections better. The problem with the first method is that because it is likely the top section will not run at the same rotation as the bottom section, having the two using coordinated movement when performing a special action is not possible, and at most, could only happen in rare circumstances.

Therefore, depending on the nature of the special action, it may be necessary to want the unit to perform an action based on coordinated movement with both the top parts, and the bottom parts performing integral movement.

The only way this can happen is to use the latter method. The latter method means that although the parts might not always line up together right before the animation is played, the two will always perform the action together. The part out of place before the animation begins would depend on which part has the transparent animation, and which part has the ‘real’ animation.

The part with the transparent animation will not dictate the rotation at which the animation will start. When the translucent part is facing in the opposite direction to the part with the animation at the time the special action is undertaken, then it will most likely look incorrect at the transitional phase. For example, a bottom part which was facing one way one game cycle ago, might suddenly be facing in the opposite direction in one game cycle. The effect that this has on the unit will depend on the art involved, and the standards you, as the designer set for what is visually acceptable.

Having used the model to design and create the unit sections, the sprites for a complex unit are made in the same way as the sprites for the simple units. First use the makerow program to compile the source bitmaps together into sectional bitmaps, and then use the makesprite program to convert the section bitmaps into a sprite for each part of the unit.

Making Building Sprites

Building sprites are much simpler to create than unit sprites. Building sprites are composed of three images, which represent the building at three different stages. The first image represents the building in the construction phase. The second image is used to represent the building in normal game play when the building is fully constructed and healthy. The third image represents the building in the destruction phase, and occurs when the building has been damaged by attacks.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The second image is also used as a preview when the user attempts to build a building. When the building is sold, then the building uses the construction frame to represent deconstruction. Each of these three building views is a different frame in a building sprite. Buildings have only one rotation, which is shown no matter where the building is placed. Buildings use the same colour constraints as the units, including the team colour mappings.

To make a building, first create three 24bit colour bitmaps of the same size to be used for the different views of the buildings.

Buildings can be terrain specific, and can therefore use colours from the terrain palettes. Terrain specific buildings are necessary because their outlines should look like they meld with the terrain on which they are built; an effect hard to create without using the colours from the terrain.

For example, you might want a building on a snow tileset to be covered in snow which is accomplished using the snow tileset. It is possible to have one image of the building that is made from the game palette, and then have terrain specific buildings for only certain tilesets if necessary. Having terrain specific buildings does not use more memory, because sprites which are not used in the game are not loaded. These buildings will only use more memory if they are actually a different size, which is also possible using the text files. It is possible to have completely different games based on which tileset the user plays, or anywhere in between, with only a few buildings, units, or overlays changing from tileset to tileset. This concept will be discussed further in the document.

The makerow program is used to compile the bitmaps together into a column, by using one image per row. The column should then be converted into the 8 bit game palette. The makesprite program is then used to change this bitmap into a sprite for use by the game. For example, for the *numy* building, which is 40 pixels wide, and 30 pixels high, the following technique would be used to first make the row:

makerow numy 1 1 3

This would compile the source bitmaps numy0001.bmp, numy0002.bmp, numy0003.bmp into the row numyrow.bmp, which would be one image wide and three images high. The images then are converted into the game palette, without overlay specific colours. The text file entry to use the makesprite program on this bitmap would be the following;

```
sprite   = goyt.spr
xsize    = 40
ysize    = 30
minremap = 32
maxremap = 39
section  = numyrow.bmp 1.0 hotspot0.bmp hotspot1.bmp hotspot2.bmp
```

The hotspots on a building are used when health explosion animations are played. The animations can use a random hotspot to play on as one of their parameters in the explosion definition. A building sprite can have as many hotspots defined as necessary. Any animation that is played in a health explosion can accept different hotspots. It is possible then, to have an explosion that goes off in one of three random places in the bottom right corner of the building when the building goes below 60% health, and when the building drops below 30% health, animations could be played at the top left corner of the building, by just keeping the ranges separate. Of course, any explosion can play several animations, so each animation could also appear on any designated hotspot or range of hotspots.

Building Sections

The first four sections of a building sprite must conform to the convention. Each of these four sections must be one frame, and one rotation. The convention for building sprites is as follows;

Every building must have at least three sections, these being, construction, normal, and destruction. The construction section is shown while the building is in the construction phase. The normal frame is shown at all times that the construction or destruction sections are not being shown. The

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

destruction section of the building is shown when the building's health is less than 33%. If the building is repaired, it goes back to using section 1 (normal section).

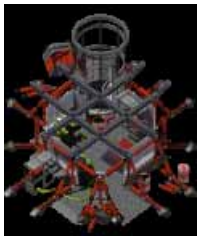
If a building is to be upgraded to another building, then the building sprite will need another section, section 3, that will have the section that is shown when the building is in the process of upgrading. Note that this section *must* be included as section 3 of the building that *will be upgraded*.

So the standard to follow for building sections is as follows

0. Construction frame
1. Completed and operational frame (any idle animations play on this section)
2. Damaged frame, displayed at 33% health (any idle animations play on this section)
3. Upgrade frame (if there is an upgrade for the building or an idle animation
4.+ Idle/Specific Action Animations

Beyond these standard 4 sections, a building may have a special function that can play through a section when it occurs. For example, when a building repairs, it can be defined to play through section 4. These special functions that use sections are explained in the build.txt section of this document.

SECTION 0

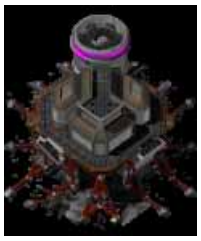


Constrcution Frame

SECTION 1

Completed Frame

SECTION 2



Damaged Frame

SECTION 3

Upgrade Frame

One of these worth noting is the building idle animation. A building idle animation is played when the building is not performing any other function. Something to note, is that the idle animation needs to be able to be played on both the normal frame, and the destruction frame, while still looking visually possible.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

When one of these special animations occurs, the animation is played over the top of whatever building section is being used to denote the building at the time. In this way, only the part of the building that is actually being animated needs to be included in the section. The rest of image part of each frame in the section can be left blank. This saves memory too :).

Making Animation Sprites

There are three other common uses for sprite files apart from units and buildings. These are overlays, weapon projectiles, and explosions. Sprites for this use do not have rotations, but are merely single or multiple frame images. Any animation can be multiple frames, but it depends on the needed effect as to whether or not the animation needs to do this. Each frame can be defined to play for a certain length of time, and the frames can be played in any order, as needed.

Making Isoview Sprites

The isoviews are the sprites that are put into the build lists (in the interface) for both units and buildings. It is a picture that the user will associate with the unit/building. This sprite adheres to certain restrictions. An isoview sprite is 54pixels wide and 40 pixels high. It has only one frame, one rotation, and one section. It is made as any sprite, but doesn't need hotspots. Team colours in the isoview will be remapped.

Hotspots

The hotspot system is used for attaching sprites to either other sprites, or terrain. Hotspots are used because the centre of a sprite is not always the place that another sprite should be referenced against. An example of this is a projectile. A projectile might be long and thin (perhaps a laser bolt). The engine needs to have a pixel point that it references the projectile by, so that it can make the calculations to tell when the projectile has hit the target by pixel, but then has a way of referencing the pixel point on the map that the projectile is, back to the sprite that is displayed for it. The pixel point of a projectile should probably be at the front of the projectile (for every rotation), so that the projectile "hits" the target when the front of the projectile appears to touch the target (the projectile will be removed at this point, and the hit explosion is played), rather than the projectile appearing to pass the target somewhat, and then exploding.

Animation/Explosion Hotspots

Sometimes there may need to be multiple hotspots per unit/building/explosion/animation section. Hotspots always start numbering from 0, and there is a convention that is used for each purpose. The only time that you will configure which hotspot to use is for explosions. When an explosion is defined, you can define a minimum and maximum hotspot. When the engine looks to play this explosion on a unit/building/overlay (health explosions), it picks a hotspot in the range (inclusive) and then plays the explosion on that hotspot, matching hotspot 0 of the sprites defined in sprites used in the explosion, with the chosen hotspot on the unit/building/overlay. One thing to remember is that the hitexplosion() that is played at the destination of a projectile (whether it hits its intended target or not) is always exploded at the detonation point, and is not based on the unit/building/overlay it was targeted at.

Unit Hotspots

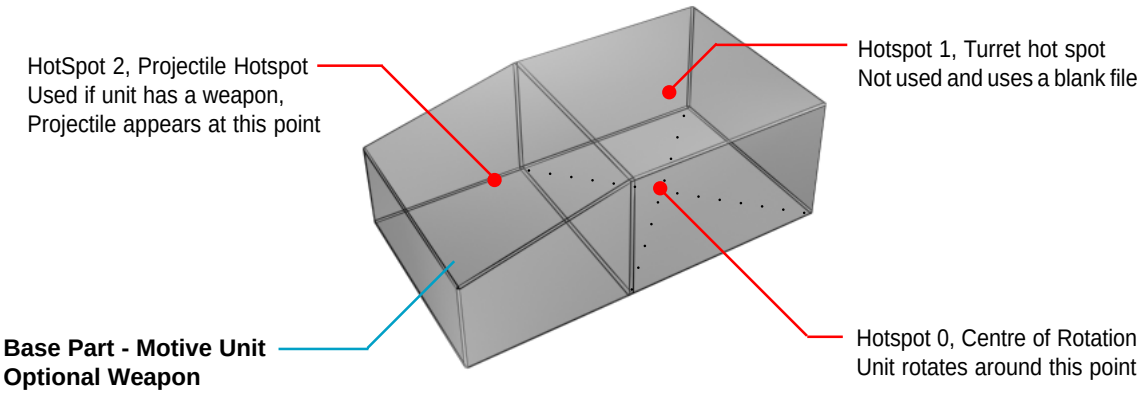
So, hotspots are useful for the placement of explosions, and animations. Hotspots are also used for the placement of unit parts. Hotspot 0 on a unit sprite is defined as the centre of rotation for the unit. The unit is rotated on the pixel it occupies at this point. This works for all unit parts. So the top part of a unit is rotated around this same hotspot. The hotspot on the part under it that the top part attaches to is hotspot 1. This hotspot is called the turret hotspot. So, to place the turret of a unit onto the bottom part, hotspot 1 on the bottom part is matched up with hotspot 0 on the top part.

Hotspot 2 on the unit part is the projectile hotspot. If this part has a weapon, then this is the hotspot of the point of origin of the projectile of this weapon. This hotspot is likely to be placed on the barrel of a gun, for example.

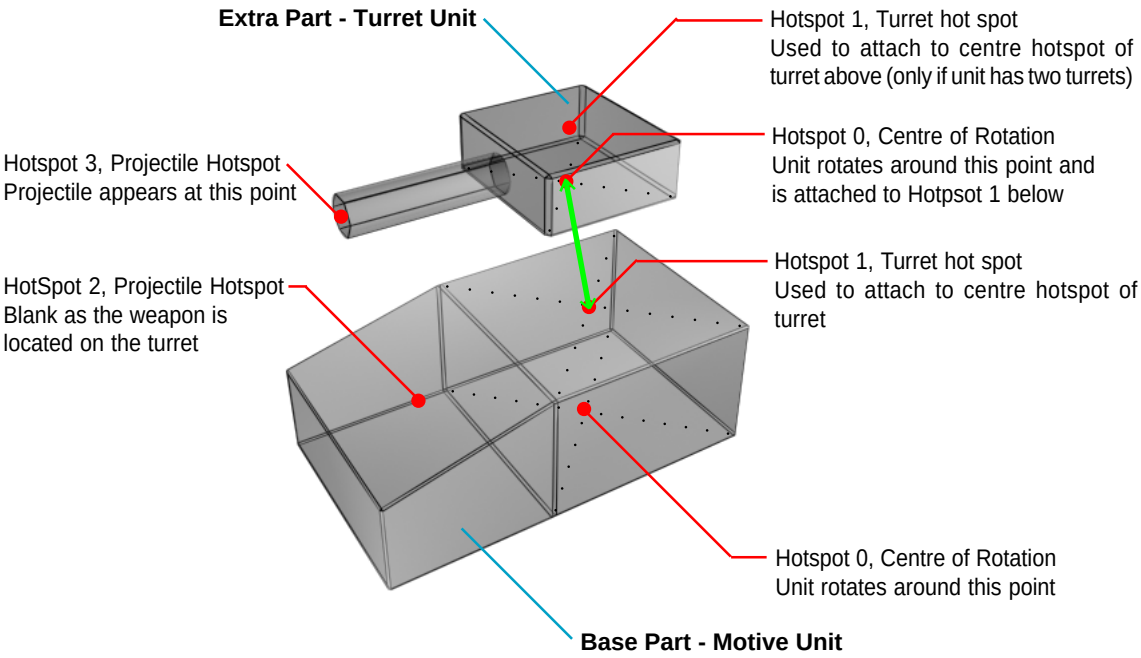
So, the hotspot convention for unit parts:
Hotspot 0: centre of rotation
Hotspot 1: turret hotspot
Hotspot 2: projectile hotspot

Any hotspot greater than 2 can be used for health explosions.

For simple units the following illustration shows where these hot spots should be located.

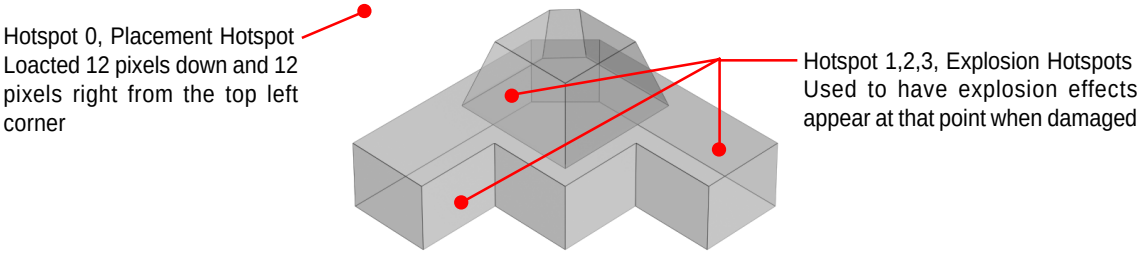


For complex units with a turret it would look the following hotpsots would be needed.



Building Hotspots

Building hotspots are used for placed the graphic of the building on terrain. Hotspot 0 of a building should be 12 pixels to the right, and 12 pixels down from the uppermost left corner of the building sprite. Any building hotspot greater than 0 can be used for health explosions. You only should be put the health explosion hotspots on the sprite used for the top layer of the building.



EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Making Hotspot Bitmaps
The bitmaps that are used for making hotspots are made in the following way. The hotspot bitmap should be the same size as the section bitmap that they apply to. There is a hotspot for every frame and every rotation of the sprite. The hotspot bitmap should be completely black (RGB 0 0 0), except for a white (RGB 255 255 255) point which defines the pixel that should be used as the hotspot. When the makesprite program is processing the bitmap for use as a hotspot, it calculates the average of all white and grey pixels in the bitmap to determine the hotspot. The hotspot bitmap should be 8 bit (256 colour).

Section 3 - Sounds

The sounds specifications for use with the tactics engine are as follows:

format	PCM, .wav file
sample rate	11025 khz
quantization	8 bit mono

The sounds are packed into a pak file using the utility pak.exe, and then are loaded into the engine through the use of the packman.cfg file. Addon sound paks can be loaded from any directory as specified in the packman.cfg file, default game sounds are found in the $game\dark\sndfx directory, in .ftg format.

To make a sound pak, just note the name of the .wav that your file will replace, call your replacement wave file the same thing, and then pak it into a .ftg file, then define your pak file in packman.cfg and bob's your uncle!

Section 4 – Text File Configuration

This section is meant to be used as a supplement to the previous sections that dealt with engine capabilities. Therefore, it delves into many areas with little explanation. If necessary, you may wish to refer to earlier sections that describe the functionality of those points referenced in this section.

General Information about the Textfiles to Keep in Mind
Within the textfiles, it can be easy to create a game so complex that it becomes impossible to trace all aspects by glancing at the definitions in the textfiles. So to ensure their ease of use, you can include comments in the files. When the Tactics Engine loads one of the files, it follows a few rules. One of these rules is that it ignores anything on a line of the text file that comes after a ";" character. This is the way to describe the text files which are completed. A comment might be in the following form.

```
; The file ANIMATE.TXT, which defines all animations and explosions in the game.
DefineAnimationType(blah_animation)   ; This animation is merely for test purposes
; the next line is for the actual definition of the animation.
{
        SetSprite(blah.spr) ; this is the sprite for the Octaschmon unit's missile
        Frame(0 1)
}
```

Your definitions become a lot easier to read later on in the development of a game if these comments are used. They also make it easier to follow the work without having to trace animation names to sprite names throughout the game every time an attribute to a unit or overlay needs to be changed.

We recommend you also think of conventions and standards that will help you while you are developing a game using the Tactics Engine. This will make your work simpler, as the process can become quite involved depending on the scope of the game changes you are intending to implement. Factors that require conventions are likely to be files (graphics, audio), as well as animation names. For example, we kept animation names the same as their source sprite files, but with the word "animation" tacked on the end. For example, the gooby.spr file becomes the gooby_animation. Little steps along the way can make the whole process more effective and easier to manage.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

Animate.txt

So, you want to know how to configure your animations?

The text file animate.txt is a listing of all animations used in the game (animations that are not part of unit sprites). At the bottom of this file, all explosions used in the game are defined. Explosions and animations are two separate things, as mentioned in the first section. Each animation uses a sprite that is referenced in its definition. The explosions and animations are only defined in this order to make sure that the animations that are referenced in the explosions are defined before the explosions in which they occur. If it is necessary for formatting reasons that any explosion can be referenced at any point in the file, so long as all the animations it references are defined previously.

Defining Animations

Having first created the sampanim.spr as shown in Section 2 of this document, this will show you how to define an animation using this sprite.

```
DefineAnimationType(sampanim_animation)
{
 SetSprite(sampanim.spr)
 Frames(0 5 2)
 Frame(6 4)
 LinkAnimation(
}
```

The first line of this animation defines the animation's name. This is the name that the engine will use in other text files to refer to this animation. The sprite that the animation is made from will not be referenced anywhere else in the text files. Every animation must have a unique name, which can use any alpha-numerical characters, as well as the underscore ("_") character. The rest of the animation is then defined in between the curly brackets. The third line in this definition defines the sprite to be used for the animation. This sprite will have a number of frames, but only one rotation. The sprite name should be included in the place of sampanim.spr. The sprite's name can be any number of characters long, in between and including one and eight.

The third line defines the frames that are played in the animation, and the duration for which they will play. The three parameters this line takes are the first and last frame of a series, and the number of game cycles for which each frame plays. When the animation plays, the engine plays the first frame, and then every concurrent frame in the sprite, until it plays the last frame in the series, at which point it stops. It is important to note that the engine steps incrementally higher through the frame numbers. This means you should ensure that the first_frame is a lower number than the last_frame. The format of the line is:

```
 Frames(first_frame last_frame duration_in_game_cycles)
```

The fourth line plays a single frame as part of the animation, which plays for a defined duration of game cycles. The first parameter is the frame number to play, and the second parameter is the number of game cycles for which the game is played. The format of the line is:

```
 Frame(frame_number duration_in_game_cycles)
```

These lines can be used multiple times in a frame definition, and in any order. Also, any line can reference any frame of an animation, regardless of what is used elsewhere in the definition. This allows the same frame of a sprite to be played several times through an animation. For example:

```
DefineAnimationType(sampanim_animation)
{
 SetSprite(sampanim.spr)
 Frames(0 5 2)
 Frame(6 4)
 Frame(3 2)
 Frame(6 10)
}
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The link animation line in the definition allows the animation to be linked to a previously defined animation, or the animation currently being defined. This allows the animation to link to another animation previously used, and has the added flexibility of being able to link to an animation that can use a different sprite. If an animation is to play constantly (for example, most standard overlays like trees, rocks), then the animation can be linked to itself, by putting the name of the animation being defined in the Link() line. The other way that animations can be linked is to link them to a previously defined animation. Note that animations can only be linked to animations that occur earlier in animate.txt, or to the animation being defined. Most of the time the linking animation will be used as a loop, but sometimes it may be necessary to link to another animation, which this method allows.

Defining Explosions

An explosion plays any number of animations with various different attributes. Firstly, the animation can be made to start at a number of cycles after the explosion starts. When the explosion is played, the counter will start and then play the animations at each game cycle as it is reached. It is possible to have an explosion that does nothing for a couple of thousand cycles, and then plays an animation, or a number of them.

Any animation defined can take place at any number of hotspots in a group of hotspots. As described in the section on making sprites, this function can make the animations for explosions/smoke clouds appear at random groupings on another sprite. The animation will accept two numbers for this purpose, an upper and lower hotspot for the random grouping. The engine uses the upper and lower hotspots, and all the integers in between, and randomly picks one of these hotspots which it then uses to attach the animation. The hotspot for the sprite used in the animation is matched to the random hotspot which has been chosen for the explosion when the animation is played.

Each animation in an explosion can also have a translucency type. The values of these translucency types are defined (in shades.dat) when a tileset is made. The values are explained in the Making Tilesets sections earlier in this manual. The translucency type can be either one of these types, or ‘opaque’. If the animation is opaque, then all pixels just appear over the top of anything underneath the animation. Here is a sample explosion definition;

```
DefineExplosionType(sampanim_explosion)
{
  PlayAnimation(0 3 6 sampanim_animation opaque)
  PlayAnimation(56 0 2 sampanim2_animation translucentA0)
}
```

When this explosion is played, the animation sampanim_animation will begin playing instantly, on any of the hotspots 3 to 6 inclusive. After 56 game cycles, the animation sampanim2_animation will begin playing on any of the hotspots 0 to 2 inclusive, with a translucency type of translucentA0.

Build.txt

Build.txt contains all the information about buildings in the game. The file is a list of buildings one after the other. The order of the buildings in the file determines the order in which the buildings show up in the build menu in the game for the different sides. It is a good idea to separate the sides in the game so that they appear in distinctly different parts of the text file; because they will never appear in the list together, the order of the sides in the text file does not matter.

Buildings are comprised of a sprite, which conforms to the sprites from the “Making Building Sprites” section described earlier, and many other attributes. Some of these attributes are common to all buildings and some are specialised to give each building its individual function within the game.

I’ll use multiple sample buildings to demonstrate how the file works, and its capabilities and disadvantages.

Firstly, the simplest building possible with no function:

```
DefineBuildingType(building_identifier)
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
{
 SetBuildingImages(bottom.spr top.spr isoview.spr)
 SetDescription(Building_Name_To_Appear_In_Game)
 SetRequirements {
  SetType(unique_identifying_number_for_this_unit)
  SetPrereqs(identification_numbers_for units_that_are_needed_as_pre-requisites)
  SetMaker(makernumber1 makernumber2)
  SetEquivalence()
 }
 SetSide(side_number)
 SetRepairCost(hitpoints_per_repair_cycle   cost_of_repair_increment)
 SetHitpoints(hitpoints padding)
 SetSeeingRange(radius_in_tiles_of_seeing_range)
 SetSeeingHeight(height_in_tiles_that_sight_should_be_taken_from)
 SetCost(cost_in_credits time_to_build)
 SetSell(price_in_credits time_to_sell)
 SetVulnerability(armour_type percent_of_damage_to_send_through)
}
```

The building is first defined with its unique named identifier. This identifier is used in all other textfile references where a textfile needs to refer to a specific unit type. The identifier should therefore be descriptive so that it will be easier to remember when searching through the text files later.

The rest of the building is then defined.

SetBuildingImages()

The building images are split into two parts (as referred to in the Making Building Sprites Section), the top section is the picture part of the building units walks under, and the bottom sprite is the actual part of the building units walk under. Flying units do not adhere to this convention however, as they appear above everything except projectiles. The first sprite of the three is the building image that the units appear above, the second sprite is the image that the units appear below. An example of the use of this feature is a canopy or similar structure that the unit walks out from when created, and then walks onto an open part of the building, and then down below it. The third sprite in the *SetBuildingImages()* line is the isometric view of the building, which will appear in the build menu.

SetDescription

The SetDescription line specifies the name of the building in the game. This string is first matched against the strings in the multi-language support text file, and once the match is found, the new string is used in the game.

SetRequirements

The SetRequirements section defines the identification number used elsewhere in the text files. Each unit and building needs a unique number, anywhere in the range of one to 100,000. It is a good idea to group the units and buildings into subsets of this range, so that they can be easily identified later on. This section also defines the pre-requisites for the unit. Pre-requisites work on the premise that the player must have at least one of each of the pre-requisites to be able to be constructed. After the unit or building is constructed, it no longer relies on these pre-requisites to exist. However, if the player wants to build another of these buildings, they must again have all the pre-requisites that are listed in the definition. The pre-requisites for a building or a unit can be both units and buildings, and can number anywhere from none to as many as needed.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SetMaker()
This section also describes the 'maker' of the unit or building, which is the identification number of the unit or building which actually creates it. In the case of a building, its maker needs to always be a unit, and in the case of a unit, the maker must always be a building. A unit can have multiple makers, which is a necessity for the method used to upgrade buildings. When a building upgrades from one building to another, it also changes identification numbers. So, for example, a unit that can be built from a certain factory, will list that factory's identification number as its maker. However, if the factory is upgraded, because the first level factory no longer exists, the first level units can no longer be made. To allow first level units to be produced from a second level factory, the units must have the identification numbers listed in the SetMaker() field of all the buildings needed to produce those units.

SetEquivalence
SetEquivalence is used to specify units and buildings on different sides to be equivalent in the game engine. When players steal plans from other players, and thereby acquires the knowledge of how to build a building/unit, this does not necessarily mean they have the technology necessary to build the building/unit. The stolen building/unit will have its own set of pre-requisites for its own side.

For the user to build a unit/building for which they have stolen plans, they must already have the pre-requisites for the stolen unit or building. The way that the Tactics Engine allows this facility, is to have an equivalent unit to the one that has had its plans stolen. For example, if a player was to steal the plans for the Baloney Tank, but the Baloney Tank required prerequisite buildings that the player could not build (because they were from an opponent's side) then the player would not be able to build the Baloney Tank. But, because the buildings required to make the Baloney Tank have equivalents on the player's side, as long as the player has these equivalent buildings, then he or she can build the Baloney Tank.
This is the reason for the inclusion of Equivalencies in the Tactics Engine. To make two buildings Equivalent, you, the designer picks a unique equivalency ID, and then the buildings which are to be equivalent should have the same ID in their equivalency fields. The equivalency ID can be any number between 1 and 65000, and need only be unique among itself, irrelevant of what numbers are used for unit identification.

SetSide()
The SetSide() line allows the designer to specify the side which the building will be on. If the side number is set to (-1), then the building will be available to all sides (reliant on pre-requisites, and makers). The scenario editor uses the side values to differentiate between sides when allowing the user to place buildings and units, and there are other places in the engine where the side number is used, mostly in interface issues. The first side should be 0, and then the sides should be numbered incrementally from 0. One thing to remember is that the Tactics Engine interface is only designed with two visible sides in mind. It is possible to have buildings and units that are not specific to either of these sides by making the side number of the building or unit to be two. This will allow the units or building to be placed in the scenario editor as the third side.

SetRepairCost()
SetRepairCost() specifies a building's ability to repair, and the rate at which it repairs. The first parameter in this line is the number of hitpoints per repair cycle the buildings should repair. The second parameter is the cost in credits to repair this increment. Money is taken straight from the player's account when a building begins repair. If money runs out before a building is fully repaired, the building retains the repair order, and when money becomes available again, the building will continue repairing itself.

SetSeeingRange()
SetSeeingRange() is the range radius in tiles that the building is able to see. The actual amount of the map that the building can see is affected by overlays and altitude changes in the ground.
Because buildings can have height above the ground which may differ from building to building, it is possible to define the height from which the building "sees". In this way, a taller building could see easier into a valley on one of its sides, whereas a building with a lower seeing height would be unable to see into a steep valley, unless it was very close to the edge of the building. The SetSeeingHeight() line is used to specify the height in altitude marks above the altitude of the ground that the building is to see from.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SetCost()
The SetCost() line in the building definition defines the cost in credits and time that the building will take to build. The first parameter is the credit cost, and the second parameter is the time cost, which is measured by the number of game cycles.

SetSell()
The SetSell() line in a building's definition specifies the amount of money in credits that players get when they sell their building, and the time the building takes to sell. When selling upgraded buildings, the player is credited with the total sell amounts of all the buildings in the upgrade path. The 'time to sell' is based on the time it takes to sell the last building in the path.

SetVulnerability()
The SetVulnerability line sets the armour class of the building, as well as a percentage damage. The armour classes are relevant to damage.txt, which is discussed in detail in a later section. The percentage damage to let through is used mainly for play balancing purposes. This value is redundant with flexibility allowed in damage.txt, however, it is the easiest way to strengthen or weaken a building's armour at will. A value of 100 for this parameter means that 100 percent of any attack on this building will be sent through to the damage tables to be further calculated into a hitpoint value the building loses. A value of 200 means that any attack on the building will be twice as strong, while a value of 50 means that any attack on the building will be half as strong. A value of 0 means that the building will be indestructible.

Buildings that Rely on Efficiency
First, a simple guard tower.

The Tactics Engine's ability to allow a unit part to be attached to a building part, means a building can be made with artillery gun part to shoot at enemy units. A building with this attribute should have no other special functions, because of the interface with the constructed building. The engine allows the building to do other things, but because of the structure of the interface, a building with an attached unit part faces some restrictions. Although other functions can be added to such a building, these functions are not recommended or supported as there are interface issues that arise when using this type of building with other special functions.

```
DefineBuildingType(blahgtower)
{
 SetBuildingImages(blahg0.spr blahg1.spr blahgiso.spr)
 SetDescription(Blah_Guard_Tower)
 SetRequirements {
  SetType(12874)
  SetPrereqs(1004)
  SetMaker(7234)
  SetEquivalence(9001)
 }
 SetEfficiencyResource{
  (10 0)
  (100 90)
 }
 NeedResource(2 50)
 SetSide(0)
 SetRepairCost(100 2)
 SetHitpoints(600 1)
 SetSeeingRange(8)
 SetSeeingHeight(1)
 SetCost(200 15)
 SetSell(100 7)
 ActivePart(Blahgtowergun 30 36 none 0 0 none 0 0 none 0 0)
 SetTransportUnit(BlahPackingThing 15 30)
 SetVulnerability(BuildingArmour 100)
}
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

There are a number of new concepts that should be introduced at this point. The first is the concept of efficiency based on resource availability. There are a set number of functions for buildings that rely on efficiency.

NeedResource()
The NeedResource() line defines both the number of the resource required, and the amount of the resource which must be allocated to the building in order for it to be fully functional. The first parameter for this line is the resource number and the second is the amount of the resource needed. The units for resource amounts are not named, but the units are constant among all places where the amount of a resource is used in a definition.

The efficiency of a building is defined within the Tactics Engine as the ability to perform the role or function of a building, depending on resources available. In most cases, this will apply by making a building less efficient, depending on how much resources it has. The classic implementation of this concept is the use of power. When a base runs low on power, this means that several key buildings that are affected by the power output of the base will lose efficiency, making their functions much slower to complete. For example, a factory can be linked to the power resource, and when the power drops below a certain level, the efficiency of the building will drop, making the building produce units slower than normal.

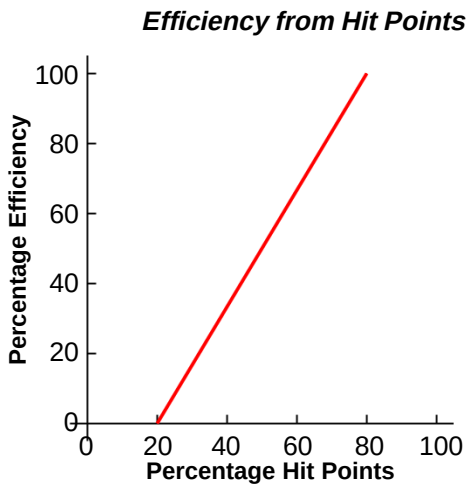
How is it Calculated?
The efficiency of buildings is calculated from a combination of the efficiency due to hit points and the efficiency due to available resources.

Efficiency from Hit Points
(Not operable in current build but can be enabled in successive versions)

The following definition denotes how the building's current percentage of hit points effects the building's efficiency.

```
SetEfficiencyHitPoints
{
  (0 20)
  (100 80)
}
```

The numbers are in fact a pair of coordinates, which would be graphed as follows.



The graph indicates that when this building has 80% hit points it is 100% efficient and when it has 20% hit points it has 0% efficiency. Also, when it is above 80% hit points it remains 100% efficient and likewise when below 20% hit points it is 0% efficient. This allows through configuration, the disabling of buildings when they have been sufficiently damaged. The graph also shows that the efficiency calculation is linearly interpolated between those two points, hence when the building is at 50% hit points it is 50% efficient.

EMAIL
HELPDESK@AURAN.COM

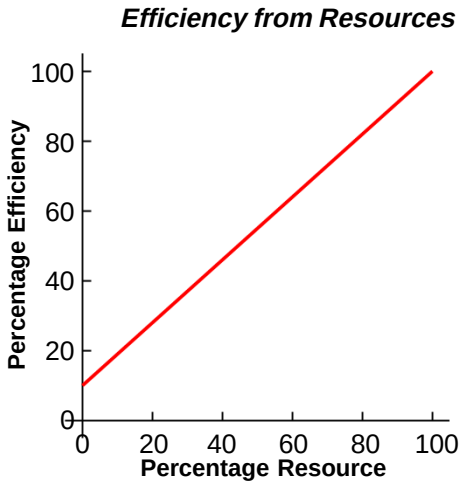
SUPPORT
SUPPORT@AURAN.COM

Efficiency from Resources

The following definition denotes how the buildings current level of resources effects the building's efficiency.

```
SetEfficiencyResource
{
  (10 0)
  (100 90)
}
```

Like the efficiency from hit points, these are points on a graph.



This is combined with the NeedResource definitions in a building type. For example the following definitions denote that this building type required 50 of resource 2 and 100 of resource 3.

```
NeedResource(2 50)
NeedResource(3 100)
```

To calculate the efficiency of this building the engine looks to see what the current supply of the resources 2 and 3 are… If this building has 40 of resource 2 and 70 of resource 3 then the following calculation is made to find the percentage of resources available.

From resource 2 we find this building has 40/50 * 100 = 80%.
From resource 3 we find this building has 70/100 * 100 = 70%.
These percentages are combined 80% * 70% = 56%.

Hence this building has 56% of its required resources. This is then used on the graph to find that the building is running at 54.8% efficiency, although the 0.8 will be lost resulting in 54% efficiency.

What does it affect?

Efficiency affects the following building functions.

The speed at which buildings construct units. A building that is 50% efficient, takes twice as long to produce a unit compared to 100% efficient building.

The speed at which buildings repair/heal/re-arm units. A building which is 50% efficient, take twice as long to repair/heal/re-arm units, compared to a 100% efficient building.

The amount of resources collected from the ground is affected by the efficiency of the building, which is collecting them. A building that is 50% efficient, collects resources at half the rate compared to a building which is 100% efficient.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The firing rate of units which are parts of buildings is reduced when the efficiency of the building its on is reduced. If the efficiency of a building is 50% then the firing rate of the unit is reduced by half.

The time a units, which are on buildings, weapons take to recharge increases as the efficiency of the building decreases. When the building is 50% efficient, the unit's weapons take twice as long to recharge compared to a building that is 100% efficient.

The other new concept is the *ActivePart.* The way a guard tower works using the Tactics Engine is that the base of a guard tower is defined as a building, and the top of the guard tower, or the gun part, is defined as a unit. To put them together, the gun part is defined at a place on the base using a pixel reference. A building may have up to four active parts. The format for the Activepart line is as follows:

SetActivePart(unit_to_attach xpixeloffset ypixeloffset unit_to_attach xpixeloffset ypixeloffset unit_to_attach xpixeloffset ypixeloffset unit_to_attach xpixeloffset ypixeloffset). If there are less than four active parts being attached to the building, then the unit_to_attach should be "none" for the unused slots, and the x and y pixel offsets should be 0. In the example above, for the guard tower, there is a single active part attached, called the "Blahtowergun", which is attached to the base 30 pixels from the left edge of the base sprite, and 36 pixels from the top edge of the base sprite.

This sort of configuration creates a building which appears in the build list, but which has the unit attached to the top as specified by the ActivePart line. The properties of this building allow the unit and the building to behave as a single entity, a unit that cannot move. The building, when selected, presents the attack cursor over enemy forces. It behaves in every way like a unit with a turret that cannot move. However, the health of the guard tower is taken from its bottom part, as is its armour. Health and armour values in the attached unit's definition are ignored, but some values must be present.

There is another line which is not specific to guard towers, and can be used in the definition of any building. The *SetTransportUnit()* line allows the specified building to be "packed" into a mobile unit, which can be moved around the board to a destination, and then unpacked. Packing up a building does not cost the user any credits. The unit the building is packed into spawns when the building is packed, and the packing and unpacking time can be defined. The first parameter is the unit that the building is to be packed into, the second parameter is the time taken for the building to pack into the unit, and the third parameter is the time taken for the building to unpack from the unit. So, in the sample guard tower above, the building packs up into the unit defined as the "BlahPackingThing" unit in 15 time cycles, and takes 30 time cycles to unpack.

The next building that we will look at is a demonstration factory building. This definition will take you through the simplest case factory.

```
; Blah Factory that makes first level units.

DefineBuildingType(factory)
{
 SetBuildingImages(nfvcy1l0.spr tfvcy1l0.spr bfvcymn0.spr)
 SetDescription(FG_Assembly_Plant)
 SetRequirements {
   SetType(100)
   SetPrereqs(10)
   SetMaker(2)
   SetEquivalence(1100)
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
 }
 SetEfficiencyResource {
  (10 0)
  (100 90)
 }
 SetSpyType(UNITPLAN)
 NeedResource(2 100)
 SetSide(0)
;  SetEquivalentBuilding(ic1)
 SetBay(2 2)
 SetRepairCost(10 10)
 SetHitpoints(1000 1)
 SetSeeingRange(5)
 SetSeeingHeight(1)
 SetCost(2200 66)
 SetSell(1100 33)
 CanMake()
 MakesCrater(2)
 SetTransportUnit(FGGroundTransporter 15 30)
 SetVulnerability(BuildingArmour 100)
 SetHealthExplosion(75 building_stage1_explosion)
 SetHealthExplosion(50 building_stage2_explosion)
 SetHealthExplosion(25 building_stage3_explosion)
 SetHealthExplosion(0  building_death_explosion)
}
```

A factory produces units which appear on its bay and, by default, will then move to a spot about three spaces below the lower extremity of the building. For this reason, the building should have the exit somewhere near its base, and should have the bay around the middle of the building.

Units simply “appear” on top of the bay, so the trick is to get the building to have its bay somewhere where the units are masked from the view of the player when they appear (by the top level of the building), and then the units will attempt to move to a spot below the building. There should be a path (by the use of the overlay effects file explained later) allowing the units to travel to this spot. Obviously, it would be ideal if the opening of the building was positioned in such a way as to have this path leading outside the building and ending up pointing downwards, either at an angle or directly down. This will ensure the units do not have to move far to exit the building.

CanMake()
The CanMake() flag specifies that the building can produce units. The engine relies on this to determine whether the building’s unit producing capabilities. Therefore, the *CanMake()* line needs to appear somewhere in a factory’s definition.

Buildings which can produce units are dependent on their set efficiency values. The efficiency of a factory determines just how fast it can produce units.

Upgrading Buildings
The Tactics Engine allows buildings to be upgraded based on prerequisites. The *IsUpGradeOf(building_identifier)* gives the player the opportunity to upgrade the building listed in this line to the building which contains the line in its definition. It will only allow this to occur when prerequisites are satisfied (that is, the player owns a unit/building of each prerequisite ID, as listed in *SetPrereqs(prereqid prereqid).*

Building Giving Minimap
A building with the *GivesMiniMap()* line in its definition allows the mini-map to be seen in-game. The mini-map goes to a powered-down state when this building is anything less than 100% efficient.

Repair Action Indicator
The *SetRepairActionIndicator(overlay_identifier)* line in a building’s definition defines the visual indicator to be used for this building when it is being repaired.

Decoy Buildings
Decoy buildings behave something like the Decoy feature for units. The Decoy for a building must be defined after the real building. The first line of the Decoy's definition should be *IsDecoyOf(building_identifier).* The decoy building should not have a *SetBuildingImages()* line, but can have different prerequisites set to the original building. The player who owns the decoy building will see the description as defined in the decoy's definition, but other players will see the building's definition as that of the original building.

Buildings that should not consume a Construction Crew
If it is necessary for a unit to build a building and not be consumed in the process of the construction of the building, then the *IsNotBuilderEater()* flag should be included in the definition of the building. This line takes no parameters.

Craters
When a building is destroyed, an overlay can be left behind, to denote the building/'s previous existence on that turf. The *MakesCrater(number)* line takes a parameter which corresponds to the matching overlay with the *IsCrater(number)* line in its definition.

Building which can be Spied On
Any building can be spied on in one of the following ways:

Resource: if an enemy/neutral/friendly spy infiltrates a building with the line *SetSpyType(RESOURCE)* in its definition, the infiltrating player is able to see the players usage of resource type 0, and resource type 2.

Unit plans: if an enemy/neutral/friendly spy infiltrates a buildings with the line *SetSpyType(UNITPLAN)* in its definition, the enemy player is able to steal the plans for any unit for which the player has plans, irrespective of whether or not he or she has built it, or have its prerequisites.

HQ: if an enemy/neutral/friendly spy infiltrates a building with the line *SetSpyType(HQ)* in its definition, the infiltrating player gets the line of sight of the infiltrated player, and can also steal the plans for any building the player has plans for, irrespective of whether he or she has built it, or have the prerequisites for it.

Buildings Healing Units
A building with the *CanHeal()* flag in its definition is able to heal units on its bay.

Buildings Repairing Units
A building with the CanRepair() flag in its definition is able to heal units on its bay.

SetResourceSale()
A building with the SetResourceSale() line in its definition can play through a section whenever a resource sale occurs. The SetResourceSaleAnimation(section_number) line defines that the section section_number will be played at this point.

SetRepairAnimation()
A repair animation occurs when the building with the line SetRepairAnimation(section_number) repairs a unit.

SetBoardAnimation()
The boarding animation is played by a building with the SetBoardAnimation(section_number) in it's definition. This animation is played when a unit enters a building with rooms.

SetRearmAnimation()
A building that can rearm units can play through a section in it's sprite when it rearms, using the line SetRearmAnimation(section_number) in its definition.

Buildings transporting Units
There are two ways buildings can transport units. The first is by way of the underground transporter. A building which will transport by these means needs to have three flags. The *SetRooms()* flag and

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

the *AssociatedUnit()* flag. *The SetRooms(max_number_of_units max_weight_per_unit)* defines that the building can hold max_number_of_units of units, each unit can weigh a maximum of max_weight_per_unit. The *AssociatedUnit(unit_identifier 1)* line defines the unit which will be transporting the other units. The unit associated with this building should have bays which are equal in number and size to those defined in the SetRooms() line in the unit's definition. The first parameter in *the AssociatedUnit* line defines whether the unit will die when the building dies. If the flag is set to 1, the unit will die when the building dies. If it is set to 0, it will not die when the building dies.

The other way a building can transport units is via the teleportation travel mode. To teleport, a building should have the *SetRooms()* line, with the necessary number of bays and maximum weight per unit in the bays. The *IsTeleport(charge_time explosion_identifier)* sets the charge time on the teleport device, and an explosion to play over the centre of all units transported upon arrival at their destination. The player can teleport units into any previously seen area of the map, whether they have line of sight there or not.

Building Idle Animations
A building can be defined to play through a section constantly when not performing another function. The *SetIdleAnimation(section_number)* function is used to facilitate this.

Damage.txt
Damage.txt takes care of the amount of damage the various attack types do to the types of defined armour. If a unit uses an armour class not defined in this file, then the game will not run. This file allows configuration of the percentage damage each offence class does to each armour class. Basically, any attack on a unit or building will have a hitpoint attack value, and an offence class. The Tactics Engine takes this hitpoint attack value and then looks at the relevant armour type in damage.txt. Within the armour type, the engine looks at the percentage damage that an attack with this offence class does to the armour type. The hitpoint value is then multiplied by the percentage, and this number of hitpoints is deducted from the unit.

Attacks can come from both weapons, defined in weapon.txt, and terrain, defined in trneff.txt.

The premise behind this concept is to let units with similar weapon types have the same offence class, but which might have different attack values to differentiate them.

As well as the percentage damage the attack will to a unit/building, there are a number of other attributes that can be modified. These are a modifier to the percentage, as well as a critical hit percentage. The modifier is used to change the percentage of the attack transmitted to the unit/building. For example, if the modifier was 5, and the unmodified percentage was 90, then the engine will pick a percentage damage between 95 and 85% randomly. Furthermore, if the hitpoint attack is 200, then the final damage to the unit would be somewhere between 190 and 170.

The critical hit percentage determines how likely it will be that the attack on the unit/building will be a critical hit, inducing immediate death/destruction. If this variable is set to 5, for example, there is a 5% chance that any attack coming through this offence class will critically injure and destroy the target that bears this armour type.

At the beginning of the damage.txt file, there is a list of all offence classes which will used in the various armour types later in the file. This must include all offences which are referenced in all the other text files. An offence class name must take the form of a letter-single-digit-number. The letter must be upper case. Examples are A1 B6 O1 O2 and so on. You can use whatever method for grouping the classes you wish. Here is an example damage.txt file:

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

If a defined offence class has no entry in an armour type's vulnerability listing, then the attack will use defaults that are read from the second, third and fourth parameters of the vulnerability definition.

```
DeclareOffenses()
{
  A1 B1 B2 C1
}

DefineVulnerability(HumanArmour 100 10 0)
{
  A1 50    10   0
  B1 100    0   0
  C1   0    0   3
}
```

So, for example, let's look at an attack on a unit/building with armour type HumanArmour, each time for an attack value of 50:

If the offence class is A1, then the damage to the unit will be between 20 and 30 hitpoints.

If the offence class is B1, then the damage to the unit will be 50 hitpoints.

If the offence class is C1, then the damage to the unit will be nothing, however, there is a 3% chance that the unit will be destroyed immediately upon being hit by this attack.

Care should be taken with the modifying percentage to ensure it does not become possible to do negative damage to a unit or building. An attack like this will in fact add hitpoints to a unit. Units which repair/heal other units are special cases of this, but are treated differently, as explained later in the UNITS.TXT and WEAPON.TXT sections.

Overlay.txt

Overlay.txt is responsible for all the definitions for the overlays used on maps based on the Tactics Engine. Overlays are the sprites which appear as scenery on a map such as rocks, trees, burnt out wrecks of units and so on. In this file, overlays are given general attributes like their image, and their shadow image (which are in the form of an animation), and attributes to define whether units move over them, or under them. This file also defines hitpoints for overlays if they can be destroyed. The flag that determines which overlays can be morphed into if a unit has that ability is also set within this file.

The overlay system is also used for some building actions, such as repairing. So that these overlays do not show up in the Scenario Editor when placing overlays, there is a flag that excludes the defined overlay from that list.

All lines present in the rock_basic overlay must be included in all overlay definitions. Everything else is optional.

```
DefineThingType(rock_basic)
{
 SetThingImage(rock_basic_animation)
 SetLevel(2)
}

DefineThingType(rock)
{
 SetThingImage(rock_animation)
  SetThingShadowImage(rock_shadow_animation)
 SetLevel(2)
 CanBeMorphedInto()
  NoEdit()
  SetHitPoints(100 0)
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
 SetHealthExplosion(75 stage1_explosion)
 SetHealthExplosion(35 stage2_explosion)
 SetHealthExplosion(0 stage3_explosion)
}
```

The *SetThingImage(animation_id)* line sets the appearance of the overlay, and specifies an animation which should be looping, unless it is specifically required that the Overlay eventually change into another animation (if it is so linked in ANIMATE.TXT), or disappear. If the animation runs out, the overlay is not destroyed, and therefore neither are its effects. The overlay's effects only disappear if the overlay is destroyed.

The Shadow image of the animation is defined in the *SetThingShadowImage(animation_id)* line. Obviously, it should be realised at all times that the shadow image of the overlay should link to the overlay at all times.

The *SetLevel(level_id)* line is used for display sorting the overlay when the overlay is on-screen. This is used to determine whether a unit moving across an overlay appears below or above it. For example, a unit should move on top of a group of loose rubble, but would appear beneath a tree. Listed below are the levels of other sprites used by the Tactics Engine:

0 The terrain surface is at level 0.
1 The base layer of buildings, Overlay and Unit shadows
2 Units with movement mode Hover, and movement mode Ground
3 The top layer of buildings, and building shadows.
4 Units with movement mode Fixed (guard tower turrets)
5 Flying Units
6 Missiles and explosions

So, if a unit is to appear above the overlay in question then it should be placed at level 0. If units are to appear below the overlay (except flying units), then the overlay should be *SetLevel(2).*

Overlays with the *CanBeMorphedInto()* flag can be morphed into by units with the *CanMorphIntoOverlay()* flag.

Overlays with the *NoEdit()* flag will not show up for placement in the Scenario Editor.

Overlays can be defined to have a set number of hitpoints. Overlays that do not have hitpoints defined are indestructible. The first parameter in *the SetHitPoints(hitpoints padding)* is the number of hitpoints the overlay has. The second parameter is used as padding (read: unused). Attacks on an overlay do not go through the damage table, and therefore 100% of any attack on an overlay is deducted from its hitpoint amount.

```
 SetHealthExplosion(75 stage1_explosion)
 SetHealthExplosion(35 stage2_explosion)
 SetHealthExplosion(0 stage3_explosion)
```

Health explosions define explosions that occur when a unit reaches a certain hitpoint percentage. The first parameter is a hit point percentage for the explosion, and the second is the name of the explosion to be played. The first health explosion in this group is triggered when the unit's health drops below 75%. The explosion imhit_explosion is played. If this explosion includes a looping animation, those animations will keep playing. When the unit reaches the next boundary, all animations from the previous hit explosion are stopped. For example, if the hit explosion at 75% contained an animation that looped continually, then when the unit drops below 35% health, the animation from the hit explosion for 75% that is looping will be stopped, and the explosion imdying_explosion will be played. Again, if this explosion contains an animation that is looping, it will be cancelled at the point the unit is killed (0% hitpoints). However, including an animation that loops forever as the lowest percentile health explosion in the part definition will mean that the animation will never be stopped, and will continue until game end. Therefore, animations that occur in the health explosion at the lowest percentile should finish at some point. Animations occur on their respective hotspots as defined in the *DefineExplosion* section of animate.txt (see the section

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

on animate.txt).

As decried in the Build.txt section, an overlay can be defined to be a crater that is left behind when a building is destroyed:

```
DefineThingType(rock_basic)
{
 SetThingImage(rock_basic_animation)
 SetLevel(2)
 IsCrater(5)
}
```

The *IsCrater()* line defines that this overlay is associated, upon destruction, with the building that has the *MakesCrater(5)* line in its definition.

There is a special case where an overlay is used as the visual cue that building repairing is in progress. The overlay in this case should be a simple overlay, and should also include the *NoEdit()* flag so that it cannot be placed in the map editor. It need not have any effects defined in ovleff.txt.

Ovleff.txt

The OVLEFF.TXT file defines how buildings and overlays in the game affect unit movement across them, and whether certain tiles are within a building's effects or not. Each tile that is defined to be part of an overlay can have a different effect. This is the way the Tactics Engine makes it possible for building/overlay sprites to overlap other tiles, without actually affecting them. Each sprite is determined to be a rectangular area, with a defined height and width.

However, it is not necessary to make sure that the sprites fill the whole rectangle. It is always a good idea to attempt to make overlays fill a certain tile grid, in the sense that any tile inside the rectangular area should be able to be defined as either part of the overlay, or not part of it. In some cases having the transversal from sprite to terrain in the middle of a tile rather than a boundary may look visually incorrect.

Each tile of the overlay is also allowed to have a different height. This value is added to the terrain height, and is allowed to be negative. Its purpose lies in the line of sight of units and buildings. Trees and buildings are allowed to block line of sight in this way. There is a maximum and minimum of this value which are 2 and -2. Values outside this range will cause difficulties for the engine, so it is best not to use them.

A sprite's effects are defined by the sprite name of the overlay or building. In a building's case, because it is composed of two sprites, only the bottom one needs defined effects. All overlays defined for use as game overlays need effects, however overlays specially used by the game for building animations do not. For overlays, remember that the effects are defined for a sprite file, not for an animation, so the overlays' effects should be defined with respect to the sprite used for the animation of an overlay.

There are four coded terrain effects which should be set up in the terrain effects text file trneff.txt.

(-1) This effect means that the overlay has no effect on this tile. It should be used for tiles on the sprite's edges that are within the defined rectangle.

(2) This is used for the tile that is used as a building bay in the case of a unit creation building, and any tile that the unit must travel on during its exit from the building.

(3) This effect is used to denote that units cannot travel onto this tile. It should be used for all tiles of overlays onto which units cannot move. Flying units *can* pass over tiles with this effect.

(5) This effect is assigned by the code and should not be assigned to any tile in an overlay / building's overlay effects definition. This effect is given (at runtime) to the bay of a building that produces units.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The tile structure is set out in a grid array, each tile has a pair of numbers pertaining to its effect and its altitude effect.

Here is an example of an overlay effect definition.

```
DefineOvlEffect(blah.spr 3 4) {
-1 0  -1 0  -1 0
 3 0   3 1   3 0
 3 0   2 0   3 0
 3 0   3 0   2 0
}
```

This is a building sprite that is three tiles wide, and four tiles high. The top row of tiles of this building has no effect on the terrain, and allows units to pass freely over these tiles. If the overlay visually overlapped into this top row of tiles then any units moving onto these tiles would either appear above or below this overlay. This characteristic is determined by whether or not the centre of the unit sprite is higher or lower than hotspot 0 on the overlay. If the centre of the unit is above the hotspot, then the unit will appear below the overlay. If the centre of the unit is below the hotspot, then the unit will appear above the overlay. Therefore if the hotspot is at the very top of the overlay sprite, then the unit will appear to be over the overlay at all points. Conversely, if the hotspot were to be at the bottom of the overlay, then the unit will appear to be under the overlay at all points.

The tiles in the second row all have overlay effect 3, but the middle tile also has an altitude effect of 1, which makes it one altitude unit higher than all other tiles for this overlay. This will block some ground units from seeing past this particular tile in the overlay.

The bay for this building is in the middle on the third row, and is set to allow units to be created on it. It is possible for a unit to be created on a tile that has effect 2. The unit is forced to exit through the bottom right of the building after being created.

Note: Buildings that are upgrades must *always* have the same effects as that of the base building. This is extremely necessary, and will result in problems if not adhered to.

Trneff.txt

This file is responsible for the speeds of the various units on the different terrain numbers. Within this file, it is also possible to set certain terrain numbers to be able to damage units. Any terrain number can be specified to do a constant attack with an offence class and hitpoint value on units that occupy a tile with this terrain value.

Units use movement effects that are listed in this file. Each movement effect has a defined speed percentage over each of the 16 different tile types, and also over the eight possible special tile variations. The maximum altitude difference that a unit with this movement effect can traverse is also definable.The different movement effects are listed one after another in this file. An example movement effect line follows:

```
DefineEffectType(Tracks 100 2)
{
  SetEffect(0 -1 0 6)
  SetEffect(1 -1 100 6)
  SetEffect(2 -1 100 6)
  SetEffect(3 -1 100 6)
  SetEffect(4 -1 25 6)
  SetEffect(5 -1 25 6)
  SetEffect(6 -1 25 6)
  SetEffect(7 -1 25 6)
  SetEffect(8 -1 200 6)
  SetEffect(9 -1 200 6)
  SetEffect(10 -1 100 6)
  SetEffect(11 -1 100 6)
  SetEffect(12 -1 100 6)
```

EMAIL
HELPDESK@AURAN.COM
SUPPORT
SUPPORT@AURAN.COM

```
  SetEffect(13 -1 100 6)
  SetEffect(14 -1 100 6)
  SetEffect(15 -1 100 6)

; special effects

  SetEffect(16 0 0 0)
  SetEffect(16 1 0 2)
  SetEffect(16 2 100 2)
  SetEffect(16 3 0 0)
  SetEffect(16 4 0 0)
  SetEffect(16 5 100 8)
  SetEffect(16 6 50 2)
  SetEffect(16 7 0 0)
}
```

The special tile types are the ones referenced in ovleff.txt. You can define and use any of the special effects in your overlays/buildings which are not specifically used in the code as described in the section outlining ovleff.txt. This means that if, for example, you wanted an effect that units could walk through, and yet tanks could not (on a building or overlay), you can set special effect 5 to movement speed 0 for any movement type used by units that you want to restrict, and > 0 for any movement types used by units for which you want to allow passage.

Units.txt

Units.txt is likely to be the largest text file that you will need to work on for the configuration of the Tactics Engine. This file holds the structures for all the units in the game. The order of the units in units.txt determines their order in the unit list for the player.

As mentioned earlier in the document, units are defined as a number of parts. We will first use an example unit definition that has one part - a simple foot soldier with a gun:

```
DefineUnitType(FootUnit)
{
 SetDescription(FootUnitThing)
 SetMenuImage(ftunitmn.spr)
 SetSelectSounds(yessir.wav haha.wav kilforfn.wav done.wav)
 SetResponseSounds(blah.wav hoopla.wav yippee.wav)
 SetSide(0)
 SetCost(300 9)
 UseEffects(Foot)
 SetMoveMode(Ground)
 SetStrength(100)
 SetPhysics(1 6)
 SetHitSize(5)
 SetSeeingRange(8)
 SetVulnerability(LeatherArmour 100)
 SetShadowImage(footshad.spr)
 IsHuman()
 SetIdleAnimation(50 100 2 2)
 SetRequirements {
   SetType(9)
   SetPrereqs(290)
   SetMaker(602 540)
   SetTechLevel(0)
 }
 SetRundownInfo {
   SetAttack(0)
   SetDefence(20)
   SetSquishSound("squish.wav")
 }
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
 AddPart {
  SetRotationRate(30)
  SetRotationalArc(180)
  SetScanDelay(500)
  SetImage(footguy.spr)
  SetMoveAnimation(0)
  AddWeapon(RubberBandGun 1 0)
  SetStandingAnimation(3)
  SetHealthExplosion(0 imdead_explosion attached)
 }
}
```

DefineUnitType(identifier)
The first line, DefineUnitType(identifier*)* specifies a tag to which the unit is referred in all other text files used by the engine. This tag follows the same usage as the identifier for buildings in build.txt (mentioned in the BUILD.TXT section of this chapter).

SetDescription()
The SetDescription line specifies the unit's name, as sent to the MLString.cfg for translation. This procedure is mentioned in the BUILD.TXT section.

SetMenuImage(ftunitmn.spr)
SetMenuImage(ftunitmn.spr*)* specifies the sprite which will be displayed as the isoview image of the unit in the multi-function display (MFD) for the player to click on to begin building this unit.

SetSelectSounds()
The line SetSelectSounds sets the wave sounds which are played when a unit is selected. If a group of units is selected, only the primary unit in the group plays its select sound. One of the sounds is played randomly every time the unit is selected. There is a maximum of four sounds which can be defined for any unit as a select sound, however there may be any number of sounds less than this defined, including none.

SetResponseSounds()
SetResponseSounds*()* acts in a similar way as SetSelectSounds, but these sounds are played when a unit responds to an order, such as fire, or move.

SetSide()
The SetSide() line in the unit definition defines the side of the unit, or whether the unit is available to all sides. This definition works the same way for buildings. Setting the side to "-1" will mean that this unit will be available to all sides (whether a player builds it or not still depends on pre-requisites).

SetCost()
The SetCost*()* line in the unit definition outlines the cost in credits and time the unit will take to build. The first parameter is the credit cost, and the second parameter is the time cost, which is the number of game cycles, multiplied by a constant.

UseEffects()
The UseEffects*()* function in the unit defines which effects the unit uses for its travel over terrain. The effects used as parameters for this function are defined in the trneff.txt, as explained previously in this document.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SetMoveMode()
The SetMoveMode() function defines which type of movement mode the unit uses over the terrain. The values for this function are the following: ground, fixed, hover, flying, tunnel. Ground is the generic movement mode for all units that move along the ground. Units with legs, wheels, tracks and so on should use this movement mode. Fixed is only for the unit parts of guard towers, hover is for vehicles that should move in a way that makes them appear to be moving very smoothly over terrain, and flying is for units that should appear to fly over terrain. Note that these movement modes do not in any way define which terrain the unit can traverse, nor the speeds at which the unit will traverse them.

SetStrength()
The SetStrength() line defines the number of hitpoints of the unit. This number must be a positive integer. Of course, it is unwise to set this value to 0.

SetPhysics()
The SetPhysics line defines the weight and speed of the unit. The weight of the unit is taken into account for the transportation of units. When a unit is defined to be able to carry units, it is given a maximum weight per unit that it can carry. Therefore, units that are above this weight will not be able to board the transport, whereas units that are below this weight can. The speed of the unit is the base speed, from which variations based on the definition of its *UseEffects()* movement type change its speed.

SetHitsize()
Normally, a unit is treated as being one pixel in size. This means that unless the target unit is standing still, or the weapon fired has a broad area effect, then the target unit will be missed. The unit must, therefore, be given a physical size. The way this is done is to give it a radius in pixels that the unit inhabits. This is the function of the *SetHitsize()* line in a unit's definition. The parameter is the radius in pixels of the unit. Any strike by a weapon within this radius will damage the unit as if it were hit fully by the weapon.

SetSeeingRange()
The SetSeeingRange() line in the unit's definition defines a radius in tiles for which the unit can see, assuming that line of sight restrictions are not present. These restrictions work in much the same way as the restrictions on buildings work.

SetVulnerability()
The units' armour is defined with the SetVulnerability() line in the definition. This line sets which type of armour the unit will have, and what initial percentage of a strike on the unit will be calculated in the damage tables. This works identically to buildings receiving damage.

SetShadowImage()
The SetShadowImage() line is optional, and contains the name of a sprite which will be used as the shadow of the unit.

IsHuman()
The *IsHuman()* flag is used to designate the unit as a human unit. This flag is only used where the unit is a human, and is excluded at all other times. The IsHuman() flag is used by several special functions in the Tactics Engine.

CanOnlyShootHumans()
CanOnlyShootNonHumans()
Two of the major ones are the CanOnlyShootHumans() flag and the CanOnlyShootNonHumans() flag in the definition of a weapon in weapon.txt. These flags are explained later in this chapter in the Weapon.txt section. Another example of a special function that depends on the *IsHuman()* flag is the *CanGrab()* function, which facilitates conversion of units with the *IsHuman()* flag into a defined unit.

SetIdleAnimation()
SetIdleAnimation() allows the unit to have an animation which will be played after the unit has not performed any actions for a period of time. This time period is picked at random between two

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

numbers that are accepted as parameters by the function. The unit can also have several different sections that are played randomly. The format of this function is; SetIdleAnimation(min_time max_time min_section max_section). This will play a random section from the unit's sprite between min_section and max_section inclusive, after a time period in game cycles between min_time and max_time inclusive. All numbers used in this function need to be integers.

SetRequirements
The SetRequirements section of a unit's definition controls when and from which building a unit will be built. This section controls the prerequisites of the unit, in much the same way as the role of building prerequisites for buildings.

```
SetRequirements {
   SetType(9)
   SetPrereqs(290)
   SetMaker(602 540)
   SetTechLevel(0)
 }
```

SetType()
SetType(number) defines the prerequisite identification number for this unit. This number can be anywhere between 0 and 999 999 inclusive. This number must be unique across both unit and building prerequisite ID's.

SetPrereqs()
SetPrereqs() is a sequential listing of the identification numbers of buildings/units which are required as prerequisites to build the unit. The maximum number of prerequisites for any building is 8.

SetMaker()
SetMaker*()* is a sequential listing of the identification numbers of the buildings which can produce this unit. Identification numbers listed in the *SetMaker()* line should refer to buildings, and these buildings should always be capable of building the units (otherwise you'll never be able to build them anyway). It is valid for the *SetMaker()* line to not include any makers at all. For example, you might want to have a unit that a player could never build, but which could only be placed in the construction kit. There is a maximum of six makers that can be defined for any unit. The unit can be made from any of its makers, and only relies on one of them to exist to be built. However, at least one of them needs to exist to create the unit. Makers do not look down the upgrade hierarchy to see if the building from which they upgraded could make certain units. This means it is important that you list all building identification ID's in the upgrade heirarchy as makers of the unit if the upgraded buildings also need to build the units. The maximum number of makers is 8.

SetTechLevel()
SetTechLevel*()*. Any scenario may have a defined TechLevel from 1 to 100. TechLevels work on the premise that at any given TechLevel in a scenario, the player can build any units and buildings with a defined TechLevel lower than or equal to it. For example, if a scenario has a TechLevel of 17, the players on this scenario will be able to build units and buildings with the following TechLevel values: 14, 1, 0, 17. However, the player will never see the plans for units and buildings with the following TechLevel values: 18, 50, 28. The *SetTechLevel(number)* line defines the TechLevel for this unit.

SetRundownInfo()
The *SetRundownInfo()* section in the unit's definition is to facilitate the "running over" of smaller units by bigger units (well, at least it was when it was conceived, but you might find another use for it - maybe eating other units? :)). The rundown of another unit occurs when its defence value is lower than another unit's attack value. If the values are equal, no rundown will happen. When the rundown occurs, the sound effect defined in *SetSquishSound* is played. So, in this example unit, it cannot run any unit over (because it has an attack value of 0), and is rundown by any unit with an attack value greater than 20 (because its defence value is 20). When a unit is run over, the

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

squishsound, as defined by the *SetSquishSound("playme.wav")* line, of the unit that is running over the other unit is played.

```
SetRundownInfo {
   SetAttack(0)
   SetDefence(20)
   SetSquishSound("squish.wav")
  }
```

Addpart()
The Addpart() section of the unit defines several important characteristics of the unit that are part based, rather than the overall unit. Remember that the Tactics Engine supports units which have up to three parts. The first part in the definition is the bottom part of the unit, the second is the part of the unit on top of the bottom part, and then the third is the part on top of the part on top of the bottom part. The easy way to name the parts is bottom, middle, and top, but because both the middle and top part of the unit are not required, we have made the distinction.

```
 AddPart {
   SetRotationRate(30)
   SetRotationalArc(180)
   SetScanDelay(500)
   SetImage(footguy.spr)
   SetMoveAnimation(0)
   AddWeapon(RubberBandGun 1 0)
   SetStandingAnimation(3)
   SetHealthExplosion(75 imhit_explosion attached)
   SetHealthExplosion(35 imdying_explosion attached)
   SetHealthExplosion(0 imdead_explosion attached)
  }
```

SetRotationRate()
The *Rotation Rate* of the part is set in the *SetRotationRate(number)* of the Addpart Section. This rotation rate is specified as the turn rate of the part in degrees per unit cycle relative to the part below it. The rotation rate of the bottom part is the rotation rate of the unit relative to the ground. A Rotation Rate of 30 degrees per unit cycle is fairly high, and a unit with this rotation rate will turn around fairly quickly.

RotationalArc()
The *RotationalArc* of a part defines in degrees how far off straight ahead a part can move. This is primarily of use for a middle or top part of a unit, to make a turret have a narrow traversable arc. The rotational arc for a unit's bottom part should always be 180 degrees. If the Rotational Arc of a turret is set to 30, then the turret has a rotational arc that can be 30 degrees to the left and right of the frontal position. This in effect gives the turret a net traversal of 60 degrees.

SetImage(footguy.spr)
The SetImage(footguy.spr) line defines the sprite for this part of the unit.

SetMoveAnimation(section_number)
SetMoveAnimation(section_number*)* tells the engine which section of the sprite (defined in the *SetImage()* line) will be played when the unit is moving. Using the convention, this would likely be 0.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

AddWeapon(Weaponname section_number unit_cycles)
The *AddWeapon(Weaponname section_number unit_cycles)* line in a parts definition allows a weapon of type "Weaponname" to be associated with this part. Up to four weapons can be attached to any part. Usually the weapons will all acquire the same target (however, this relies on the weapons having the same range, and targetable characteristics – these will be explained in more detail in the weapon.txt section of this chapter). When the weapon is fired from a unit, section number section_number is played through, and after unit_cycles number of unit cycles has passed since the unit was given the "attack" order, the weapon called Weaponname will fire its projectile from hotspot number 3. Setting the unit_cycles to 0 will allow the weapon to fire as soon as the unit is given the attack order (always assuming that the target can be attacked by the firing unit, and that the target is within range). It may be preferred that the weapon fire a few unit cycles into the firing section that is playing, which is what this parameter is really for. The *AddWeapon()* line in a part definition is optional, and may be omitted as necessary.

SetStandingAnimation(section_number)
SetStandingAnimation(section_number) allows a point to be defined to where the unit will move and rest. Quite often, the unit will be moving along, playing its movement section continuously, but will then stop at a point in its movement section which may not be desirable visually (eg. standing on one leg). To stop this interfering with other actions the unit might perform (such as idle animations and firing animations), the unit is given a base point to return to after completing an order, which is the standing frame. Basically, this needs to be only as many rotations as the unit normally has, and just one additional frame. Giving a unit a standing frame is optional.

SetHealthExplosion()

Health explosions define explosions which occur when a unit reaches a certain hitpoint percentage. The first parameter is a hit point percentage for the explosion, and the second is the name of the explosion to be played. The first health explosion in this group is triggered when the unit's health drops below 75%. The explosion imhit_explosion is played. If this explosion includes a looping animation, those animations will keep playing.

SetHealthExplosion(75 imhit_explosion attached)
SetHealthExplosion(35 imdying_explosion attached)
SetHealthExplosion(0 imdead_explosion attached)

When the unit reaches the next boundary, all animations from the previous hit explosion are stopped. For example, if the hit explosion at 75% contained an animation that looped continually, then when the unit drops below 35% health, the animation from the hit explosion for 75% which is looping will be stopped, and the explosion imdying_explosion will be played.

Again, if this explosion contains an animation which is looping, it will be cancelled at the point the unit is killed (0% hitpoints). However, including an animation that loops forever as the lowest percentile health explosion in the part definition will mean that the animation will never be stopped, and will continue until game end.

Therefore, animations that occur in the health explosion at the lowest percentile should finish at some point. Animations occur on their respective hotspots as defined in the *DefineExplosion* section of animate.txt (see the section on animate.txt). If the health explosion is attached, the explosion will occur on the unit, and any animations playing will stay attached to the unit as the unit moves. If the explosion is unattached, then the explosion will occur at the position of the unit, but will stay fixed relative to the map. If the unit moves off the spot, any animations which are still playing as part of the explosion will stay fixed.

Now that the basic unit has been defined, there are a number of special flags and definitions that need to be explored.

Using a Unit as a Resource Transporter.
There are a several parameters a unit needs to have in order to transport resources. The most important one is the *SetResourceTransport()* line in the unit's definition.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SetResourceTransport(0 750 270 500)
The first parameter of this line is the resource identification number of the resource that will be transported by the unit. The second parameter is the maximum number of units of resource the unit can hold. The third parameter is the number of units of resource the unit grabs every time it loads. The fourth parameter is the number of units of resource the unit dumps every time it unloads.

The following two lines also control how fast the transport loads and unloads:
SetTransportLoadAnimation(90 5)
SetTransportUnLoadAnimation(0 6)

The parameters for both these functions are the same, the first is the angle to which the unit will rotate (in degrees) before commencing loading/unloading, and the second is the sprite section which is played while the transport is loading.

When this unit moves to a building that supplies a resource, it rotates to angle 90 (which is pointing straight up), and then plays through section 5. Once it has finished this animation, the transport fills up with 270 units of resource. It will then repeat this process until it gets the full 750 units of resource (that is, after the third time it plays section 5). The transport will then have finished loading. If the unit cannot get its full complement of resource straight away, it will continue to try to get the resource from the building, until it has the full complement of resource.

When this unit goes to a building which will receive the resource, the unit will rotate to angle 0 (which is pointing to the right), and will then play through section 6. After it has finished this, it will have offloaded 500 units of its load to the building. If it doesn't have 500 units, it will offload all the units it has.

The animations affect how quickly resource can be grabbed from a facility because the animation restricts how often the transport attempts to load. If there were no animations for the loading and unloading of the unit, it would grab 270 units of resource every unit cycle, and would therefore be full in three unit cycles. When the unit loads *with* the loading animation, it takes the unit three times longer to play through the loading animation.

Another element which should be established for units that carry a resource is the *automatic resource pathing*. The units can be given a source and destination between which they will travel (as in the act of carrying a resource from a supplying building to a destination building). This is the . . .

```
SetBuildingSrcAndDst {
  (supply1 dest1)
  (supply0 dest0)
}
```

section of the unit's definition. When the unit is selected, and the mouse is moved over one of the buildings in this section, the interface will allow a path to automatically be established between the pairs of buildings. Furthermore, if a building is created with an associated unit which has these pairs established (for a description of associated units, see the section on build.txt), then the unit will automatically be assigned a path from the building at which it was created to the nearest pair. Each parameter in the *SetBuildingSrcAndDst* section is a building identifier, as defined in build.txt. There must be a pair from each destination to source the possible path. In the above definition, the unit can either automatically path between supply1 to dest1, or supply0 to dest0. The interface will always attempt to use the closest buildings possible, unless the user specifically chooses otherwise. If it was required for the unit to either path between supply1 and dest1, or supply2 and dest1, then the section would look like this:

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
SetBuildingSrcAndDst {
  (supply1 dest1)
  (supply0 dest0)
  (supply2 dest1)
}
```

A Unit which Constructs Buildings
Any unit can be told it can construct buildings, and as long as the pre-requisite system says that a certain unit can build a certain building, then the unit will build. When this unit is selected solely, the build menu will automatically change to be the BuildingConstruction MFD, rather than the UnitConstruction MFD, and will list the buildings the selected unit can build, rather than the units that can be built.

Units that can Phase
Any unit (except a unit with movement mode *tunnel*) can be ordered to phase. This includes flying units (but it is up to you to come up with a unit definition that has it flying and phasing :).

Units with weapons that shouldn't AutoTarget Enemies
Sometimes units have control of weapons that should never auto target enemy units/buildings unless specifically directed to by the user. For this reason it is possible to include the *NoAutoTarget()* flag in the unit definition, which thereby prevents them from autotargetting. The unit will autotarget if its Independence is set high, however, if its independence is set to Medium or Low, then the unit will not Autotarget.

Units Morphing into Overlays
Units with the *CanMorphIntoOverlay(explosion_to_play_on_morph_explosion unit_cycles)* flag can morph into any overlay that has the *CanBeMorphedInto()* flag in the body of its definition. The *CanBeMorphedInto()* flag does not need any parameters. When the unit is told to morph, the explosion "explosion_to_play_on_morph_explosion" is played, and "unit_cycles" unit cycles later, the unit will morph into the targeted unit.

Units morphing into Units
Units with the *CanMorphIntoUnit(explosion_to_play_on_morph_explosion unit_cycles)* flag can morph into any unit that has the *CanBeMorphedInto()* flag in the unit's definition. When the unit is told to morph, the explosion "explosion_to_play_on_morph_explosion" is played, and "unit_cycles" unit cycles later, the unit will morph into the targeted unit.

Units Spying on Buildings
Units with the *CanSpy(minimumspytime maximumspytime busttime)* line in their definition are able to spy on enemy building/unit plans and see their resource usage, as defined in build.txt. The unit is ejected from a building on which it is spying after a random period between *minimumspytime* and *maximumspytime*. The unit is not allowed to re-enter any other player's buildings before the busttime time has passed.

Units with Camouflage Capabilities
Units with the *CanBlend()* flag in their definition are automatically camouflaged with their surroundings. When the unit is moving, the camouflage is less effective. Enemy and Neutral players find it difficult to see units with this flag. Allied players see this unit in the same way the player sees the unit, which is as a very weak camouflage type.

Units able to Carry Other Units
Units with the *SetCarry(number_of_bays max_weight_per_unit)* flag are able to carry a number of other units, and each unit is calculated to fit in a bay or not. Any unit that has a weight equal to or less than the "max_weight_per_unit" is able to enter the vehicle, up to a maximum of "number_of_bays" units.

Units which should not be placed Independently in the Scenario Editor
Sometimes there are units that should not be able to be placed in the scenario editor. Players should come by these units only through normal game play. This usually means that the unit in question is associated with a building, and should not be played in the game without the building.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

The Underground Transporter
CanBoomerang()
The underground transporter flag *CanBoomerang()* should only be placed specifically in the definition of a unit which is associated with a building, and which transports units. Whenever the unit unloads other units, the unit will return to the bay of the building from which it was created as a result of the *AssociatedUnit()* line in that building's definition.

Units Attached to Buildings
As described in the build.txt section, guard towers are special cases of units being attached to buildings. A unit that is to be attached to a building shouldn't have any special properties such as morphing and phasing. As described in build.txt, the unit is attached to the building at a defined pixel offset from the top of the base sprite. This unit will always have the movement mode fixed, and should have an entry in the trneff.txt file that has 0 speed over any terrain.

Boosting Units
Units with the *CanBoost()* flag have modified behaviour in the following ways: they will attack units on their own side that have health less than or equal to one-third, and will attack enemy units which have health greater than or equal to two-thirds.

CanGrab()
The Tactics Engine allows the conversion of "human" units into a previously defined unit. A unit with the *CanGrab(unit_identifier time_to_convert)* flag can attack and convert any human unit which is not on the player's side to the unit "unit_identifier" in its bay, and will take "time_to_convert" time to convert it. Note that the unit_identifier used in this flag must be previously defined in the UNITS.TXT file. To attack, this unit relies on the *IsHuman()* flag being in the victim unit's definition. The unit should be configured to have one bay, with a maximum weight per unit being whatever is needed (using the *SetCarry()* line in its definition).

Decoy Units
Decoy Units behave something like the Decoy feature for buildings. The Decoy for a unit must be defined after the real unit. The first line of the decoy's definition should be *IsDecoyOf(unit_identifier time_to_last)*. The rest of the decoy's definition should be whatever you wants the decoy to be, including sprites for the unit, unit description and so on. After being built, the decoy will last for time_to_last unit cycles. The player who owns the unit sees a bar which denotes the time left before the decoy unit dies. The player that owns the unit also sees the description as set in the decoy's definition body, where as other players see the description listed in the original unit's definition body.

A decoy of the example unit would be as follows:

```
DefineUnitType(FootUnitDecoy)
{
 IsDecoyOf(FootUnit)
 SetDescription(FootUnitThingDecoy)
 SetMenuImage(ftunitmn.spr)
 SetSelectSounds(yessir.wav haha.wav kilforfn.wav done.wav)
 SetResponseSounds(blah.wav hoopla.wav yippee.wav)
 SetSide(0)
 SetCost(300 9)
 UseEffects(Foot)
 SetMoveMode(Ground)
 SetStrength(100)
 SetPhysics(1 6)
 SetHitSize(5)
 SetSeeingRange(8)
 SetVulnerability(LeatherArmour 100)
 SetShadowImage(footshad.spr)
 IsHuman()
 SetIdleAnimation(50 100 2 2)
 SetRequirements {
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
    SetType(9)
    SetPrereqs(290)
    SetMaker(602 540)
    SetTechLevel(0)
   }
   SetRundownInfo {
    SetAttack(0)
    SetDefence(20)
    SetSquishSound("squish.wav")
   }
   AddPart {
    SetRotationRate(30)
    SetRotationalArc(180)
    SetScanDelay(500)
    SetImage(footguy.spr)
    SetMoveAnimation(0)
SetStandingAnimation(3)
    SetHealthExplosion(0 imdead_explosion attached)
   }
}
```

You will notice that we have removed the rubber band gun weapon from the decoy's definition. This is not necessary.

Units that can have Alternate States
Units can have two states, between which the health of the unit is preserved. This feature is really cool even though its very hard to describe.

Basically, if a unit has the CanAlternate(unit_identifier sprite_section) flag in its definition, then when the player clicks on the Alternate button in the Special MFD, the unit will change to be the unit as described by unit_identifier. When the unit alternates, it scales its hitpoints, so that the unit it alternates to has the same percentage of hitpoints as the unit it alternates from.

Once the unit is in the alternate state, it can be changed back to the original form by again clicking on the alternate button. There are many possibilities for this feature, including changing nothing but the weapon used by the alternate state of the unit. Because the unit basically becomes another unit type when it alternates, a lot or a little can be changed. The *CanAlternate()* flag is only placed in the unit which will be the initial form of the two types, and must point to a unit that has been defined earlier in the file (in other words, the secondary state of the unit must be defined before the initial state). When the unit alternates, it takes all the characteristics of the secondary unit, including name, movement type . . .everything.

Units with Weapons that Charge
ChargeWeapon(unit_cycles)
Standard weapons can be assigned ammunition amounts, so that the weapon can run out of ammunition at which point the unit might be sent back to a building to rearm. The charging ability *ChargeWeapon(unit_cycles)* allows the unit to rearm itself, and will take the time unit_cycles to do so. Units with this flag will not return to a rearming station when they run out of ammunition. The weapon for a unit with this flag should have 1 round of ammunition, and a 0 unit cycle delay between firing projectiles. The charging of the weapon takes care of the delay between shots.

Weapon.txt
Weapon.txt defines all the weapons units will be using to attack other units:

```
DefineWeapon(RubberBandGun)
{
 SetMovement(Linear)
 SetAnimation(eochfpr0_animation)
 SetAttributes(0 8 0 15 2000)
 SetSpeed(13.0  0.0  13.0 1)
```

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

```
SetOffense(C1 8 50)
CanShootFlyer()
SetFireSound(gxflkwc0.wav 80)
SetHitExplosion(chaff_explosion)
PersistentDamage(10 5 C1 3 50)
}
```

SetMovement()
The SetMovement() line defines the movement type of the projectile being fired. The two standard movement types are linear and homing. We'll explore them first.

SetAnimation()
The SetAnimation(animation_identifier) line defines the animation of the projectile while it is in flight.

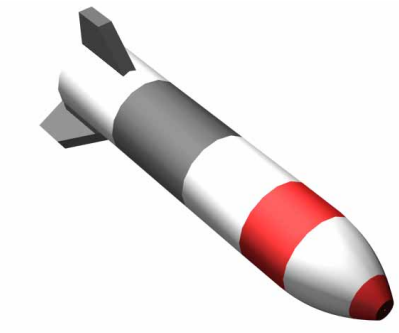
SetAttributes()
The SetAttributes(min_range max_range max_ammunition firedelay energypershot) line has five parameters, the last of which is no longer used. The min_range of the weapon is the distance away from the target at which the unit with this weapon must be to fire. The maximum range of the weapon is the distance outside of which the attacking unit is too far away from the target.

If the maximum ammunition of the weapon is set to 0, the unit has infinite ammunition. If the weapon is given a finite number of shots, the unit will then attempt to rearm itself. It will look for the nearest building with the *CanRearmUnit()* flag if the unit is *Movemode(GROUND)* or *Movemode(HOVER)*, or the nearest building with the *CanRearmFlyer()* flag if the unit is *MoveMode(FLYER)*. However, if the unit is set to be charging (units.txt), then the unit will not seek the nearest rearming building.

The firedelay of the weapon is the time between shots fired. This value is expressed in the number of unit cycles.

SetSpeed()
The *SetSpeed(initial_speed acceleration maxspeed rotationrate)* line defines the speed and rotation characteristics of the projectile. The projectile is fired at the initial speed of initial_speed pixels/unit cycle, and then accelerates at an acceleration pixels/cycle/cycle. If acceleration is greater than 0, then the projectile will not be able to go faster than maxspeed pixels/cycle. The rotation rate of the projectile is only applicable if the weapon is *SetMoveMent(Homing).* The rotation rate of a linear projectile should be set to 1. The rotation rate is defined in degrees/unit cycle.

SetOffense()
The offence of the weapon lists the offence class, the attack, and the blast radius of the exploding projectile. Damage.txt calculates how much damage to do to a unit based on which offence class the attack comes through on, and the hitpoint value listed in the line SetOffense(offense_class hitpoint_value area_effect). The area_effect of the projectile is expressed in pixels. Damage from an area effect is linearly scaled to 50% at the outer edge of the circle expressed using the pixel radius area_effect from 100% at the pixel hit by the projectile.



SetFireExplosion()
SetFireExplosion(explosion_identifier*).* This line in the weapon's definition activates the explosion explosion_identifier on hotspot 2 of the part to which the weapon is attached. This is the same hotspot from which the projectiles are fired.

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

SetFireSound(blah.wav volume)
SetFireSound(blah.wav volume) plays the sound effect blah.wav at volume "volume" where "volume" is an integer between 0 and 127 inclusive, and 127 is the loudest possible.

SetHitExplosion(explosion_identifier)
SetHitExplosion(explosion_identifier). This line in the weapon's definition activates the explosion explosion_identifier when the projectile fired from this weapon hits a target. Note that if the weapon is defined to be able to hit ground then this explosion will also be played when the projectile hits the ground. If the projectile missed its intended target, this explosion will be played when the projectile the point it was targetted at.

There are several flags in a weapons definition which define which types of units/buildings a weapon can hit.

CanShootGround()
The CanShootGround() line in a weapon's definition signifies that this weapon can attack a specific tile, without needing to target a unit on top of it. Weapons without this flag will be unable to target a tile without a unit on it. Weapons without the *CanShootGround()* flag in their definition will not be able to attack phased units.

CanShootGroundUnit()
The CanShootGroundUnit() line in a weapon's definition shows that this weapon is able to attack units with MoveMode(HOVER) and MoveMode(GROUND).

CanShootBuilding()
The CanShootBuilding() line in a weapon's definition specifies that the weapon can attack buildings.

CanShootFlyer()
The CanShootFlyer() line in the weapon definition allows the weapon to target and attack flying units.

CanShootIndirect()
CanShootIndirect() allows units with this line in their unit definition to attack outside their own line of sight, as long as the target is in range, and is in line of sight of another unit/ building. Units without this line must be able to see the target themselves, and cannot attack it if they cannot.

CanOnlyShootHumans()
The CanOnlyShootHumans() line in the weapon definition allows the weapon to target only units with the *IsHuman()* flag in their definition. A weapon with this line in its definition should still use one or more of the *CanShootGround(), CanShootBuilding(), CanShootGroundUnit(), and CanShootFlyer()* lines.

CanOnlyShootNonHumans()
The CanOnlyShootNonHumans() line in the weapon definition allows the weapon to only target units that do not have the *IsHuman()* line in their definition. A weapon with this line in its definition should still use one or more of the *CanShootGround(), CanShootBuilding(), CanShootGroundUnit(), and CanShootFlyer()* lines.

PersistentDamage()
This line gives the weapon a secondary attack that can be made to last for several cycles by attacking at periods after the initial attack by the weapon has been made. The *PersistentDamage(no_of_hits cycles_between_hits offense_class hitpoint_attack area_effect)* line

EMAIL
HELPDESK@AURAN.COM

SUPPORT
SUPPORT@AURAN.COM

specifies that every five cycles for 10 times the three hitpoints will be sent to the damage tables with offence class C1 and area effect 50. The last three parameters of this function are standard to the *SetOffense()* line as described above.

Weapon Type Shredder
This weapon type is very specific, and although the *SetAttributes()* and *SetSpeed()* lines need to be included in the weapon's definition, the values are not used in any way, and so should all be set to 0. This weapon provides a constant attack to the eight tiles around it, with offence class and hitpoint damage as defined in the *SetOffense()* line in the Shredder weapon's definition. The shredder weapon does not attack allied units, and will never damage any unit that is either the player's own or one of the player's allies.

Weapon Type Vortex
The only attributes used in the *SetAttributes()* and *SetSpeed()* lines follow, all other lines (SetOffense, targeting lines) are as they appear in the default weapon types, linear and homing.

For the vortex, the only parameters in the *SetSpeed()* line that are used are initial speed (parameter 0), and the rotation speed (parameter 3). The initial speed of the vortex weapon defines how long the vortex is active. The rotation speed of the vortex is a visual effect only, and defines how fast the vortex appears to rotate. Only the range of the vortex is derived from parameter 1 in the *SetAttributes(0 range 0 0 0)* line in the vortex's definition.

Weapon Type Wave
The only attributes used in the *SetAttributes()* and *SetSpeed()* lines follow, all other lines (SetOffense, targeting lines) are as they appear in the default weapon types, linear and homing. For the wave, the *SetSpeed(speed_of_wave no_of_bubbles bubble_distance 1)* line uses the first three parameters, the fourth has no relevance. The speed of the wave defines how fast the wave will travel after it is fired. The number of bubbles and the bubble distance define how long the wave will be. The bubbles are placed in a line perpendicular to a line drawn between the firing unit and the target, spacing by bubble_distance pixels between them.

Conclusion

As you can see, the Tactics Engine gives you a great deal of control over games made with it, and it also gives you the power to create entirely new games. At Auran, we particularly wanted to develop a game and an engine which gave players the flexibility they (like us) had been craving for a long time. There are so many features available to you that the results you achieve will only be limited by your time, your persistence and your creativity.

Of course, if you need any support, you can contact us through our support message board on our website.

Enjoy yourself and we look forward to seeing your new creations.