Super Space Escape

May 10

2011

This game is of the Arcade RPG style. We divided our group into two subgroups. One created the tile engine along with our own map editor application. The other group focused on the game engine itself along with other game mechanics.

Wintendo

Member Contributions

Subgroup 1

- John Keech
  - GUI of map editor
  - Draw and erase codes
  - Implemented keyboard movement
  - Sprite effects
  - In game menus(inventory, game over, etc)
- Michael Perkins
  - Graphics
  - Dealt with backgrounds and foregrounds
  - Level transitions
  - Helped with erase code
- Zachary Pollock
  - Level design for game
  - Parser for level designs to transfer into game
  - AI polishing

Subgroup 2

- Bryant Poffenberger
  - Sprite handling
  - Collision interaction
  - AI of incoming enemies
  - General gameplay engine
- Matt Nidoh
  - User interface
  - I/O handling with movement and shooting
  - Menu and graphics display
- Stephen Dawkins
  - Character and object sprite sheets
  - Creating Character and weapons classes
  - Report
- Nathan Kreitz
  - Enemy sprite sheets
  - Creating Character and items classes
  - Developed scoring system

Subgroup 1

MapEditor Overview

The map editor was created to create a simple interface for designers to create rather complex levels with ease. This program allows the designer to use drag and drop brushing, like in a paint program, rather than having the encode each square by number. In this way, the developer can also preview their work and adjust while working on the design, rather than have to code and hope for the best. It provides easy access to edit all layers of the game and to view any layer combination at once to gain maximum usefulness.

The key part of Map Editor is file creation. When a new file is made from the newForm interface, a default map is created, saved, and loaded into the window. This map has no object, foreground, or collision elements by default and provides a starting place for making a new map.

The next important part is tile selection. When the user clicks on a tile in the palate on the left, that tile is set active, displayed in the active tile box above the palate, and any clicks on the map will result in this tile being set to the clicked position on the selected layer. If the eraser is selected however, the tile at the clicked position on the selected layer is set to clear and the graphics are reloaded to refresh the pixels to what was behind the erased layer. This brings us to the functionality of the main view window.

Whenever a click is made within the large map picture box, if a file is loaded for editing, the currently selected tile is calculated, the file is updated, and the new graphics are pushed out to the view pane. If a tile is erased, the specified part of the selected layer is cleared and all layers are redrawn in order to erase artifacts.

A major functionality that was added was arrow key support. The display is set to appropriate dimensions so that the user sees the same amount of the map that will be visible on the phone. This view can be moved, however, by using the arrow keys. In this manner, all parts of the map can be edited, and the user can even edit while scrolling across the map. Editing this way allows the user to get a good feel for what their map will look and feel like in-game, without losing functionality. The one drawback to this method is that the foreground cannot be viewed in its entirety in the editor.

In order to make clouds and other foreground objects move faster than the player, the map for the foreground is twice as large as that of the normal map. Because the map is edited as a single bitmap, it is not possible to have the foreground move faster than the background in the editor. To account for this, several premade foreground types were made. The user first selects what object they want in the foreground, and then selects the style that it is to be applied. This style is consistent for the entire map, so it is not necessary to view all of the foreground when editing.

Another critical feature that was added was the ability to hide and show different layers. Because layers cover each other up, the designer may wish to view only one or a few layers, rather than all at once. The editor essentially redraws the entire map when layer views are changed so that only the desired layers are shown. The hidden textures are still stored so that when the layers are brought back into view, the map is redrawn to include them.

The final important part of the editor is saving. When save is clicked, the user is prompted to name their file and the directory it is to be saved to. The editor then applies the changes that have been made to a text file, with each level stored in its own matrix in the file. These files can be reopened in the editor for future adjustments and can be resaved without harm to the data. The windows operating system takes care of the actual saving, allowing the user to save their .lvl file in the directory of their choice.

The map editor allows the user to easily create levels at all parts of the design process by taking the tedious task of editing text files and making into something akin to a drawing program. Users can add and erase textures to their hearts content on a map that is the size they choose and the editor takes care of file creation, editing, and storage so to user doesn't have to.

Form1.cs

This file is the key file that runs the entire project. Within the project, the Form1 is broken down into two parts: the graphical interface and the code backbone. The graphical interface is fairly simple, as it is comprised of simple picture boxes and buttons that are referenced in the code of Form1. Events are set up to call different functions based on buttons pressed.

Variables

-*int TILE_SIZE :* The size in pixels of the tiles

-*TileEngine engine :* The TileEngine object that contains the map data

-*TileLoader tileLoader :* The TileLoader object that creates and edits map data

-*int selected tile :* Reference of the number of the tile currently selected from the palate

-*int mapX, mapY :* The current coordinates of the map bitmap on the main picture box

-*bool isEditing :* Value that states if a map is loaded and in the state to be edited

-*bool erasing :* Determines if the eraser has been selected

-*bool editSinceLastSave :* Determines if data has changed since the map was loaded.

Functions

redrawMap( )
This function clears the current bitmap on the main picture box and redraws each visible layer to it in order.

clearFore( )
This function clears the foreground data and graphics in preparation for new data

drawFore( )

This function creates data for the foreground based on the *ForeType* and then draws it to the screen if the foreground is visible

erase(object sender, EventArgs e)
This function cleared the data from the selected position on the selected layer and puts the eraser image in the selected tile picture box

addTile(int X, int Y)
This function adds a tile to the map. It sets *editSinceLastSave* to true and calculates the position selected in relation to the map. The layer then has the selected tile added at the calculated location. If the eraser has been selected, the calculated tile is set to -1 and the graphics are removed from the layer at the location. Otherwise, the texture layer has the data added at the location in the file using and the map is updated on the GUI to reflect it using *TileLoader.addToMapLayer( ).* At the end, the whole map is drawn so that layering effects work using *redrawMap( ).*

getEditLayer( )
This Function returns the LayerType of the layer that is currently selected.

getForeType( )
This Function returns the foreground style that is currently selected.

MainForm_Load(object sender, EventArgs e)
This function initializes *tileLoader* and *engine,* and calls the method populateTiles( )

populateTiles(LayerType layer)

This function sets the initial bitmap to the palate image box.

createNewFile( )
This function calls newForm GUI to load, then, if a file is to be created, calls eninge.loadLevel( ), redraws the map, and enables editing.

loadFile( )

This function calls the engine.loadLevel( ) method to open an existing file, then redraws the map and enables editing.

SelectLoadFile(string initialDirectory)

This function brings up a prompt for the user to input the desired file to load.

saveFile( )

This function transitions through the various levels of the map, writing them to disk. It starts by writing the title and the map dimensions, and then it transcribes the background, collision, foreground, and objects, in that respective order. After the dimensions, each block is separated by a blank line to indicate the end of the layer, so the parser knows when to stop. It then marks that the file has not been edited past the last save and checks for thrown errors.

newForm.cs

This is the form used for creating a new document. It takes in the name of the map to be created, along with the dimensions of the map in tiles. Hitting 'Create New Map' makes the program check the bounds to make sure they are at least 25x15, the minimum to run the map on a phone. If it checks out, a default map is made with the name given, and all tiles on the background are set to a specific tile, in this case 72. The tiles on all other layers are set to -1. The GUI then closes, restoring focus to the map editor.

Variables

-int width, height : The width and height of the new map, in tiles

-string txtname : The filename for the new map ('.lvl')

Functions

createButton_Click(object sender, EventArgs e)

This function attempts to create a file with the string in the name textbox. If it is successful, and the minimum and maximum requirements are met, the new file is opened and the default values are written for the new map. The program writes the name on the first line, the dimensions on the second, and then skips a line. It then enters the matrix values for background, collision, foreground, and object, in order, separated by blank lines. The function then closes the file for future saving and closes the GUI.

cancelButton_Click( )

This function closes the newForm GUI without taking action.

Tile.cs

The Tile class holds basic information about a single tile on the map. It has an int reference to a 32x32 pixel square on a texture map. The program uses this to determine where to pull pictures from when putting together a rendering of the map, as a Tile object can be used to store the texture of any layer of the map.

Variables:

-*int texture :* A reference to the place on the source map that contains the texture for this tile.

-*bool collision :* A reference to the collision status of the tile on the map. Collision set to true will show up red on the map editor and will not allow the player or monsters to cross the tile in-game.

Functions:

Tile(int tex, bool col)
This function creates a Tile object with the *texture* set to *tex* and *collision* set to the value *col*.

setTexture(int tex)

This function sets the Tile *texture* reference to the value *tex*.

getTexture( )

This function returns the int value of the Tile's *texture* value .

setCollision(bool col)
This function sets the *collision* to the value of *col*.

hasCollision( )

This function returns the value of *collision*.

TileLayer.cs

The TileLayer class is what holds together the collection of Tile objects for each of the background, foreground, objects, and collision layers. Its functions allow the program to retrieve the matrix dimensions and to edit the matrix of Tile objects by assigning new Tile objects.

Variables:

-*Tile[,] layer :* Matrix of Tile objects
-*int width :* value of the width of the Tile matrix *layer*.
-*int height :* value of the height of the Tile matrix *layer*.

Functions:

TileLayer(int w, int h)

This function creates a TileLayer object with a *layer* that is width *w* and height *h*. Sets *width* to value *w* and *height* to value *h*.

setTile(Tile tile, int x, int y)
This function first checks to make sure that the Tile object at [x,y] is within the bounds of the *layer* matrix. If it is, the Tile object at [x,y] is set to be object *tile*.

getTile(int x, int y)

This function first checks to make sure that [x,y] is within the bounds of the *layer* matrix. If it is, it returns a reference to the Tile object at [x,y].

getWidth( )
This function returns the value of *width*.

getHeight( )

This function returns the value of *height*.

TileLoader.cs

The TileSet class stores the bitmaps of the individual Tile objects for a layer.

Variables

-*int numTiles :* the number of tile bitmaps in the TileSet object.
-*Bitmap allTiles :* The unaltered bitmap containing all textures for the layer.

-Bitmap [] tiles : The list of all tile bitmaps from *allTiles*.

<u>Functions</u>

<u>TileSet( )</u>
This function creates a default empty TileSet Object.

The TileLoader Class cuts up the texture files according to the designated tile size and saves each tile into a corresponding TileSet object.

<u>Variables</u>
-int tileSize : The pixel width/height of an individual tile bitmap.
-Bitmap collision : The bitmap to represent active collision.
-Bitmap eraser: The bitmap for the eraser selection representation.
-Bitmap[] mapLayers : The list of complete layer bitmaps based on the data created so far.
-Bitmap[] tileLayers : The list of bitmaps to be used for the palate of each layer.
-TileSet[] tileSet : The list of the TileSet objects, one for each layer.

<u>Functions</u>

<u>TileLoader(int size)</u>
This function creates a TileLoader based on the input *size*. The function first sets the *tileSize* to value *size*. It then sets the bitmaps for *eraser* and *collision*, which are always set to the same thing unless the source image is modified. The *mapLayers, tileSet,* and *tileLayers* objects are initialized to hold values for the four layers. The texture bitmaps that correspond to *tileSet* are set and the function pushes the individual pieces of the texture map to the *tileSet* and function *convertTileSetsToBitmap( )* is called.

<u>convertTileSetsToBitmap( )</u>
This function traverses the tiles of the four layers and creates the bitmaps for the *tileLayers* list, which contains the image that will show up in that palate of the final GUI.

<u>getNumTiles(LayerType layer)</u>
This function returns the number of tiles that are in the *tileSet* at the designated layer.

<u>getTile(LayerType layer, int number)</u>
This function returns the bitmap image stored in the *tileSet* at the designated position of the designated layer.

resetMapLayers(int w, int h)
This function resets the bitmaps for the display map of each layer to a new bitmap of the designated width and height, based on *tileSize*w* and *tileSize*h*.

addToMapLayer(LayerType layer, int x, int y, int id)
This function overwrites the current pixels on the map of the designated *layer*. If it is collision, it sets the area to clear or opaque red. If it is another layer, it sets the area based on *id*.

getMapLayer(LayerType layer)
This function returns the bitmap of the specified *layer* of *mapLayers*.

getTileLayer(LayerType layer)
This function returns the bitmap of the palate *layer* in *tileLayers*.

TileEngine.cs

The TileEngine Class is primarily responsible for storing all the data for a certain map and keeping track of it as it changes. It holds a list of Map objects and is responsible for transcribing file data into a viewable map.

Variables

-*int numTiles :* the number of tile bitmaps in the TileSet object.
-*Bitmap allTiles :* The unaltered bitmap containing all textures for the layer.
-*Bitmap [] tiles :* The list of all tile bitmaps from *allTiles*.

Functions

TileEngine(int tileSize, TileLoader loader)

This function creates a TileEngine Object. The value of *TILE_SIZE* is set to *tileSize, levels* is initialized, *currentLevel* is set to -1, and *tileLoader* is set to *loader*.

loadLevel(string file)

This function first opens a stream to *file*.lvl . The name and dimensions are read in and applied and a new Map *newLevel* is created. The *tileLoader* is reset for new tiles and the file is read in. For each matrix level block in the file, a for loop traverses the data and creates Tile objects based on the data read for the corresponding spots on the map. All of the levels are add to *levels* and the current level is incremented. The stream is closed and the function exits.

getCurrentLevel( )

This function returns the current level.

getCurrentMap()
This function returns the current Map object being edited.

Map.cs

The Map Class holds the data for a certain map, specifically the TileLayer list for each layer. The file also contains the Enumerations LayerType and ForeType that are used to reference the various layers in almost all the functions in the program.

Variables
-*TileLayer[] map :* Contains the TileLayer objects for each layer on a map.

-*int width :* The width, in tiles, of the map.

-*int height :* The height, in tiles, of the map.

-*string title :* The title of the map.

-*ForeType fore :* Used in the foreground creation algorithm to govern foreground

appearance.

Functions
Map(string name, int w, int h)
Creates a map object, setting *title* to *name*, *width* to *w*, and *height* to *h*. The function then initializes the *map* elements based on *w* and *h*.

getLayer(LayerType type)
This function returns the TileLayer object stored in *map* that corresponds to *type*.

getName()

This function returns the value *title*.

getHeight()
This function returns the value *height*.

getWidth()
This function returns the value *width*.

AutoID

This program is technically outside of the Map Editor, but it is used to make rapid adjustments to the text files themselves, or to mod code if needed. This program essentially takes in any picture, the tile size in pixels, and a save destination, and labels squares with the tile size as their dimensions, each numbered how the game would access them. The program displays the picture and makes a copy in the save destination

Functions

SelectLoadFile(string initialDirectory)
This function opens the windows load prompt and returns the path to the file as a string.

SelectSaveFile(string initialDirectory)
This function opens the windows save prompt for the directory where the output is to be saved.

button2_Click(object sender, EventArgs e)
This function opens the original picture and saves it to a bitmap. The bitmap then has the string labels applied to it, spaced according to tile size. This image is displayed on the screen and saved the specified directory. If no directory is specified, no output image is made, save for the one displayed on the GUI

Subgroup 2

Super Space Escape GameEngine

Super Space Escape is composed of multiple, intricate components that all work together, much like a human body.

The all-encompassing "nerve center" of the program is the GameEngine class. The GameEngine class facilitates interaction between all the smaller components.

GameEngine provides:

-Initialize function to set-up all the smaller components and allocate memory

-Load content function loads up graphics, music, levels, and other various content

-Draw function to draw all the various objects on the screen every frame

-Update function to update all the objects on screen and all the game objects every frame

GameEngine is also responsible for keeping track of a GameState object.

GameState

The GameState object is a single object that contains a reference to ALL the various engines and components:

-Player (local player object)

-MonsterEngine (engine that keeps track of monsters)

-TileEngine (loads and handles all the levels/maps)

-BulletEngine (engine that keeps track of bullets and allows for creation of new bullets)

-CollisionEngine (engine keeps track of CollisionTokens and alerts the Token when a collision has been detected).

-EffectsEngine (engine that handles new effect requests and the activation of current requests)

-ObjectManager (manager that activates only on initial load. Adds Monsters, items, etc to the proper engines. Also keeps track of item locations).

Notes:

1. It should be noted that, for the engine, MOST of the data classes (like Player, Bullet, etc) have been kept separate from the sprite code. What this means is that GameEngine handles all sprite code for the object. GameEngine decides what to draw based on information from the Player, Bullet. HOWEVER, there is one important exception. Monster's keep track of their own Sprite object. GameEngine is still responsible for drawing this Monster sprite however.

GameState allows a HUGE amount of functionality to occur within in the game. If a sub-engine needs specific information from another sub-engine, acquiring said information is a very simple process of just getting the reference to the target sub-engine through GameState. This allows for lots of different types of communication to occur through GameState. For example:

MonsterEngine can tell how far a given monster is from a player by getting the reference to the Player class from GameState.

BulletEngine can grab information about a monster that a bullet is colliding with.

Etc.

MonsterEngine

The MonsterEngine API has an AddMonster() function to add a monster to the list of currently handled monsters. The primary user of this AddMonster() function is the ObjectManager class since the ObjectManager reads monsters that are on a map that has been loaded. More on this later.

The other important function in the MonsterEngine is the Update() function. This function is called in the GameEngine's Update(). This function facilitates the update of each individual monster that is currently active.

Each monster is checked to see if it has collided with anything. If it's a bullet, then the monster's health is damaged. If the monster's health reaches 0 or less, the monster is marked for deletion. If the monster collides with a collision area from the map, the monster's last position is loaded. What this means is that when a monster is updated and their new location is a collision tile, then the monster's position must be reverted so he isn't in a collision state anymore.

If the monster is not marked for deletion and the monster is visible on the screen, the monster position is updated. During this time, the monster also performs AI actions if they're needed. [AI INFO HERE]

BulletEngine

The Bullet Engine, as previously mentioned, keeps track of Bullet objects. Bullets allow damage to be done to Players and Characters. If you need to implement something that does damage, use Bullets! For example, the sword is implemented using a temporary, stationary Bullet object.

The main BulletEngine API is the fire() function. The fire() function takes an initial x and y position, a direction, the owner type (Player or Monster), and bullet type. This function creates a new bullet, set's the bullet's attributes (based on the bullet type). This allows for bullet attributes to be defined in BulletEngine alone. Bullets have an x,y and a velocity x,y. The velocity is added to the x,y every time the bullet is updated.

The other important function is the Update() function. Like the MonsterEngine, each bullet's position is updated each Update() call. It checks collisions for bullets as well.

Bullets are a great way to deal damage to a specific target. They should really be thought of as objects that deal damage to a specific area. When the bullet detects collision with an object, it deals damage to that object based on the bullet's power. This makes BulletEngine a powerful class since it can deal damage and access other objects from other sub-engines.

CollisionEngine

The CollisionEngine is one of the bigger engines. Each object that is non-static must have and keep track of it's own CollisionToken. You can register an objection with the engine by calling register_object(). It takes the x,y width,height, and the collision type of the object. The object returned is the CollisionToken that must be kept track of.

Objects (like Player, Bullet, Monsters) are all responsible for calling the update() function on their CollisionToken. This marks the object in the collision system as updated and requiring inspection.

In the CollisionEngine's Update function, it checks for two different types of collisions. It checks for collisions with map types and it checks for collisions with other CollisionTokens. If a collision is detected, the affected CollisionToken is alerted. In order to keep collision checking to a minimum, I created a "marked" list that updated CollisionTokens are added to when they're updated. CollisionTokens are only checked for collision when they're added in this marked list. The idea behind this is that collision is only checked on a "need to know basis"

In the newer version of CollisionEngine I added a much needed feature to the CollisionTokens. Each CollisionToken now stores a generic object type called "parent". This allows for other engines to find the exact object that they're colliding with an to act properly. For example:

As mentioned above, the bullet engine is responsible for dealing damage to monsters and players when one of the bullets collides with them. With the "parent" object, we can figure out if the "parent" had any kind of attack/defense bonuses applied to their attack that would effect damage.

Essentially, the “parent” variable gives us greater control AND more information about the CollisionToken and the interaction that is occurring.

EffectsEngine

The EffectsEngine handles requests for special effects (like sounds or explosions). You can request effects using either RequestSound(), RequestExplosion(), RequestRumble*(. The sound types and explosion types are defined in enums in EffectsEngine.cs. Content for these effects (sound or explosion) must be loaded using LoadSound or LoadExplosion.

EffectsEngine is very flexible and can easily be added to. Simply adding to the defined enums allows for expansion and the addition of new sound and explosion types.

During the development cycle, I was able to add around 3 new sounds in a couple of minutes. It was a matter of loading the sound and then adding it to the enum of sound types.

Effects engine can also handle vibration (force feedback). If a specific engine requests vibration, effects engine will tell the hardware to rumble for a given amount of time.

The main motivation behind the EffectsEngine was to allow other aspects of the GameEngine to have special effects without either A) defining the same special effect within different game engines B) Filling up GameEngine.cs with special effects code. All it takes for an engine to make an effect is to just call the proper effect Request function.

Once again, EffectsEngine Update() function plays or draws the sound/explosion/vibration.

ObjectManager

The ObjectManager load() function takes a TileLayer for input and parses the Tile to figure out what items and monsters are part of the map. In the load() function, if the tile is a tile that represents a monster spawn, the corresponding monster type is added to the monster engine.

The other ObjectManager API function is getItemAt(). getItemAt() checks the location passed to it and sees if the tile that it corresponds to has an item. If it does, the Player is given the item and the item is removed from the TileLayer.

Note that what makes ObjectManager different from the other subsystems of the GameEngine is that the ObjectManager doesn't require the call of an Update() function. The ObjectManager simply needs to be called just once when the tile layer is loaded in since it loads Enemies and Items based on the tile layer.

ObjectManager allows for flexibility in the GameEngine by allowing game data to essentially be stored in the tile code. The ObjectManager, in a way, acts as a parser of the tile engine layer. Based on the state of the tile layer, enemies and items are handled. This allows for a much simpler time in designing levels: we don't need files to keep track of the level AND the monsters AND where the items are placed. ObjectManager simply parses the tile layer and understands what's happening in the level.

Description and Review of Components

- **GameEngine.cs**
    - The main center piece of the program. It uploads sprites, sprite batches, and map tiles. All content is loaded through GameEngine.cs.
    - All other engines are called through this component then updated and drawn here as well.
- **Character.cs**
    - Constructors for both the player and enemy classes.
    - Both have x and y locations along with health, speed and attack. Health determines how many hits a character can take. Speed determines how fast a character can move across the screen. And attack determines the strength of a bullet or collision.
    - Enemy attributes are determined by class whether it is GRUNT(lowest), TROOPER, or BOSS(hardest).
    - Player attributes can be increased by Items defined in the item class. Items include attack bonuses, defense bonuses, and new weapons
- **Sprite.cs**
    - Creates, uploads, and updates the sprite batch to control all sprites displayed on the screen

- **ObjectManager.cs**
    - Uses a hash table to control objects as they are created, spawned and removed from the screen.

- **MonsterEngine.cs**
    - Creates and controls monster movement.
    - Movement is determined by AI engine describe later in the report

- **CollisionEngine.cs**
    - Creates “Collision Token” that have length and width attributes.
    - Checks distance between two tokens to determine a collision.

- **BulletEngine.cs**
    - Creates a bullet object and displays it on screen through a two dimensional vector.

    - Takes into account the direction of player and whether or not player or enemy is shooting.
    - Also deletes bullet when it runs of the screen.

- **EffectsEngine.cs**
    - Creates game effects such as explosion animation along with the sounds that go with it.

AI Engine

Design by Bryant Poffenberger

The challenge of making enemies in a simple arcade game is discovering a way to let the computer controlled monsters determine how to act based on what's going on in their environment. My implementation allows for different monsters who can have varying personalities and react different based on their environment and other stimuli.

Note: From here, I will refer to the AI unit as a "monster." The basic functions that each monster performs are:

2.Think
3.Act

While this may seem simple, they are both comprised of multiple parts. The monster is also aware of four different factors or stimuli from the environment:

1. Distance from the player
2. Distance from the closest bullet
3. Alignment with the player (whether or not the player is a straight shot away)
4. Health

Based on these factors, the monster can perform five different actions.

1. Flee (run from the player)
2. Align (align with the player for a straight shot)
3. Advance(move towards the player)
4. Evade (attempt to dodge a bullet)
5. Attack (fire or use the monster's weapon)

In the thinking stage, the monster gathers information about its environment and constructs a rating on a scale from 1-100 regarding the previously discussed factors.

The rating for each factor can be calculated by using the following formula:

y = (-100/max)*x + 100

Where max is the maximum value of the factor, x is the current factor, and y is the rating. Once ratings for all factors are calculated, the monster can choose what action to take.

The Decision Matrix is a two-dimensional 4x5 array that corresponds to a given monster type. The rows of the matrix represent the possible actions and the columns represent the given factors. The values of the matrix represent how much the monster lets the given factor affect its consideration of an action. The likelihood that the monster will choose a given action can be calculated by:

action[i] = f1M(i,1) + f2M(i,2) + ... + f5M(i,5)

Where action is an array where each elements represents the likelihood of the ith action, f is the rating of the given factor, and M(x,y) is the value in the Decision Matrix that corresponds to the x,y values. Once all these values are calculated, the action that is performed by the monster is equal to max(action)

Each AI unit thinks and acts every given time interval. In the final game, Team Wintendo decided that the AI should think and act upon it's decision every 2 seconds.

This is where the A* pathfinding comes in. Once the AI decides to complete an action, the act() functions calls the function that relates to what the AI wants to do. In the case that the AI wants to advance, idle, or flee, we need to use pathfinding.

If the AI wants to advance, we simply choose the player's tile as the target tile to move to.

If the AI wants to flee, we choose a target tile that is in the opposite direction of the player.

If the AI wants to idle, we just choose a target tile that is in the AI's general vicinity.

Once we know the target tile, we begin to employ the A* pathfinding technique.

The general gist of A* is that each tile is given a ranking. The ranking consists of the movement cost from the origin of the AI and the distance from the target position.

Once we add our initial tile to the “open list”. The open list is essentially a priority queue where the lowest value is at the top of the queue. The basic formula for calculating the A* is as follows:

```
while(1) {
    Tile x = open_list.pop(); //Grab the tile with the lowest value off the open_list

    if(x == tile_target) {

        return; //Found exit!

    }

    closed_list.push(x); //Add the closed list. Tiles we like.

    List<Tile> neighbors = x.GetNeighbors(); //get all neighbors. Leave out tiles with
collision


    foreach(neighbors) {

        if(neighbor.at(i) != open_list.Contains(i)) {

            open_list.push(i);

        }

    }

}
```

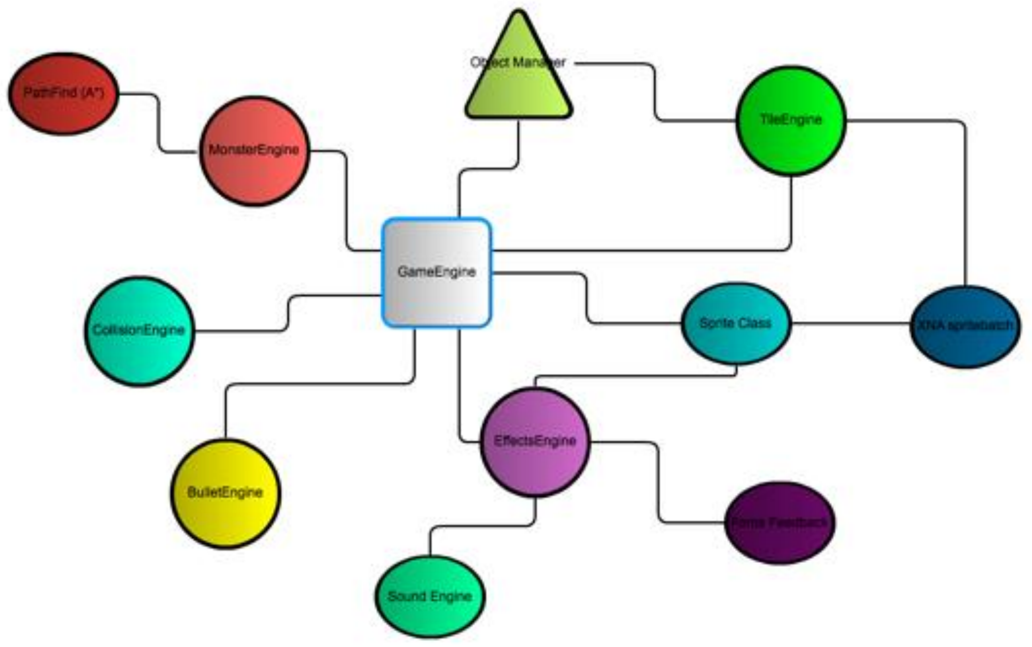
Following this algorithm allows the AI to find the shortest path between an origin tile and a destination tile.

A frustrating caveat working with XNA, though, is that there is no kind of priority queue implemented naturally. The workaround I used was taking a List of tiles and calling the List sort() function. The problem with this is that it has a very slow worst-case speed of O(n^2). In contrast, a normal priority queue is normally implemented with O(log n) time. This is MUCH faster.

As a result of this XNA problem, the AI engine is very easy to overload when large amounts of AI's need their path calculated at one time.

Software Architecture

It’s important to notice that our software architecture is based around the Game Engine. The Components around it were split in half between the tile engine/map editor and the game components that include but not limited to game play and the user interface. Although these two halves were approached concurrently, the testing of game play depended much on the completion of the tile engine and maps. Thus the tile engine was a main concern in the beginning stages to move forward. The game play component began with the class creation of characters with a basic outline of attributes. As work progressed, characters and items were given more concrete attributes as we became more aware of our capabilities. Eventually collision and animation was built into the engine. The menus and interface were less dependent on game play mechanics and the tile engine until it was fully integrated into the Game Engine. During the integration process with the Game Engine technical issues became a priority to make sure the game could run all together without crashing. Not until all issues were solved did we make efforts to polish the game. Polishing includes but not limited to sound, vibrations, extra effects to name a few.



Team Wintendo claims no ownership of the sounds, music, or graphics used in Super Space Escape. They are all used under fair use and creative commons license.