University of the West of England

Commercial Games Development

Charlie Thomas 16032542

Individual Report

Games Research and Development
UFCFCK-60-M

Contents

Workstream 1. 3
Previous iterations 3
Editor Application: 4
CTNote 4
CTTrack 4
CTNoteSaveData 4
CTNoteSaveDataSO 4
CTTrackSaveDataSO 5
CTJSONUtility 5
CTJSONHelper 5
CTIOUtility 5
CTTrackEditor 7
Level Editor Application “EditorApplicationReWrite” scene: 7
CameraControl 7
Cyberus 8
Workstream 3. 8
The Data Engine 9
CTTurnData 9
CTModifiers 10
CTSeed 10
CTChange 11
CTCost 12
CTResourceTotals 12
Disaster 12
DataSheet 12
CTPolicyContainer 12
CTPolicyCard 12
PolicyGen 12
GameManager 13
Data 13
CGD : CGS : Personal Development 17

CGD : CGS : BigStack Individual Report

Workstream 1.

Previous iterations

The first iteration of the WS1 Level Editor was developed as a Unity Editor tool as a proof of concept.

This first iteration used editor extension methods to set up and configure notes in world-space. These notes were prefabs set up as different note difficulties that could be dragged into the scene and would snap to the nearest node position using a RoundToV3 method that would round the reference parameter to the nearest 1, 3 or 5, floor Y at Zero and round Z to the nearest BPM step gap. These extension methods could be accessed through UnityEditor->Edit->Snap Selected Notes.

This Editor tool version had a TrackGizmos class that would generate visualisation as shown in Figure1. The length of these Gizmos would be determined by setting track_length and track_bpm.

As notes are added to the scene, they added themselves to the List of notes on the WriteFile class. When writing the file in this system, a “WriteFile” script would loop through each note in this List, evaluate the X and Z coordinates of the GameObject associated with it in world-space and adjust these to 2D coordinates.

These notes are then written to a .TXT file with the same layout as defined with the notes’ world space positions.

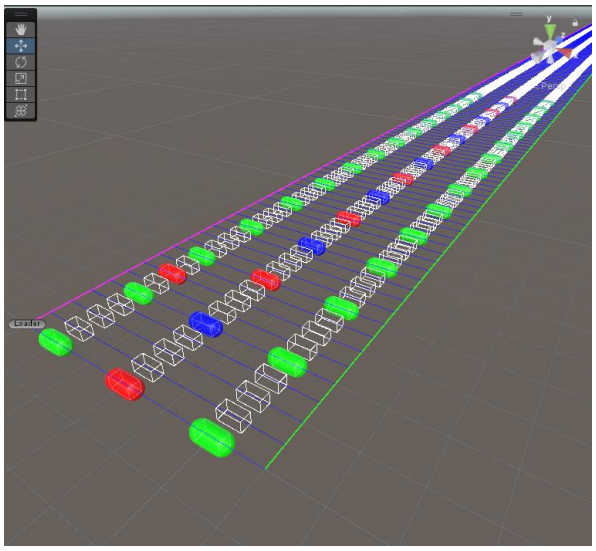


Figure 1: Gizmos track layout with notes

```
1 reference
private void CreateTextFile(int _total_beats)
{
    TimeSpan t = DateTime.UtcNow - new DateTime(1997, 5, 1);
    int t_since_me = (int)t.TotalSeconds;

    // Create text file in directory "/Track_Files/"
    string txt_document_name = Application.streamingAssetsPath +
        "/Track_Files/" +
        track_name +
        " " +
        t_since_me +
        ".txt";

    // Only create file if it does not exist
    if (!File.Exists(txt_document_name))
    {
        Debug.Log("Generating text file");

        // For every beat
        var beat = new StringBuilder();
        for (int b = 0; b < _total_beats; b++)
        {
            // For each note per beat
            var notes = new StringBuilder();
            for (int n = 0; n < 3; n++)
            {
                notes.Append(data[b, n].ToString()); // Add note to notes for beat n
            }

            beat.AppendLine(notes.ToString()); // Add notes for beat n to new line
        }

        File.WriteAllText(txt_document_name, beat.ToString());
    }
}
```

Figure 1: WriteFile: CreateTextFile method

https://drive.google.com/file/d/1WNCTYib7xcxmQFB8OfZeJXVX8kBCccG6/view?usp=sharing

Editor Application:

CTNote is the base Note class used to define data properties of a note. These properties are informed by the existing Cyberus note data:

- int *ID*
- Vector2 *LanePos*
- bool *IsActive*
- bool *IsEndOfTrack*
- CTNoteDifficulty *Difficulty*

It contains an initialise function, which takes parameters for each of the stored values. These parameter values are then set to the local values.

CTInteractableNote is a MonoBehaviour inheritance class used to define the behaviour of the interactable note objects in the *LevelEditor* scene. It has references to materials used to denote note status visually, as well as a parent gameobject in the scene to keep the hierarchy tidy. The methods of note are:
OnClick() defines behaviour on right or left click. In simple terms, this sets the data controller object "*CTTrackEditor*" object's associated note's Difficulty property appropriately based on the type of note selected in the editor. Right click will always set this *Difficulty* to "*None*", and Left click will set the difficulty to the selected difficulty type. This is done through a reference to the *CTTrackEditor* singleton object's *CTIOUtility* member, where the stored track's dictionary_notes at index of current note *ID* is assigned this difficulty property.

UpdateMaterials() UpdateMaterials is a switch statement void function that updates the material of an interactable note appropriately based on its current Difficulty property in *CTIOUtility*.

CTTrack is a class that defines a track. It has member variables:

- SerializableDictionary<int, CTNote> *dictionary_notes*
- int *track_bpm*
- int *track_length*

It has methods:
Initialise() takes bpm and track length values as ints, and sets the member variables to these values. It initialises the *dictionary_notes* and calls *GenerateNotes*.
GenerateNotes() is used to generate the notes in data using the bpm and track length values. A "*total* beats" value is calculated using bmp * track length (in seconds), and a nested for loop is used between zero and the total beats value to generate new *CTNotes* for each of the three lanes for each beat, initialise them with values and add them to *dictionary_notes* with a Key of *note.ID* and a value of *CTNote*. If the note is one of the final three notes, the *CTNote* property *IsEndOfTrack* is initialised to true.

CTNoteSaveData contains the properties of a *CTNote* that should be saved. These values are copied to this class prior to saving.

CTNoteSaveDataSO is used for dirty serialisation of the note data into the unity format *ScriptableObject*.

CTTrackSaveDataSO is used for dirty serialisation of the note data into the unity format *ScriptableObject*.

CTJSONUtility is a static class used to perform *JSON* read/write operations. It contains two public static methods: *SaveToJSONAtPath* and *ReadFromJSONAtPath*. *SaveToJSONAtPath* takes a list of objects of type *T*, a file path and a file name, and saves the list as a *JSON* file at the specified location. *ReadFromJSONAtPath* takes a file path and file name, reads the *JSON* file at that location and returns a list of objects of type T.

The class also contains two private helper methods: *WriteFileToPath* and *ReadJSONFromPath*. The *WriteFileToPath* method takes a file path and content string, creates a new file at the specified location and writes the content string to the file. *ReadJSONFromPath* takes a file path and name, reads the content of the file and returns it as a string. The class also logs errors to the Unity console through *Debug*.*Log*.

CTJSONHelper is a helper class for working with *JSON* data. It contains two static methods: *FromJson* and *ToJson*. *FromJson* deserialises a *JSON* string into an array of objects of type *T*. *ToJson* serialises an array of objects of type *T* into a *JSON* string. It also contains a nested templated class "*Wrapper*", which is used to hold the array of objects during serialisation/deserialisation.

CTIOUtility is a utility class for managing input/output (IO) operations such as saving and loading data from files. It has member variables:

- string *track_name*
- CTTrack *track*
- CTTrackSaveDataSO *track_save_data*
- List<CTNoteSaveData> *loaded_track_note_data*
- string *track_file_path*
- string *note_file_path*
- string *file_name*
- string *persistent_path*

***Initialise*( )** is a void function taking parameters for track name, bpm and track length. It uses these values to set up the data structures of the *CTTrack* track member variable as well as the save data types. Furthermore, it sets the member variables' path strings based on the track name and calls the method *CreateDefaultFolders* to set up paths based on these strings.

SaveTrackData( ) is used to save the current track data as persistent data. It calls the methods: *CreateDefaultFolders*, *SaveTrackToScriptableObject*, and *SaveNoteDataToJSON* twice; once to the local StreamingAssets folder and again to the *AppData* persistent path.

SaveTrackToScriptableObject( ) is used to save the track as a *UnityEngine*.*ScriptableObject*. It does this by creating a local *CTTrackSaveDataSO* object, writing data to it through a foreach loop over the *track*.*dictionary_notes* container and calling the *SaveAsset* method on this local track data object.

SaveNoteDataToJSON( ) is used to save the track as a *JSON* file. It does this by creating a list of *CTNoteSaveData* objects, and writing each *CTNote's* data in track.dictionary_notes to this list by method of creating a new *CTNoteSaveData* object and writing the current *CTNote's* data to it.

An alternative and likely superior method would be to create a custom constructor for *CTNoteSaveData* that takes a *CTNote* as a parameter, and assign values this way. This would reduce

code duplication from manually assigning values every time this copy functionality is required. This also applies to *SaveTrackToScriptableObject*.

LoadNoteDataFromJSON( ) is used to load note data from a *JSON* file at a specified path. It takes two string parameters for file path and file name, and uses these to locate the requested file. It assigns *loaded_track_note_data* the value returned from *CTJSONUtility.ReadFromJSONAtPath<CTNoteSaveData>*(*PATH*, *NAME*) and then calls the *LoadNoteDataIntoTrack* method.

LoadNoteDataIntoTrack( ) is used to translate loaded data into the current data. It clears the current track of notes and for each *CTNoteSaveData* object in *loaded_track_note_data* creates a new *CTNote* and initialises it with stored values. These new objects are then added to the track data.

CreateDefaultFolders( ) is used to ensure that folders used for writing data exist prior to attempting to write to them. It makes use of the *CreateFolder* method, doing this for local *Resources*, *StreamingAssets* and *AppData* persistent path folders.

CreateFolder( ) takes two strings as parameters and uses these as path and folder names to create new folders. It first checks if the requested folder already exists, and if so returns. Otherwise, it makes use of the *AssetDatabase*.*CreateFolder* method, passing the parameter path and name strings.

RemoveFolder( ) is unused. It takes a string parameter for path, which is used with the *FileUtil*.*DeleteFileOrDirectory* method to remove the folder from the specified path. This function would see use if further development were done on this project for asset cleanup.

CreateAsset( ) is a templated type function requiring the templated type to be of type *UnityEngine*.*ScriptableObject*. It takes parameters of path and file name and creates a "*full_path*" string from these parameters. Prior to attempting to save this asset, it first performs a load check using the templated *LoadAsset* method looking for an asset of this type with the same name at the specified path. If this asset is null, the asset is saved.

LoadAsset( ) is a templated type function requiring the templated type to be of type *UnityEngine*.*ScriptableObject*. It takes parameters of path and file name and creates a "*full_path*" string from these parameters. It then makes use of the *AssetDatabase.LoadAssetAtPath<_T>* method, passing *full_path* as a parameter, returning the asset at this path.

SaveAsset( ) is a function taking a parameter of type *UnityEngine*.*Object*. The function *EditorUtility*.*SetDirty* is called on this asset, flagging it to the Unity engine as "dirty". This tells unity that the asset has been modified and needs to be included in the save when the project is saved. The methods *AssetDatabase*.*SaveAssets* and *AssetDatabase*.*Refresh* are then called.

RemoveAsset( ) is unused. It takes a string parameter for path and asset name, which are used with the *FileUtil*.*DeleteFileOrDirectory* method to remove the folder from the specified path. This function would see use if further development were done on this project for asset cleanup.

CTTrackEditor uses the singleton pattern. It has member variables:

- *bttn_print_beatmap*: Serialised reference to Canvas UI
- *bttn_diff_one*: Serialised reference to Canvas UI
- *bttn_diff_two*: Serialised reference to Canvas UI
- *bttn_diff_three*: Serialised reference to Canvas UI
- *note_interactable_object*: Serialised reference to interactable note prefab
- *note_list*: A list of associated interactable note GameObjects
- *cameraControl*: Serialised reference to a *GameObject* with component *CameraControl*
- *CTNoteDifficulty*: An enum for the current difficulty to paint to notes
- *AudioPbUI*: Sam's audio work

Initialise( ) takes parameters for track name, bpm and length. If there are any existing notes in the track, they are cleared. Using the *CTIOUtility*, it sets the selected difficulty and adds listeners to buttons. It also instantiates note prefabs for the track and checks for existing save data, loading it if it exists.

InstantiateInteractableNotePrefabs( ) creates intractable note objects in a 3xN grid, where N is the length of the track. It calculates the position of each note as a product of it's X and Z coordinates in an absolute 2D grid map of the track and multiplies these values by a constant offset. New *CTInteractableNote* objects are instantiated at these coordinates using a *ScriptableObject* prefab as a template. It then assigns a unique ID to each note object based on its position in the grid and adds it to a list of note objects.
LoadTrack( ) calls the *LoadNoteDataFromJSON* method in *CTIOUtility*.
SaveTrack( ) calls the *SaveTrackData* method in *CTIOUtility*.
CheckKeyboardShortcuts( ) is a function that listens for keyboard shortcuts for track saving, clearing and note difficulty selection.
UpdateSelectedDifficulty( ) is a void switch statement function that takes a parameter int and sets the selected note draw difficulty to that value using Enums.

ClearTrack( ) loops through each stored note and clears the difficulty by setting them all to *CTNoteDifficulty*.*None*.

Level Editor Application "EditorApplicationReWrite" scene:
The majority of this scene is code based and generated on *Start*( ). The notes in the scene are generated based on the selected track length and beats per minute. The camera position is set as a *zero->one* float relative to two positions, where the origin is a constant and the destination is set on *Start*( ) based on the number of notes generated such that it can travel as far as the notes do.

CameraControl is a class responsible for handling the behaviour of the main camera in the editor scene. It has methods for keyboard control, scroll control, and automatic control. Keyboard control listens for W or S key inputs and moves the camera forwards or backwards in the scene between *Camera Pos 0* and *Camera Pos 1*. Scroll control does much the same thing, but has an inertia system whereby the camera will slowly decelerate much like scrolling on a touchscreen phone. This is achieved by having a *mwheel_scroll_momentum* float that stores the camera's current "*momentum*" value and multiplying this value by 0.99 every update to decelerate it. If the user clicks while the camera has momentum, it completely arrests the momentum by setting it to *zero*.

Otherwise, during track playback, the camera position is set on a scale of *zero -> one* between *Camera Pos 0* and *Camera Pos 1* relative to the track playback slider value between zero and one.

Cyberus:

Integration with Cerberus makes use of existing functionality from the LevelEditor application, including: *CTJSONHelper*, *CTJSONUtility* and *CTNoteDifficulty*. There is one new script added to load information from persistent path:

PersistentDataLoader( ) is used to load JSON data saved using the Level Editor application. It creates an object of the *HH*.*Track* type from a given JSON string at path.

It should be noted that I am unsure if this was used in the final implementation of the Cyberus integration, and I wrote this as proof of concept that the file sharing through persistent path would work correctly.

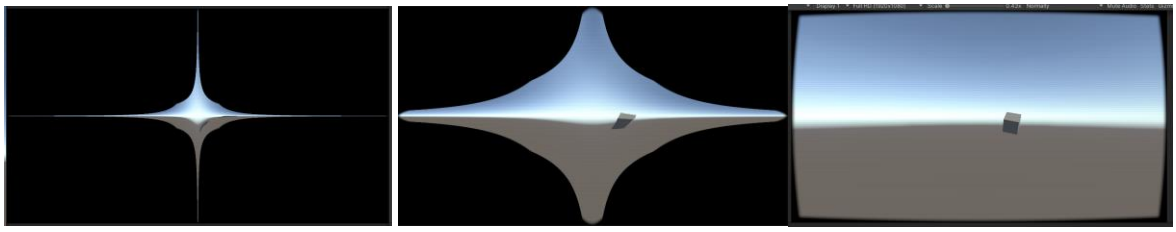
3rd Party Utilities: This project makes use of a *SerializableDictionary* class made by "**azixMcAze**". This is not a utility written by any member of CGS BigStack. It is used in some cases where dictionary serialisation is desirable, as a replacement for the standard Dictionary class, because Unity does not natively support serialisation of dictionaries. This utility is available from: https://github.com/azixMcAze/Unity-SerializableDictionary

Workstream 3.

Previous iterations of the Data engine were poorly conceived. They attempted to manage multiple large data sets containing all information holistically, while preserving timeline continuity. This approach was unmanageable as a data engine, debugging was borderline impossible and propagation of data throughout the timeline was computationally expensive. Removal of changes or events triggered by the player was equally undefined. Attempting to manage such a large set of data with so many intricate moving parts in this way was a mistake.

I wrote a CRT Shader (following a tutorial) and implemented it as a renderer feature for use on virtual Cameras for the game city view and technology tree screen views. Unfortunately, this was done as more of a personal interest side development bit of work and I am not super familiar with writing shaders in general, let alone for URP. As it turned out this shader, while it provides the desired effect, does not preserve URP postprocessing (e.g.: glow effects). As there are several elements in both scenes that use these, this method was scrapped in favour of other techniques that produced the desired effect while preserving URP postprocessing.

It would have been something I would have liked to look into had I started work on it sooner, however it was a feature I suggested late into the development cycle of WS3 and there were other features that were more pressing. One of the potential further uses for this effect would have been to use the curvature property of the shader to simulate a monitor "turning on" effect, by raising the curvature from Zero to the desired value quickly:



This shader, along with the ScriptableRenderFeature and ScriptableRenderPass scripts for it can be found in the project files at Assets>Shaders>CRT Effect.

Charlie Thomas 16032542 CGS Individual report

UML Diagram:
https://drive.google.com/file/d/1cQ_bXUNN2kDJQdGN_nVzEKuqcuM_5Ess/view?usp=sharing

The Data Engine functions on a tracked changes model, with a prime initial timeline state (*CTTurnData*) that has defined starting properties. This "seed" turn is used to generate further turn data on request by means of calculations that factor in "changes". A function can be called in the *GameManager*, returning a *CTTurnData* containing all information required to define the turn.

CTTurnData is a class used to store turn data. This includes what turn it is, whether it is considered a "failed turn", faction distribution, owned technologies, applied and revoked policies, cost modifier totals, money, science, food and population count.

Constructors: Has a copy constructor taking a *CTTurnData* parameter and creates deep copies of non-primitive data types.

Member variables:

- *uint turn:* stores the turn the CTTurnData is associated with
- *bool failed_turn:* is a getter that returns Population <= 0
- *Vector4 faction_distribution:* stores the faction distribution
- *Dictionary<CTTechnologies, bool> technologies:* stores technologies with the Boolean value denoting their status.
- *List<CTPolicyCard> applied_policies:* stores applied policies
- *List<CTPolicyCard> revoked_policies:* stores revoked policies
- *Vector4 cost_modifier_totals:*

Resources: Resources are stored as private float types with public int type getters. This is done so that return types are always int, and float multipliers can be applied to the private type to preserve accuracy.
private float data_money stores a value for the "money" resource.
public int Money has a getter that returns data_money cast to an int. The setter checks that the asserted value is not equal to the current value, and returns if so. If the asserted value is less than zero, set data_money to zero. Finally sets data_money to the asserted value.

public float data_science stores a value for the "science" resource.
public int Science has a getter that returns *data_science* cast to an int.The setter checks that the asserted value is not equal to the current value, and returns if so. If the asserted value is less than zero, set *data_science* to zero. Finally sets *data_science* to the asserted value.

public float data_food stores a value for the "food" resource.
public int Food has a getter that returns *data_food* cast to an int.The setter checks that the asserted value is not equal to the current value, and returns if so. If the asserted value is less than zero, sets *data_food* to zero. Finally sets *data_food* to the asserted value.

public float data_population stores a value for the "*population*" resource.
public int Population has a getter that returns *data_population* cast to an int.The setter checks that the asserted value is not equal to the current value, and returns if so. If the asserted value is less than zero, set data_population to zero and set *faction_distribution* to all zeroes. Finally sets *data_population* to the asserted value.

Getters: there are several getters that do not have stored values, and instead calculate the return type.

- *float AssignedPopulation:* returns total of *Workers*, *Scientists*, *Farmers*, *Planners*.
- *float UnassignedPopulation:* returns *Population - AssignedPopulation*.
- *int Workers:* returns *Population* * worker faction distribution ratio.
- *int Scientists:* returns *Scientists* * scientist faction distribution ratio.
- *int Farmers:* returns *Farmers* * farmer faction distribution ratio.
- *int Planners:* returns *Planners* * planner faction distribution ratio.

Methods:
Initialise( ) takes int parameters for money, science, food and population and sets their corresponding data types with these values. Also initialises the technologies Dictionary's values as false.
Actions:
ApplyCosts() takes a *CTCost* object and an *Enum CTCostType* to determine what costs should be applied to the stored data and how.

- For upkeep costs, these costs are applied with faction net modifiers using the *GetFactionNetModifiers* method.
- For purchase costs, these costs are applied literally.
- For disaster costs, these costs are applied as a percentage of the resources modified by the safety factor.

ApplyTechnology() sets the input parameter technology as true in the technologies dictionary.
SetFactionDistribution() sets the faction distribution to the input parameter value.
ApplyBuffNerfs() applies buffs and nerfs based on active technologies. It loops through all technologies the faction has and applies the buffs/nerfs associated with them. It then loops through all set policies and revoked policies to find a list of active policies within the scope of the current turn and applies the buffs/nerfs associated with these policies if the requirements are met. These total buff/nerf values are added to cost_modifier_totals.
BuffNerf() is a switch function that takes buff/nerf types and degrees and calls other functions based on what type of buff/nerf is passed.
ApplyFlatValue() is a void switch function that takes parameters of *CTModifierType* and a float for the value. Depending on the *CTModifierType*, the value is added to the appropriate resource type as a one time flat value bonus.
ApplyCTCostModifier() is a void switch function that takes parameters of *CTModifierType*, a float for the value and a reference to a *Vector4* for modifier totals. Depending on the *CTModifierType*, the value is added to the referenced modifier totals at the appropriate index.

Utility: There are several utility getter functions, a method to scale the population starvation based on the deficit scale, and a method to swap resource values at a given ratio.

CTModifiers is a struct of floats that define modifiers to the base money, science, food, safety, awareness modifiers as well as flat gains for money and science per turn.

CTSeed is a static class used to deterministically generate "random" data in the game. It contains a variable "*gameSeed*" that is set to the *System.DateTime.Now.Ticks* value. It contains one function, *RandFromSeed* which takes *uint* "*_turn*" and *string* "*_label*" parameters. These are used to create a string "*local_seed*" through an interpolated string using the index and label parameters. *RandFromSeed* returns *System.Random(local_seed)*. This is useful, as given the same seed, index and label this function will always return the same value (deterministic).

CTChange is a base type used to define different types of changes that may be tracked in the data engine. The base class defines an int *Year*, as well as an abstract public void *ApplyChange*, requiring a *CTTurnData* reference as a parameter. Derived types of *CTChange* implement overrides of *ApplyChange* to meet their specific requirements:

ApplyDisaster : CTChange:

- Has a custom constructor, with the override taking a Disaster type and setting local member variables to the parameter values.
- ApplyChange calls the ApplyCosts method on the reference CTTurnData object, passing in a reference to the DataSheet base disaster impact cost multiplied with the local intensity value. It also passes an Enum for the CTCostType as CTCostType.Disaster.

PurchaseTechnology: CTChange:

- Has a custom constructor taking a parameter of *Enum CTTechnologies*. The local *CTTechnologies* "tech" variable is assigned this value.
- *ApplyChange* calls the *ApplyTechnology* method on the reference *CTTurnData* object, passing the local *CTTechnologies* enum type as a parameter.
- *ApplyChange* calls the *ApplyCosts* method on the reference *CTTurnData* object, passing in a reference to the *DataSheet* cost for the local *CTTechnologies Enum* type and an Enum for the *CTCostType* as *CTCostType*.*Purchase*.

SetPolicy : CTChange:

- Has a custom constructor taking a parameter of *CTPolicyCard* type. The local *CTPolicyCard* variable is assigned this value.
- *ApplyChange* references the list "*applied_policies*" member variable of the referenced *CTTurnData* object and adds the local *CTPolicyCard* variable to this list.
- *ApplyChange* also calls the *ApplyCosts* method on the reference *CTTurnData* object, passing in a reference to the local policy.cost variable, as well as an *Enum CTCostType* as *CTCostType*.*Purchase*.

RevokePolicy : CTChange:

- Has a custom constructor taking a parameter of *CTPolicyCard* type. The local *CTPolicyCard* variable is assigned this value.
- *ApplyChange* references the list "*revoked_policies*" member variable of the referenced *CTTurnData* object and adds the local *CTPolicyCard* variable to this list.

SetFactionDistribution : CTChange:

- Has a custom constructor taking parameters of: *float _workers, float _scientists, float _farmers, float _planners*. These values are added into a float *total_assigned* variable and checked against a greater than one boolean. This will throw an exception, as you may not have more than 100% of the population allocated at once. Passing this check, local variables for the worker, scientist, farmers and planner floats are assigned these parameter values.
- *ApplyChange* references the *SetFactionDistribution* method in the reference *CTTurnData* object, passing a new *Vector4* with local faction breakdown floats.
- *SetFactionDistribution* also contains operator overrides for: ==, !=, <=, >=, <, and > operators.

TrackAwareness : CTChange:

- Has an override constructor taking a float parameter of awareness, setting the local variable to this value.
- The *ApplyChanges* method sets the *Awareness* variable for a given reference *CTTurnData* object to the local value.

CTCost is a class used to apply resource costs to the timeline. Essentially it is a wrapper for four floats: money, science, food and population. These are the base "resources" that may have a cost associated with them; this class is used to define the production/upkeep values for each individual faction type, the cost of purchasing a technology, the cost of purchasing a policy, and the impact a disaster has on the player's resources and population.

It has two custom constructors, one taking floats for each of the local resource member variables, the other taking an existing *CTCost* type. Each of these constructors set the local member variables to the parameter values.

It also contains operator overrides for boolean operator comparisons between both *CTCost* types and *CTCost* type and *CTResourceTotals* type.

CTResourceTotals is a neutered version of *CTCost*, containing only member variables for money, science, food and population with no methods.

Disaster is a small data class containing information associated with a disaster. This includes an *Enum* for the *CTDisasters* type, an int for the year it takes place in for UI representation, a float for the intensity of the disaster and an int for the actual playable turn it occurs in.

DataSheet is a lookup table for all base values in the game, all technology costs and bonuses, all disaster impact values, strings to generate data driven policy card text, and enums. It also has methods to return values from these lookup tables.

CTPolicyContainer is a data class for *CTPolicyCard* types. For any given turn there are seven policy cards the player may select from, and there are forty turns. Seven *CTPolicyContainers* are used, each of which have an array of *CTPolicyCards* of size 40. On start, each of these *CTPolicyContainers* assigns each *CTPolicyCard* data on start using *PolicyGen*. Setters and Getters for the current policy based on the turn.

CTPolicyCard is a class that defines what it means to be a policy card. This includes: a unique ID, information text, an associated *CTCost* with purchasing the policy, a required faction distribution for the policy to take effect, what buffs and nerfs the policy contains, as well as what scale these buffs have.

PolicyGen is a static class used to generate policies. It contains methods to generate:

- Policy Cost (*CTCost*)
- Policy Requirements (pur turn faction distribution)
- Associated Buffs (*BuffsNerfsType*)
- Associated Nerfs (*BuffsNerfsType*)
- Degree to which these buffs and nerfs are scaled
- Policy title (string)
- Helper function to return number of buffs/nerfs
- Helper function to return a random faction distribution

To do this deterministically, local variables are used: uint "seed", double "scramble", string "ID".

Each of these methods generates deterministically using the *CTSeed*.*RandFromSeed* method. *PolicyGen* has a local uint "seed", which is used as the index to the RandFromSeed method. Each call within a function passes a label with an acronymized version of the function name appended to the local string "id".

GameManager: The *GameManager* is responsible for the IO between the UI and Data Engine and is implemented using a singleton pattern.

Member variables:

ApplyDisaster[ ] is a private array of *ApplyDisaster* types. It is used to store disaster information, with the array index corresponding to the turn the disaster takes place in.

List<CTChange>[ ] game_changes is a private array of lists of type *CTChange*. It is used to store the timeline change data generated through the *AIPlayFromTurn*( ) function. The array index corresponds with the turn the changes were made in.

List<CTChange>[ ] user_changes is a private array of lists of type *CTChange*. It is used to store the timeline change data whenever the player makes a change. The array index corresponds with the turn the changes were made in.

List<CTChange>[ ] awareness_changes is a private array of lists of type *CTChange*. It is used to store awareness change data generated as the player makes timeline changes.

CTTurnData initial_year is a private static readonly object of type *CTTurnData*. It is used as the initial "seed" of the game data and is the point from which the whole timeline propagates.

CTTurnData turn_data is a public object of type *CTTurnData*. It is used to store the information calculated on the current turn the player is accessing.

uint current_turn is a private *uint* used to store the turn the player is currently accessing.

Vector3 empty_turn_resource_expenditure is a private *Vector3* used to store data on the purchases the player has made in the current turn for the purpose of updating the UI with these values.

int stored_changes_in_turn is a private *int* used to store the number of changes the player has made in the currently accessed turn. This is used to inform awareness raises.

Unity Functions:

Awake( ) is a default Unity void function used to perform a check for multiple singleton instances, and will throw an error and will call the *Destroy*( ) function on itself if there already exists another singleton of the same type.

Start( ) is a default Unity void function used to set up the initial state of the game on scene load. This means calling the *Initialise*( ) function to create data structures and set initial values, calling the *Generate*( ) function on the *DisasterManager* singleton instance to create disasters in the timeline, calling *AIPlayFromTurn*(0) to populate *game_changes* with decisions, calling the UI setup functions: *UpdateResourceCounters*( ), *UpdateFactionDistributionPips*( ) and *UpdatePipsWithCurrentTurnData*( ); and setting up the tech tree by clearing all buffs and calling the *UpdateNodes*( ) method on the *TechTree*.

Setup:

Initialise( ) is a void function used to set up default values and data container structures.

Data**:**

GetYearData(uint _year) is a *CTYearData* return type function that makes up the core of the data engine. This function takes a uint year as a parameter and returns data about the state of the requested year. The function starts by creating a new *CTTurnData* object "ret" and initialises it with data from the initial_year using a deep copy constructor. The function then loops through all years from zero to the requested year and performs the following operations:

- Apply all changes in the list at the array index of loop "i" for *game_changes* using the override *ApplyChanges* method on each item.
- Apply all changes in the list at the array index of loop "i" for *user_changes* using the override *ApplyChanges* method on each item.
- Call *ApplyBuffNerfs*( ) method on "ret" to set up buff / nerf values for each loop (based on active technologies and policies in the scope of the turn)
- Grow the population at a rate defined on *DataSheet* if resource requirements are met

- Create *CTCost* modifiers based on the *GetFactionNetModifiers*( ) method return value on "ret"
- Apply net resource worth of each assigned population member multiplied by faction totals and modifiers.
- Decay the population at a rate defined on *DataSheet* if resource requirements are not met.
- Apply net resource production / consumption to resource data values
- Apply disaster costs to the timeline if there was a disaster in the currently evaluated turn

After looping through each turn to cumulatively calculate the changes to the timeline, return the *CTTurnData* object "ret". There is significance to the order of operations here. In applying user changes after game changes, we can override the game changes with user changes if there are conflicts over finite state systems such as faction distribution.

This setup of the data engine is advantageous because adding or removing types derived from *CTChange* to or from the lists can implicitly affect the timeline, eliminating the need to develop a separate complex system for this purpose.

GetTimelineData( ) functions in much the same way as *GetYearData*( ) except it returns a List of *CTTurnData* types. For each loop in this function, the data state at the end of each loop is added to a list. The purpose of this function is for systems such as the graph view to be able to access all forty turns worth of data at once, achieving this is forty loops as opposed to calling *GetYearData* forty times and adding the return data to a list. Performing this operation as a series of *GetYearData* operations would follow the Triangular Number formula: sum = n(n+1)/2, where n = total turns. For n = 40, we get 820 meaning 20.5 x fewer loops are required to get the same data.

Another way to achieve similar optimisation would be to write a "*GetNextTurnData*" method and use it recursively, passing in the current turn state to get the next.

AIPlayFromTurn(uint _turn) is a void method used to simulate the gameplay using an automated random decision making process. It takes a year to start generating from, and will continue to do so until it "fails" - meaning it ran out of population. To do this it goes through several steps for each turn between the requested generation point and the end of the game:

- Get the year data at the requested turn to generate from
- For each turn, Generate a random faction spread
- If a change of type *CTChange*:*SetFactionDistribution* already exists in this year, replace it
- Calculate a list of valid technology purchases by looping backwards through the tech tree and checking whether all prerequisite technologies are owned and that the technology is not already owned
- Randomly decide on a valid technology and purchase if it is affordable
- Before entering the next loop, validate that all player technology purchases are still valid with *CheckAllUserTechPurchasesValid*( )

If while in the loop the function loads the new year data and finds the Population to be zero, it will return and stop generating changes.

This function is called once on start, and after is called any time the player makes a change. From the turn the player makes the change, game change decisions will be recalculated and this simulates the effect of changing the timeline by altering the past.

UI:

OnClickoutYearButton(uint _requested_turn) is a void function that calls methods needed to load data for a requested turn:

- Saves players set faction distribution, if the distribution is different to the existing one
- Checks if the player made changes, and if so apply an awareness penalty
- If changes were made, recalculate the AI turns
- Reset changes count
- Set current turn to requested turn
- Get turn data for requested turn with *GetTurnData*( )
- Load active policies at scope of requested turn
- Update UI

UpdateFactionDistributionPips( ) is a void function that updates the "pip" bar counters that indicate faction distributions with values for the current turn.

UpdateResourceCounters( ) is a void function that updates the UI for resource counters with values for the current turn.

SetAwarenessUI( ) updates the awareness visualisation with data from the cumulative awareness factor across the timeline.

Actions:

ApplyPolicy(int _year, CTPolicyCard _policy) is a void method that inserts a specified policy into a specified year as a tracked *CTChange*:*ApplyPolicy* type.

RevokePolicy(int _year, CTPolicyCard _policy) is a void method that inserts a specified policy into a specified year as a tracked *CTChange*:*RevokePolicy* type.

SetFactionDistribution( ) is a void method that takes the player's requested faction distribution, compares them against the existing stored distribution data and, if they are different, inserts the requested faction distribution as a tracked user change.

GetFactionDistribution( ) returns the requested faction distribution set by the player in the game UI as a Vector4.

PurchaseTechnology( ) is a void function that takes a specified technology type and turn, and inserts a *CTChange*:*PurchaseTechnology* change to the user_changes.

ResetAwareness( ) is a void function that resets the *awareness_changes* data container with default values.

AddDisastersToGameChanges(List<Disaster> _disaster) is a void function that

CheckAllUserTechPurchasesValid( ) serves to prevent technology timeline continuity errors, which can occur when decisions are re-generated. For example, if a player buys technology A in turn 10, then buys technology B in turn 5, triggering a recalculation of game changes, the purchase of technology B could render technology A an invalid purchase by indirectly removing a necessary prerequisite as the same technologies are not guaranteed to be purchased. This function checks for and corrects these errors to ensure the integrity of the game state.

Utility:

GetResourcesAcrossYears( ) is a function using *RAUtility*.*Vector4List* as a return type, returning the money, science, food, and population data values from each year stored as a *RAUtility*.*Vector4List* using *GetTimelineData*( ).

GetFactionDistribtion(CTFaction _faction, CTTurnData _turn) is a float function that returns the faction distribution for a specified faction type at a given turn.

GenerateRandomFactionSpread(uint _turn) some of the worst code I have ever written. Returns four floats that total 1.0, with each of the floats generated in steps of 0.1. Does what it says on the tin, but this haunts me at night.

GetTurn( ) returns turn_data.

GetResourceTotals( ) returns a *CTResourceTotals* type with data on the total resource counts for each resource type.
GetUnlockedTechnologiesInTurn( ) returns a list of *CTTechnologies*, with the list containing each technology that is owned within the scope of the current turn.
CheckDisasterInTurn( ) returns a disaster type based on the disaster type stored for the current turn, defaults to *CTDisasters*.*None*.
GetDisasterDataAtTurn(uint _turn) returns a *CTChange*:*ApplyDisaster*
GetDisasterIntensityAtTurn(uint _turn) returns a float based on the intensity property of a disaster in the current turn, if there is no disaster returns an error value of -1.
GetAllSetPoliciesInScope( ) returns a *Dictionary* of set policies between the start of the game and the current turn, with the key being the *CTChange*:*SetPolicy* and the value as the turn it was set in.
GetAllRevokedPoliciesInScope( ) returns a Dictionary of revoked policies between the start of the game and the current turn, with the key being the CTChange:RevokePolicy and the value as the turn it was revoked in.
GetActiveTechnologyTotal(uint _turn) returns an int total of technologies owned in the scope of a given turn.

CGD : CGS : Personal Development

For my personal development component for commercial games studio, I decided to step out of my comfort zone and focus on an area of personal weakness. I am not all best familiar with the Unity Jobs system, or shaders. To gain some familiarity with these areas, I followed a 'Catlike Coding' tutorial on 'Jobs, Animating a Fractal' (Flick, J 2020).

This tutorial covers Unity's Job System, a tool that may be used to improve performance by running tasks in parallel on multiple CPU cores. It allows large tasks to be split into smaller sub-tasks, which are distributed across multiple threads, allowing the CPU to work on multiple tasks at the same time. This can significantly reduce the time it takes to complete computationally intensive, high-volume operations. This was of particular interest to me as I had not worked with Unity's Job system before, though have certainly in the past worked on systems that could have used it, e.g.: scanner sombre.

FractalController: is a class that implements a fractal generator. A fractal is a geometric pattern that exhibits self-similarity at different scales. Given an infinite depth and the capability to infinitely zoom, a perfectly self-similar fractal would extend forever in something of a loop.

A hierarchical set of Fractal component objects, each with their own position, rotation and scale are created, and each are assigned matrices that transform the mesh vertices to the appropriate positions in worldspace relative to one another.

The fractal is built by iterating over each level of the fractal, and recursively creating new parts. This process is started from an initial 'core' fractal object, and the rest are generated by iterating through to a determined depth.

When updating the fractal in Update, the position and rotation of each component in the fractal is updated based on the parent's position and rotation. This is done recursively through each layer of the fractal until the maximum depth is reached. This is a very computationally expensive process.

Level	Parts
1	1
2	5
3	25
4	125
5	625
6	3125
7	15625
8	78125

Unity's Jobs system is used for parallelised computation of fractal component world positions rotations and matricies, taking into account parent child relationships for positioning and ration.

Each additional layer multiplies the number of parts by the total of the previous one. The complexity of calculating these parts matricies raises exponentially with depth. At depth 8, this is processing 78,125 fractal components.

To assess the extent of personal development attained from this, I would consider it both a partial success and partial failure. I would feel confident making use of the Unity Job system in the future, though as far as the shader programming is concerned I was very much along for the ride with the tutorial; nevertheless it was a fun one.

Flick, J (2020-12-15),
Jobs, Animating a Fractal
[Available at]: *https://catlikecoding.com/unity/tutorials/basics/jobs/*