Introduction

This assignment involves a much more complex game than Tic-Tac-Toe, and in order to create it you’ll need to add several features to your engine. You’ll be creating a tactical strategy game in which units move around a grid-based map. The enemy units will be controlled by the computer, so your engine will need to support some basic AI capabilities in order to provide an active opponent.

While this project is more open-ended than Tic, we will require that you implement one of two different types of games. You will need to decide which kind of game you will implement by the second week of the project. Each of the game types will have slightly different game requirements, but both are designed to be equally challenging to implement. The two options are:

- **Real Time Strategy (RTS).** In an RTS game, the participants position and maneuver units and structures under their control to secure areas of the map and/or destroy their opponents’ assets. Many RTS games allow the participants to create additional units or structures using gathered resources. The *Starcraft* and *Age of Empires* series are two popular examples of RTS games.

- **Turn Based Strategy (TBS).** A TBS game, like an RTS, involves participants controlling units to defeat an opponent. However, in a TBS game, the players take turns controlling their units. A typical TBS game will allow the player to move each unit once per turn, and after moving, the unit may perform an additional command, such as attack, defend, or capture. The *Fire Emblem* and *Advance Wars* series are two popular examples of TBS games.

**Week 1 – Due Sep. 20**

In the first week of this project, you will make a simplified version of the game with vector graphics instead of sprites. There will be no computer player yet, and your units will not be able to move, but you will visualize a loaded or algorithmically generated map.

**Design Check**

- How will you represent your grid-based world so that it can support drawing, interacting with units, and eventually pathfinding across a graph?

- Will you use AffineTransforms? If not, how will your viewport perform the appropriate transformations?

- How will you incorporate viewports into your existing UI toolkit?

- Will you implement file parsing or procedural generation? What is your plan for whichever you choose?
Engine Requirements

- **Viewports.** The engine must have a “viewport” UI element that can draw game objects from a different coordinate system onto the screen coordinate system. The viewport must support panning (translation of the currently viewable game state), zooming (scaling of the currently viewable game state), and event interaction with the game state. When zooming, the viewport must either keep the center of the game in the same position or keep the location currently under the mouse in the same position.

Game Requirements

Since you have not implemented movement yet, the game requirements for this week focus on visualizing certain aspects of your game:

- Using a viewport, the game must visualize state that consists of a grid of tiles:
  - The grid must contain at least two different kinds of tiles, each of which is visualized differently.
  - At least one of the tiles must contain a unit, which should be drawn on top of the tile that it occupies (note that units do not have to move this week).

- The game state must be created from one of the following strategies (be sure to document whichever you choose in your README):
  - Loading from a map file:
    * Map format must be plain text format (no serialized java files please).
    * Map format must, minimally, have passable and unpassable tiles.
    * Map format must, minimally, providing starting tiles for player and AI units.
    * Must handle bad map files gracefully.
    * Must provide at least two different map files.
    * Map format must be clearly documented in README.
  - Procedurally generating a map:
    * Map must be partially randomized based on a seed. This means that using the same seed two times would produce the exact same map.
    * Map must have comparable complexity to one generated by Perlin noise.

- There must be a main menu screen that allows the player to either choose a map to load or choose a seed to generate a map. For procedural generation, having several buttons offering different seeds is acceptable.

- Player must be able to pan the view.

- Player must be able to zoom the view in and out.

- Player must be able to click on the grid and see an indication of the selected tile.
Suggested Extras

This week focuses on implementing your viewport and visualizing your map. Since you’ll be using your viewport for the rest of the class, it makes sense to give it some additional functionality:

- Give your viewport limits on viewing parameters:
  - Don’t let the player zoom too far in/out.
  - Don’t let the player pan out of the view of the game.

- Give your viewport momentum-based panning and zooming, which will provide a smoother transition between two locations of game state.

Week 2 – Due Sep. 27

In this week you will add sprite graphics to your game by adding support for sprite images and animations to your engine. You will also start adding AI by developing the pathfinding algorithm that human and computer controlled units will use to navigate the map. You should also decide if you will make an RTS or TBS game.

Design Check

- Will you make an RTS or TBS game?
- What heuristic will you use for your A* pathfinding, and why?
- How will you represent your logical grid (which is a graph) vs. your visual grid (which is your game state used for drawing purposes)?
- If RTS, how will you prevent units from occupying the same tile?
- If TBS, what kind of system will be responsible for keeping track of who’s turn it is, and rotating turns appropriately?

Engine Requirements

- **Pathfinding.** The engine must support performing A* pathfinding over a graph with non-negative edges. The game must be able to define a graph and a heuristic and use the engine to determine the shortest path between two nodes.

- **Sprite-sheet images.** The engine must support drawing sprite images that are a logical sub-rectangle of a single image, either by splitting the sprite sheet into separate images or by drawing only a portion of the sheet image.

- **Shared images.** If the same sprite-sheet image file is required multiple times (for example, by having multiple tile definitions in the map file specify it), the engine must support storing only one copy of that image in memory and sharing it between sprites that use it. If the engine splits a sprite sheet into smaller images for each sprite, it must still load the sprite sheet only once.
• **Animations.** The engine must be able to create an animation that consists of multiple sprites played in succession.

**Game Requirements**

If you read in a map file in Week 1, you will have to modify your map format to include sprite information for *at least the tiles*. You *do not* have to include animation information for your units.

In addition, your game must have the following features:

• There must be passable and impassable tiles.

• Units may never move onto impassable tiles.

• The map must be displayed in a way that communicates which tiles are passable and impassable, and which tiles each unit currently occupies.

• The player must be able to select human-controlled units.

• Unit selection must be visualized in some way (i.e. by drawing a border around selected units).

• The player must be able to command selected human-controlled units to move to any tile on the map. The units must move to that tile if possible, but they do not need to do anything if the destination tile is impassable or if there is no route to it from their current tile.

• Units must move to their destination tile using A* pathfinding.

• Units must move smoothly between tiles, but can only stop in the center of a tile.

• When moving, units may only change direction when in the center of a tile.

• Units must be represented by animated sprites. The sprite to use for each type of unit does not need to be specified in the map file (so it can be hard-coded into the game).

• Unit animations must play while units are moving and stop when the units are not moving.

• If you are making a Real Time Strategy game:
  
  – Units must not overlap at any time.

• If you are making a Turn Based Strategy game:
  
  – Before moving, your game must visualize the path a unit will take to reach it’s destination.
  
  – Units of different teams must never overlap.
  
  – Units of the same team may pass through each other, but may not permanently occupy the same tile.
Week 3 – Due Oct. 4

In this week you will complete your game by adding an AI computer player. There will also be a gameplay requirement in which we give you a number of potential gameplay features to implement, and you get to choose which to complete!

Design Check

- Can you explain/describe how a behavior tree works, focusing on the different kinds of nodes?
- How are you planning on using a behavior tree to design your enemy AI? Have an example of a tree you might construct for a particular task.

Engine Requirements

- **Behavior Trees.** The engine must support basic behavior tree-based AI functionality:
  - Action
  - Condition
  - Selector
  - Sequence
  - Random Selector (Optional)
  - Inverter Wrapper/Decorator (Optional)
  - Loop Wrapper/Decorator (Optional)

Game Requirements

- Units on opposite teams must be able to damage each other.
- Units on the same team must not be able to damage each other.
- When units become reasonably damaged, they must be destroyed and removed from the game.
- Units on the computer’s team must be controlled by a behavior tree-based AI with at least the following nodes:
  - 1 selector
  - 1 sequence
  - 2 distinct conditions
  - 3 distinct actions
- There must be at least one form of UI element that is drawn “on top of” the game’s viewport rather than “inside” it, but whose position is based on a point “inside” the viewport. This means the element is drawn in the screen’s coordinate system rather than the viewport’s coordinate system, so it does not scale with the size of the viewport, but its position changes to follow a point inside the viewport. An example would be a health bar that hovers above each unit, following the unit while it is visible but not scaling when the map is zoomed.
• It must be possible to return to the map select screen upon completing a game.

• The game must implement “deathmatch” style gameplay. This means that a player wins by eliminating all opponent units before losing all friendly units.

• When the game ends, a win or lose message must be displayed to the player, providing an option to return to the menu screen.

• You must implement at least four of the following features. Note that some features may not make sense in the context of the game you are making, and that nested bullet points should only be done if their parents are done:
  – Allow for multiple-unit selection, perhaps using a draggable selection box or ctrl-clicking.
    [RTS]
  – Distinguish between impassable tiles that block ranged weapons, and impassable tiles that ranged weapons can still pass through. [RTS]
    * Instead of tiles always blocking ranged weapons or not, have each tile have a concept of height: ranged weapons are blocked only by tiles that have a height greater than the height of the tile they were fired from.
  – Allow for 2 human players to compete. [TBS]
    * Allow for N human players to compete. Note that you will likely need to edit your map file or generation to support the appropriate placement of units.
  – Create at least three different unit types with different attack or support capabilities.
  – Fog of war: player can only see the map area around their units.
  – Make units upgradable based on a limited resource.
  – Create a system for spawning units based on time or an economy.
  – Create at least two more tile characteristics besides passability (destructible, movement cost, defensive value, etc), and make at least three tiles with different variations of these characteristics.
  – Add a floating minimap UI element.
  – Make a second game mode besides deathmatch with a capable AI to play it, such as capture-the-flag, king-of-the-hill (capture and defend a location), or sabotage (protect a certain location/unit for a certain amount of time while the other team attacks).
  – Any other feature(s) you can think of! Just approve it with a TA!

• It must be possible to start a new game without restarting the program.

Map File Format

One possible format for the map file is described below. If you use this map file format, you can use the sample map files provided by the TAs (they’re in /course/cs195n/support/tac).

The map file will be divided into three sections: dimensions, terrain, and units. The sections will have the following lines:
• Dimensions: One line consisting of “<width> <height>”, two integers indicating the size of the map in tiles.

• Terrain: Begins with “TERRAIN” on a line by itself.
  – Any number of lines in the format “<code> <boolean> <boolean> <file> <int> <int>”, each of which defines a type of terrain tile. <code> is a two-character identifier for the type of tile, the first <boolean> is “TRUE” or “FALSE” indicating whether the tile is passable for units, and the second <boolean> is “TRUE” or “FALSE” saying whether the tile is passable for projectiles. <file> indicates a sprite file, and the two <int>s correspond to the zero-indexed column and row respectively, numbered from the upper left, of the sprite to display for the tile.
  – “START” on a line by itself indicates the start of the tile layout grid.
  – <height> lines, each of which has 2 * <width> characters. Each pair of characters corresponds to a <code> for a tile type.
  – “END” on a line by itself indicates the end of the tile layout grid.

• Units: Begins with “UNITS” on a line by itself.
  – Any number of lines in the format “<code> <boolean>”, each of which defines a type of unit. <code> is a two-character identifier for the type of unit, <boolean> is “TRUE” or “FALSE” indicating whether the unit is AI-controlled (true indicates computer’s team, false indicates player’s team).
  – “START” on a line by itself indicates the start of the unit layout grid.
  – <height> lines, each of which has 2 * <width> characters. Each pair of characters may correspond to a <code> for a unit type, or it may be filler text. A unit type code indicates the unit that occupies that grid position, while anything other than a unit type code indicates that there is no unit in that position. The non-unit-codes can be ignored and are purely used to space out the unit codes. (The “filler text” can even be map tile codes, which can be helpful for visually inspecting the map file to see where the units will be placed on the map.)
  – “END” on a line by itself indicates the end of the unit layout grid.
Example Map File

Here is an example of a file that follows the TA map file format:

```
10 9
TERRAIN
 .. TRUE TRUE sprites.png 5 3
WW FALSE FALSE sprites.png 7 2
START
 ....................
 ..WWWWWW....WWWWWW..
 ..WW.........WW..
 ..WW....WWW....WW..
 ..WW............WW..
 ..WW..WWWWWWWW..WW..
 ..WW............WW..
 ....WWWW....WWWW....
 ....................
END
UNITS
aa TRUE
bb FALSE
START
 #######aaaaaaaaaaaa###
 #######aaaaaaaaaaaa###
 #######aaaaaaaaaaaa###
 #######aaaaaaaaaaaa###
 #######aaaaaaaaaaaa###
 #######bbbbbbbbbbbb###
END
```

Handing In

Each week, you should hand in the entire directory tree for your project, including code you handed in last week. You should also include a README file that describes how to verify each requirement, and an INSTRUCTIONS file that describes how to play your game, as specified in the Global Requirements. If you’re using Eclipse, execute `cs195n_handin tac1`, `cs195n_handin tac2`, or `cs195n_handin tac3` from the root directory of your Eclipse project. If you’re not using Eclipse, the directory you hand in should include an ant build script that compiles your source code into a jar file.

For this project, it is important to hand in map files that can be used to run your game. If you do not include any map files, the TAs will use map files written in the TA format described above.