# Major Components of the Game

## Essential

* The playing field.
* The graphics engine.
* The game interface.
* Management of unit data.
* Navigation for successfully moving units from point A to point B.
* Unit states and actions.
* Unit interaction.
* Designing the game mechanics.
* Designing the races featured in the game.
* Creating balance, considering both the character stats/costs/spells, and whether one race has an environmental advantage/disadvantage with regards to map layout.

## Optional

* Networking for multiplayer play.
* Developing an AI for challenging cpu controlled players.
* Physics to modify unit-unit interactions and unit-environment interactions.

# The Graphics Engine

Very early on in the design, you must then decide on the kind of graphics your game will use. The options are:

* 2 dimensional tile-sprite based graphics (eg, Dune 2, Warcraft 2);
* 2.5 dimensional tile-sprite based graphics (includes isometric) (eg, Starcraft);
* 2.5 dimensional tile based graphics supporting three dimensional vector graphic units and buildings (eg C&C Red Alert 2)
* 3 dimensional vector graphics engine (eg, C&C Red Alert 3)

# The Nature of the Playing Field

The graphical nature of the game can be varied, but typically employs an overhead view of a playing field. However, alternative playing fields may be a spatial environment allowing full three dimensional movement, or a side view or platform-type environment, which offers a restricted two dimensional movement. Note that these alternative playing fields are in a minority as RTS games go.

Some of these have important drawbacks that must be considered during the design phase. For example, in the spatial environment the game play might be overly difficult because the game camera must always be looking in one direction, so getting adequate visibility might be a problem. Also, unless the game play is contained within a network of connected rooms and tunnels, a spatial environment consisting of the odd planet and starbase will provide few environmental obstacles, so maps will have no real structure, and no obvious boundary.

We will continue this discussion on the assumption that the playing field is a rectangular area of ground, about which the player take an overhead view. We may assume there is a varying height levels from unevenness in the terrain, while some discrete changes in elevation are permitted to create cliffs that provide a natural obstacle for walking characters. Additionally, terrain may be lowered to represent pooled water (another natural obstacle), and cliff edges can be eroded in selected places into slopes which allow those cliffs to be negotiated. This would be a state-of-the-art environment.

Once the environment is decided, the next step is to decide on how that is going to be represented. It's fairly important to decide this early on, because affects the choice of graphics engine, which is normally where a developer starts when programming a game.

## The Game Map

Typically, the ground contour would be represented by a rectangular mesh layed out in the (x,y) plane, while vertices of that mesh are raised along the z axis to suggest height. This could be represented through an array structure:

Terrain[x,y].Z

In typical RTS games, x and y could range from 32 for small maps upto 256 tiles for large maps, although depending on how fine the landscape detail you want to have (visible resolution), these figures might need to be scaled up or down.

Furthermore, each of these locations would normally index a texture graphic, such as grass or soil, or a cliff tile texture graphic to be pasted over the mesh in that grid location.

Terrain[x,y].Texture

You may choose to have more than one texture and to blend the them together across the mesh to create the appearance of stone fading to soil, for example. It depends on how complicated you want to take it.

There are a few important points to consider when determining the map description:

How are characters placed on the map? Is it one character per grid location only? Or can characters move freely around the terrain, and are not constrained to the grid locations?

How are cliffs going to be represented? (1) Will one grid location be a cliff wall, while the grid locations one step either side of that are the lower and upper plateaus? OR (2) Do two plateaus butt against each other and the cliff edge between them does not have it's own grid location?

Each method has it's own advantages, and affects the programming later on. Don't decide this trivially. Consider how slopes can be represented in each. How might these plateau levels be represented? As integers? And for other factors like visibility, how can plateau levels be used to determine visibility?

Terrain[x,y].PlateauLevel

You might also need to represent navigable space in the map description. If cliffs cannot be walked over, then you need to include this detail somewhere in the map description. You may use the plateau level data to determine where a cliff edge is, or you may use another terrain property to say what locations can be walked on or not walked on, such as:

Terrain[x,y].Navigable

# Navigation

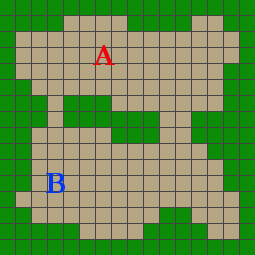
One of the more mind bending parts of the RTS engine is the pathfinding. Whenever a unit moves, then a path must be followed. That path must be computed.

This would include:

* Giving a direct order to a unit to move;
* A unit auto-acquiring something hostile nearby;
* A resource gathering unit moving to collect or return resources;
* a group of units moving at different speeds or in different directions, so that some are trying to overtake or move past others.

The pathfinding is closely related to the map definition. ie, the solution must be compatible with the way the map is defined.

One way of proceeding here is to define a pathing map, which defines true/false which squares can be walked on and which cannot. If two adjacent squares can be walked on, then we are usually able to walk between them.

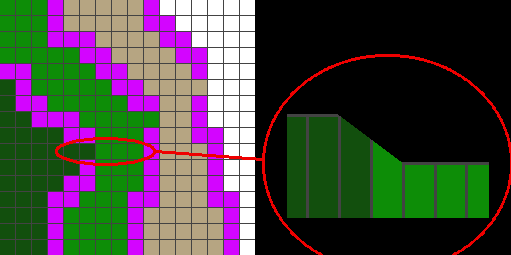
The image below shows a simple pathing map. 

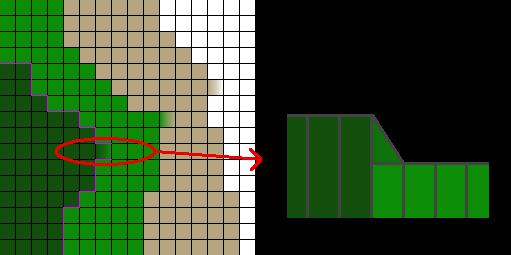
This can be solved easily with an algorithm like A-star (although that may not be exactly what we want from a path, it is nonetheless a good starting point).

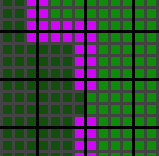
One feature we may want to consider is the collision radius of a unit. A small unit may have a collision radius smaller than one grid square, in which case it would fit through the gap on the left. A larger unit may not, and may have to go the long way around. A bigger unit than this may not be able to move from A to B at all.

A second issue with the map definition is cliff edges and similar environmental features.

One method of describing cliffs is to reserve a grid square for them as shown below. This makes pathing easier because we can use the same algorithm as above. However, the non-walkable cliff edge tiles do use up space that can make our map feel bulky.

An alternative description is to define tile boundaries as walkable or not, rather than the tiles themselves. In this description, cliffs are not represented by tiles, but in the juxtaposition of two tiles of different height. This requires a subtle change to our algorithm, but it still quite doable.

A pleasant compromise between these two methods is to subdivide the grid into subtiles, making the pathing map more densely defined.



This allows us to define non-walkable boundaries while keeping a simple walkable/non-walkable description suitable for A-star.

# Unit Data

The core of the RTS game concerns the management of the unit data.

Because this type of game involves creating several 'copies' of the same few unit classes, (e.g. you might train 12 marines, which is 12 active copies of a predefined marine), you might find it useful to organize unit data in terms of templates and instances.

* One list provides static templates for each kind of unit : UnitClass[j]
* Another list stores the details of created units: Unit[i]

(Unit data created from templates will interchangably be referred to as an 'instance' on this page.)

Typically, when a unit is created in game, a new entry can be created in Unit[i], and the unit data is copied to that from the appropriate template, UnitClass[j].

So it is advisable at this stage to spend some time listing all properties associated with a unit, however it might be used in the game.

You will discover that some of these properties (such as a name, like "Marine") could simply refer to a value in the parent template, rather than duplicating that data string many times over in each instance of the unit. Other examples of static data might be: references to graphics used to draw this unit; or the resource costs when building it; or the prerequisites (tech-tree requirements) for building this unit.

For static data, you can simply have a unit instance carry a back reference to the template, i.e.

unit[i].Class = j

Where UnitClass[j] was the parent template for Unit[i].

Then if the static properties of the unit class was given,

UnitClass[j].ClassName = "Crypt Spider"

UnitClass[j].GoldCost = 240

then unit[i] can retrieve the static data as

UnitClass[ Unit[i].Class ].ClassName

// or

UnitClass[ Unit[i].Class ].GoldCost

Notice that additional static data doesn't add more weight to the instanced unit data.

An example set of properties (with example figures) might be:

UnitClass[j].ClassName = "Crypt Spider"

UnitClass[j].SpriteGraphic = Crypt\_Spider

UnitClass[j].BaseHitpointsMax = 550

UnitClass[j].BaseHitpoints = 550

UnitClass[j].BaseArmor = 4

UnitClass[j].BaseDamage = 30

UnitClass[j].AttackRange = 4.0

UnitClass[j].BaseReloadTime = 2.0

UnitClass[j].BaseWalkSpeed = 3.2

UnitClass[j].FoodUsed = 3

The reader should now spend some time considering what data is static and which can change throughout the game. Sort you list of properties into two groups, which properties are static throughout a game, and which can change within a game.

## Unit Upgrades

It is popular in many RTS games to make upgrades available for the different units. For example, several unit classes might benefit from the same researched upgrade, such as increasing armor of all melee units. It is worth considering at this point, how upgrades to units will work. Ask yourself how could this be implemented given the way you choose to store unit data.

Obviously, an upgrade isn't static data, although when it is researched, it is applied universally. So in the case of a unit class’s base armor (base value before armor bonuses)...

* The first solution could be to have the unit base armor appear in the instanced unit data, defined such as:

Unit[i].BaseArmor

...and when an upgrade is made, the changes to BaseArmor must be applied to all valid future built units, and also to all valid existing units generated from this template. That's simple and straightforward.

But there are other options making use of the static data concept to reduce memory usage:

* We might be tempted to have the base armor refer back to the parent template as a static, such as:

UnitClass[ Unit[i].Class ].BaseArmor

...and an upgrade which increases base armor simply updates this template value. Then all instances of this unit class will then automatically inherit the upgrade.

However, there is a problem that if two or more players have units from the same template (aka a mirror game), then a single upgrade would benefit both players, which is not a good idea.

What has to be done here is a workaround, allow each player a separate copy of any "upgradable" static data.

* A third option, and likely to be better, is to list used upgrades as one of the fields in the unit templates.

UnitClass[j].UsedUpgrades = (some code representing the upgrades)

Existing instances of units can then access this information through back reference, and if they are eligible for the upgrade, then it can be applied to the instanced unit. Since the original template remains unchanged, newly created units must have any upgrades applied at time of creation. What this method gives is the advantage of not altering the template, so it will work in mirror games.

* A fourth method, perhaps the most obvious but most computationally demanding, is to apply the effects of upgrades on the fly when making calculations using the unit data. Not that we want to encourage premature optimization, but this may not be favorable if the number of units in the game is large.

## Unit Position Data

Instanced units will all need a position in the playing field. This can be set up relatively easy as follows:

Unit[i].x

Unit[i].y

Unit[i].z

Here, x and y would represent the position of a unit in the playing field.

Height (z) could normally be determined from the ground level at location (x,y) so it isn't strictly necessary to have a height vector. Even air units can be listed with a nominal height above ground level, and their height be calculated on the fly from that. However, there may be situations where this height can change, like if ground units are lifted up by a spell of some kind, or if air units are grounded by some other kind of spell. Having a height vector gives us the freedom to adjust height. It isn't strictly necessary here, since these spell induced height changes could also be calculated on the fly as the spell waxes and wanes.

Furthermore, we may not want air units to strictly follow the ground contours, as this could create some artificially steep climbing/diving as it passes over cliff edges. So having a height vector allows us to smooth out the altitude changes of mobile air units when passing over uneven ground. But again it isn't strictly necessary, since we could pre calculate a smoothed contour map specially for aerial units to maintain a fixed altitude over.

A third example might be if some ground units are able to negotiate cliffs, then their 'jump' trajectories will involve height changes. Once again, there are various ways to avoid implementing a separate height vector.

A final example which carries greatest merit, is if units are desired to have physics behaviour, such as collision dynamics, or gravity based falling, or where altitude is a variable factor in game play.

It is upto the reader to determine the purpose and value of having a height vector in the game you have designed.

Other positional data might be:

Unit[i].angle

which describes which way the unit is facing.

All these, of course are relevant only in the instanced unit data, and would serve no function in the unit class templates.

Other properties relevant to the unit class might be a movement speed or a turn speed, although these may not be static properties depending on how speed is handled.

One interesting problem that the programmer will need to solve, is getting a unit to turn to face a particular direction. Typically, this direction will take the form of a target position, which when measured relative to the unit's own position will give a direction vector. There is an article on gpwiki discussing this little problem.

So all in all, the reader should now be aware that some data is relevant only to the unit templates, some is relevant only to instanced units, while with some properties it isn't clear, and there isn't a definite correct way.

# Unit States and Actions

(content required)

# Unit Interaction

(content required)

# The Game Mechanics

(content required)

# Designing Races

## Counters

The basic idea of counters is that some units are stronger than others, but in such a way that there isn't a best choice of unit as all have relative strengths and weaknesses. Rather like Rock-Paper-Scissors.

To develop counters, it is helpful to have different types of damage and armor. Here's some examples:

Blunt damage is done by hand/melee weapons like axes and maces. Sharp damage is done by fast moving projectiles like arrows and bullets. Explosive damage is done by siege weapons like mines and grenades, or heavy projectiles like cannon balls. Magic damage is an umbrella term for non-physical damage done by casting units, such as wizards, priests, medics, or poison gas. "Magic" easily penetrates unit armor to affect the body within.

Medium Armor is an average performance armor. Hard Armor is something like inflexible metal plates, that can readily deflect heavy blunt weapons, but is easily punctured by sharp arrows and bullets. It is used on vehicles (Mech) and some tanking units. Soft Armor is soft flexible armor, like kevlar or leather, used on biological units. It's good at absorbing the energy of sharp arrows, but offers little resistance to heavy blunt weapons. Fortified Armor is used for buildings. These are often constructed with heavy aggregate materials that are mostly impervious to sharp weapons like arrows or blades. On the other hand, they are easily cracked and destroyed by explosives.

We can put that data into the form of a table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Relative effectiveness of armor | | | | |
|  | **Soft Armor** | **Medium Armor** | **Hard Armor** | **Fortified Armor** |
| Blunt Damage | Worse | Average | Better | Average |
| Sharp Damage | Better | Average | Worse | Better |
| Explosive Damage | Average | Average | Average | Worse |
| Magic Damage | Worse | Worse | Worse | Better |

As you can see, this creates some natural counters:

* Units with magical attacks (casters) are effective against units with medium armor.
* Units with sharp weapons (ranged) are effective against units with hard armor (mech).
* Units with blunt weapons (melee) are effective against units with soft armor (bio).
* Units with explosive weapons (siege) are effective against fortified buildings.

There are two philosophies for designing counters, and both have their pros and cons:

Hard counters is most like Rock-Paper-Scissors, where a particular type of unit can easily be defeated by another particular type of unit. These relationships will cycle somewhere so that A>B>C>D>A, or whatever. Hard counters are often liked because they make a game easier to understand. Gameplay is about observing what your opponent is creating, and then creating a counter to it. In the relative effectiveness table given above, the numbers would be quite diverse. eg better = 200% effective, worse = 33% effective.

Critics of hard countering say that gameplay lacks creativity because players have no real choice is what units have to be trained. Gameplay tends to favour macromanagement as teching and observation is crucial getting the right counters in the right place at the right time.

Soft counters is where all units have similar usefulness, and there aren't really any bad choices. E.g. some units will have high HP and low damage, while some units will have high damage and low HP. On their own, each will perform similarly, but in synergy they can each play their strengths: high HP units operate as tanks to absorb damage, shielding the high DPS units who'll be safely delivering damage. So in soft countering games, mixtures of two or more unit types is better than any one unit type. So gameplay is all about finding effective unit combinations.

Critics of soft countering say that there will always be a few unit combinations that are most effective, and players will always tend to go for these which can make the game boring. Gameplay tends to favour micromanagement as unit combinations have to be controlled effectively.

## Basic Balancing

In the beginning, while you're making/testing your game, you may want to define just one race, or one incomplete race, But sooner or later you will want to design things more fully.

Designing races is an art as much as it is a science. It is all too easy to end up with a dozen units that are barely distinguishable.

The first thing to do here is to set the pace of the game. How quickly do you want combat to be over? How quickly do you want a group of units to cross the screen? How quickly do you want a group of units to cross the map from one base to another? How long should a game last? How long should it take to build a big army?

There are four basic statistics to think about when designing units. Each of these stats when increased makes a stronger unit:

* Hitpoints;
* Damage per second;
* Movement speed;
* Attack Range.

All in all, you don't want any unit disproportionately strong or weak. Players will call that "imbalanced", and not want to play. ;-)

When I say disproportionately, I mean relative to food usage and resource costs and training time, so yes, there are four more statistics to think about for setting pace:

* Resource cost;
* Training time;
* Food costs (relative to what you can supply, and how quickly you can supply it);
* The pace of your economy.

Now balance is key to good design, but there are no rules to making things balanced, but it is helpful to look at these three things:

What kind of army can be built in 5 minutes? This is known as the early game when your first few units are constructed. You are usually very limited with what you can do at this time, and also very vulnerable. Launching an attack in the early game is known as rushing. If race A can build a stronger attack force in 5 minutes than race B, then that is bad. Whatever units you make available in the early game, they should be strongly balanced, and without regard to what comes later in the tech tree.

What kind of army can be built with one set of resources? You will typically start a game with a fixed deposit of resources beside your base. If you are able to harvest all of this and construct an army -- without expanding to another location -- then how strong is that army? If race A can produce something stronger than race B with those resources, then that is bad,

What kind of army do you have with max food? If you have a food supply limit, and it can realistically be reached, then races should be balanced with respect to this also. If race A and B both max out supply, but race A is stronger, then that is bad.

## Food Templates

While all this may sound intimidating, there are ways to help yourself design races. What you may find it useful to do is design what I call an "Food Template" for all units to stem from.

An Example Food Template: Food Used: F Hitpoints: 200\*F Damage per second: 8\*F Armor: 0 Movement speed: Normal Attack Range: Melee Theatre: Ground unit that attacks ground units only. Spells/abilities: none Resource Cost: 80\*F Training Time: 10\*F seconds

Then in my game, all my unit designs might start out as clones of this template, and I would adjust figures to create preferred roles and variety.

And to note, this wouldn't be a particularly fast paced game. Two identical units in a dual would be taking around 25 seconds to die (200F/8F=25). For a fast paced game, you would want this closer to 10. Or for slower paced go up to 40.

eg, with F=3, we have : food used=3, hitpoints=600, dps=24, armor=0, speed=normal, range=melee, theatre=ground, spells=none, resource cost=240. training time=30s

Also to note: Larger units (that take larger values of F), can often be weaker than expected in a game, because they can be impeded by a fewer number of curses. (4 units that take 1 food each, will be disabled with 4 crowd control spells cast upon them. Whereas 1 unit that takes 4 food, requires just 1 crowd control spell to disable it. A much easier feat.)

The way this is often balanced is by giving these larger units a bit more armor, and pushing them higher up the tech tree, so that they are available at much the same time as units bearing crowd control spells. Obviously, if these heavily armored units were available early in the game they would be imbalanced, but once crowd controls spells become available, they becomes significantly weaker threats. :-)

## Unit Roles

It isn't enough just to provide the F templates as is, because although these could be considered fairly well balanced units, they are extremely bland.

So, we introduce roles and counters, which encourages some units to be mixed and matched to increase the versatility of the group, or to focus on a specific role.

### Melee Units

General purpose Melee. One of the key units in the early game is the melee unit, that the F-templates are based upon. Generally this is upgraded with hitpoints, and is unlikely to have any special abilities. We may give this unit Blunt Damage and Medium Armor.

Ninjas. Similar to the general purpose melee, except this is a dps unit, that cannot tank. This needs lower hitpoints, but higher speed. Players might use this speed to outrun an enemy to attack workers, or for chasing down other units that have become isolated for whatever reason. This generally receives upgrades to speed and damage.

Tanking units. Another unit similar to the general purpose melee, except this is a heavyweight that has larger amounts of armor. To balance this bonus, we must reduce one of the other stats slightly, such as dps. We may also give this unit Hard armor, that gives it some advantages and disadvantages for countering.

### Ranged Units

General purpose Ranged. Another key unit in the early game is often a ranged unit. Ranged units have an immediate advantage in that they attack at distance, so already we must balance out this bonus by reducing the dps or the number of hitpoints. Generally speaking, ranged units are upgraded with increased range, damage and/or rate of healing. We may give this unit Sharp Damage and Soft Armor.

Support unit. A unit that is useful late game is a support unit, that attacks at range doing Sharp Damage with Hard Armor, and as such it's an effective counter to itself. A support unit is one that cannot hold it's own without protection from other units (eg, attacks infrequently so cannot defend itself), but in synergy with other units is very effective. It may also do splash damage (reduce dps to balance), or confer an aura to friendly units (reduce speed to balance).

### Air Related Units

Primary Air This is an air based unit that can attack ground units. Being air based is not usually advantage enough to justify penalising it's stats in some way. As long as counters are available at roughly the same time in the game, that will be enough. Air units will often have Hard Armor, or whatever is weak against the sharp damage of ranged units.

Anti Air. This can be a ground unit or an air unit, that is the most effective counter to other air units. As a a ground unit, it may be that the General purpose Ranged unit can effectively attack air. As an air unit, depending on how late in the game it becomes available, it would also serves a role as a scout, negating the need for a separate scouting unit.

### Siege Units

Raider. Towards the mid game, one favourite units is the so called raider, that is similar to the Ninjas, except that it is Mech (Hard Armor)and does Explosive damage and attacks at short range. It may also be able to train abilities relevant to it's role, such as an ability to trap units, or an ability to negotiate cliffs.

Artillery/Siege. Another favourite mid game unit is artillery. This typically does larger amounts of damage to buildings, and also Area-Of-Effect damage if it attacks ground units. These attacks are typically at long range, so it is the primary counter to defensive structures. The negatives are that it attacks infrequently, is slower to move, and has much less hitpoints. Destructive but slow and vulnerable. Typical upgrades increase damage, but artillery may have additional abilities, such as the weapon being charged (costs extra to fire a shot at higher power), an upgrade allowing it to shell longer range but cannot move, or an ability to launch flares to reveal an invisible area of the map, etc. We may give this unit Explosive damage and Medium Armor.

### Casting Units

Offensive Casting unit. These generally do greatly reduced damage and have low hitpoints, but a great ability to confer buffs and curses to combatants. This makes curses on enemy units or does AOE damage spells. We may give this unit Magic damage and Soft Armor.

Defensive Casting unit. Similar to above. This heals or protects or debuffs your own units.

Where I write things like higher speed and lower hitpoints, it will be values within 10-25% of what the F-template provides... or thereabouts. You'll have to experiment with exact figures and see how your units play. :-)

# Network Play

(content required)

# CPU Controlled Players

(content required)

# The Game Interface

(content required)

# References

<http://content.gpwiki.org/index.php/RTS_Design_Guide>

**Map Units**

1. Trainers
   1. A Trainer is the primary unit the player will use in a turn.
   2. A Trainer can perform a variety of actions in a turn:
      1. Move to a new tile
      2. Interact with objects on the map, such as Treasure
      3. Enter a Building and perform actions in the Building
      4. Command a Minion to perform an action
      5. Capture a wild Minion
      6. Perform a special Power
      7. Attack a Trainer
   3. Each player starts with at least one Trainer.
   4. Trainers can be built at Base Buildings
   5. Trainers can carry a finite number of Minions
2. Minions
   1. Minions are the offensive units of the game.
   2. Minions have a type, or a combination of types.
      1. Types can be strengthened or weakened by one another in combat
      2. Types influence what types of actions the Minion can learn and perform
      3. Types influence which Minions are genetically compatible for breeding
   3. Minions are found wild throughout the game map.
      1. The type of terrain influences what kinds of Minions can be found
      2. A Trainer can capture a Minion by starting a battle with it
   4. Minions gain experience points from performing actions and participating in battles
   5. Actions are performed in the Map View, and Battle Moves are performed in the Battle View
   6. Each Minion has a signature Building it can construct
3. Buildings
   1. Each player starts with a Base Building at the beginning of the game
   2. Base Buildings are built on fixed, on-map natural structures. The type of natural structure gives the Base Building certain characteristics
   3. Base Buildings can build Trainers
   4. Minions are responsible for constructing Buildings, and each Minion has a signature Building it can construct
   5. Some can research Improvements
   6. Others have cool miscellaneous abilities

**Gameplay**

1. Two main game views:
   1. Map View
      1. Player can see all their Trainers and Buildings
      2. Player can move their Trainers throughout the Map
      3. Player can perform actions with their Buildings
      4. Goes to Battle View when a battle begins
   2. Battle View
      1. Starts when a Trainer goes into battle with a wild Minion or another Trainer
      2. Both sides use Battle Moves on one another until someone retreats or is defeated
      3. Goes back to Map View once the battle is complete
2. Players each get a turn using their Trainers and Buildings
   1. Buildings usually use resources, or internally limit the amount of actions that can be performed in a turn
   2. Trainers have a limited distance they can travel in a turn, determined by the Trainer’s stamina attribute.
   3. If a Trainer enters battle with a wild Minion or another Trainer, screen enters Battle View

**Class Overview**

*High-Level Game Screens and Helpers*

* TanksGame
  + Maintains the current state of the game
  + Switches screens when necessary
* Assets
  + Provides static access to loaded assets
  + Controls the assetManager
  + Loading screen?
* MapScreen
  + Screen for displaying the map, performing your turns, etc.
* BattleScreen
  + Screen for doing battles
* StartScreen
  + Starting screen for the game
  + New game, Load game, Options, Exit

*Map Screen*

* TerrainGenerator
  + Generates simple maps using a set of baseId’s
  + Uses diamond-square algorithm
  + Can get seed maps to “fuzz” with randomness
* TerrainLayer
  + Has a set of available tilesets
  + Handles the application of “modifiers” to the tiles
* TerrainTileSet
  + Contains all the transition tiles for 2 or more terrain tiles
  + Uses 13 terrain tiles to represent possible terrain borders
  + Some terrain combinations are not possible
* TerrainTileSet.Cell
  + Holds the baseId, modId
  + Holds the actual tile to be drawn