TUe

Ruler of the plane

Documentation

Beekhuis, S.J. & Slot, S.J.
14-10-2019

Contents

Ruler of the plane

Ruler of the Plane is a set of games illustrating concepts from combinatorial and computational geometry. The games are based on the art gallery problem, ham-sandwich cuts, the Voronoi game, and geometric network connectivity problems like the Euclidean minimum spanning tree and traveling salesman problem.

The games are implemented using C# in the game engine Unity.

There is a website for the games, where also the online WebGL version is hosted:

http://www.win.tue.nl/~kbuchin/proj/ruler/

Asset store packages

- Unity test tools

Other utilities

- Utility tool for adding and deleting player prefs, see Assets/editor folder
  (from
  https://www.reddit.com/r/Unity3D/comments/31enwc/free_playerprefs_utility_to_add_delete_modify_in/)

Games

Kings Taxes

This game is actually a collection of three mini-games centered around some simple graph theory concepts. These concepts are the 'mininum spanning tree (MST)', the 't-spanner' and the 'traveling salesman problem (TSP)'.

For the minimum Spanning Tree game we implemented Prim's algorithm. This algorithm starts from an arbitrary vertex and builds a minimum spanning tree by extending the tree iteratively with the edge which has lowest weight and connects a new vertex.

In the t-spanner game the player needs to construct a t-spanner that beats a greedy algorithm in terms of total edge length. The greedy algorithm orders the edges of the complete graph on length and keeps adding the shortest one while the two nodes do not satisfy the t-spanner constraint (path is larger than t x Euclidean distance)

In the Traveling Salesman Problem we challenge the player to find a tour that is shorter than the distance of the tour provided by Christofides algorithm.

Divide

This game is based on the ham-sandwich theorem. This theorem states that if there are n measurable objects in an n dimensional space, then there is one (n-1)-dimensional slice that cuts them all in half.

The characters in this game are a random distribution of points. They belong to different categories, corresponding to the different classes (i.e. mage, solder or archer), and are guaranteed to have a solution (i.e. a possible cut exists).

After the initial distribution of points, the algorithm swaps a few points from varying categories around to make the solution harder to find. After that, it is up to the player make the ham-sandwich cut, which might require the player to perform a few swaps first in order to create a composition of three classes that can be separated in half with a single cut.

The player's performance is measured by detecting how many different point sets the player successfully cut in half. For each point set, the algorithm computes the area in which all possible cuts can be made. Placing a cut within this area would successfully divide that particular point set in half. To find the cut(s) that divide(s) the point sets from all categories in half, the algorithm finds the intersection between the areas in which cuts can be placed for each point set. Placing a cut within this intersection area would successfully divide all point sets from different categories in half.

When overlapping the viable cut areas from each point set, the intersection of them is the area in which a cut can be made that successfully divide all sets of objects in half. This area is shown to the player (in a primal version) when he presses Q,W,E or R.

Art Gallery

The problem that you solve in this game is known as the 'art gallery problem', hence the name of the game. It requires the player to place "torches" inside a polygon (without holes) in order to light up the entire polygon. This problem is known to be NP-hard.

The lighting area of each torch is calculated via a robust stack-algorithm. We then use the Weiler-Atherron algorithm to remove duplicate parts from the sight areas and finally check if the sum of the sight areas is equal to the area of the dungeon.

Voronoi

This game is played between two players that take turns placing castles (i.e. selecting points) in order to "capture" as much land as possible. Land is captured if is it closest to a castle that you own. The winner is the one that owns the most land after a certain amount of moves.

This problem is related to the Voronoi diagram. Given a set of locations (the castles), the Voronoi diagram is the decomposition of the plane into regions such that within a region all points are closest to the same location (castle). The Delaunay triangulation is the graph that we obtain by taking the castles as vertices, and connecting two castles if their Voronoi regions touch. If no four points lie on the same circle, this graph is a triangulation. The triangles in the Delaunay triangulation are characterized by the property that the circumcircle of a triangle contains no points in its interior.

We compute the Delaunay triangulation using a randomized incremental construction, and then construct the Voronoi diagram from the Delaunay triangulation.

Unity

Basics

Unity is a game engine, which uses C# as the underlying programming language for scripting. We can highly recommend to use Visual Studio (or maybe another modern IDE) over monodevelop that is packaged with unity. There is some package integrating Visual Studio and Unity, install this package.

The general workings of Unity are well-explained on their own site. I would recommend doing one or two tutorials. For example Roll-a-ball and 2DUFO. See https://unity3d.com/learn/tutorials

Make sure you understand at least the following:

- How to make a 2d game in unity (camera perspective, z-axis etc.)
- How a prefab object works in unity
- How to change scenes using a script

Furthermore their documentation is also very useful:

https://docs.unity3d.com/Manual/index.html

Tips and tricks

This section contains general unity tips and tricks

- Use prefab objects for objects that you want to re-use, e.g. game objects that are dynamically added to the scene.
- Use scriptable objects as data containers that do not need an associated Game Object (no transform needed, etc).
- [Serialize field] makes a field editable from the unity Inspector (even if it's private), by default only public variables of certain default types are editable.
- Use [Serializable] to make a custom class editable from Inspector
- You can give the editor a color when it's running. This is very useful since changes made in the editor while the game is running are not preserved.
- Sometimes you have prefabs containing elements several levels deep (e.g. text inside a button inside a canvas inside a game controller container object). Unity does not allow you to change prefabs this deep in the tree view and if you change some setting in a single scene and apply the change you might inadvertly make some locally changed setting global. The best solution is to make a special empty scene with a fresh prefab instance and to make your changes in this instance.

Game design

In this section we will give some specifics on how we implemented the geometric games using Unity.

Asset structure

For every game, we use a typical structure for the assets that are used. The game assets are divided into the following folders (in alphabetical order):

- Art – Any files related to game art
- Levels – Scriptable objects that capture level data
- Material – Unity materials for drawing
- Prefabs – Prefab objects for re-use of Unity game objects
- Scenes – Unity scenes
- Sprites – Sprite files
- Textures – Texture files

Note that not all folders need to be used for a game.

Assets that are re-used in multiple games (like textures, materials, art) can be stored in the 'General' folder of the Assets. Like most Unity projects, any script files in C# are stored in the 'Assets/Scripts' folder.

A typical game consists of several scenes that are linked together. A non-exhaustive list of typical scenes for a given game are:

- Menu scene
- Game explanation scene
- (Introduction scene)
- Level scene
- Victory scene

The level scene will be responsible for loading the game's level and letting the user play the actual game.

Scenes will typically re-use some components (e.g. camera, backplane, canvas, etc). Therefore, for some games these components were put into a single game object that is stored as a prefab for re-use. One should be careful, however, when changing anything in the prefab since this might make the change global (for all scenes).

Script structure

For each game we have used the principle of model-view-controller (MVC) to separate the concerns into the three different categories. A game will have a controller that is involved in the game loop (updating) and changing the game state. The controller is also responsible for initializing and advancing levels. The view part of MVC is responsible for drawing to the unity scene, using the current game state. These classes can also take user input and call the relevant method in the controller. Finally, any data containers are stored separately in the model part of MVC. Since we use Unity, some model parts can be implemented using game objects in the scene instead of writing custom classes to store the data.

For simplicity, in most games the controller holds the variables that model the game state (instead of storing an explicit custom GameState.cs class).

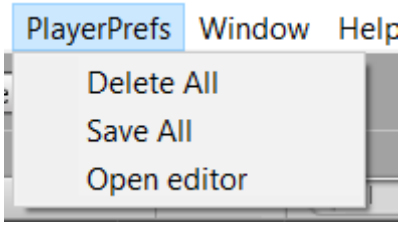
The games make use of several geometric concepts and algorithms, as well as some mathematics and data structures. These are separated from the games into a re-usable library folder name 'Assets/Scripts/Util'. The top-level structure here is:

- Algorithms – static classes containing algorithms
- Data Structures – classes implementing some standard data structures
- Geometry – the main geometry library
- Math – an extension of the functionality of Unity's Math and Mathf

Scripts that are re-used throughout many games can be put into the 'Assets/Scripts/General' folder. These typically are simple UI/Menu related scripts as well as general interfaces for scripts.

Player prefs

Any data that needs to be stored for longer than a game session can be stored using the PlayerPrefs functionality in Unity. This stores key-value pairs locally in the file system (or in the browser in case of WebGL). This is used in the KingsTaxes game to determine whether the player has already beat all regular levels and it should enable endless mode. See https://docs.unity3d.com/ScriptReference/PlayerPrefs



A tool is used to add, delete, and inspect the PlayerPrefs from the Unity Inspector. This tool appears as an item in the top menu bar, called "Player Prefs". This tool is implemented in the 'PlayerPrefsEditorUtility' present in 'Assets/Scripts/Editor'. See https://www.reddit.com/r/Unity3D/comments/31enwc/free_playerprefs_utility_to_add_delete_modify_in/

Level creation

The games that include levels (all except the Voronoi game) have a corresponding 'Levels' folder inside their assets folder. A level is implemented as a scriptable object that holds all information necessary to specify it. For each game this is a different scriptable object class. The levels placed into the level folder are used by the game controller when playing the game. The levels are simply played in sequence in alphabetically order.

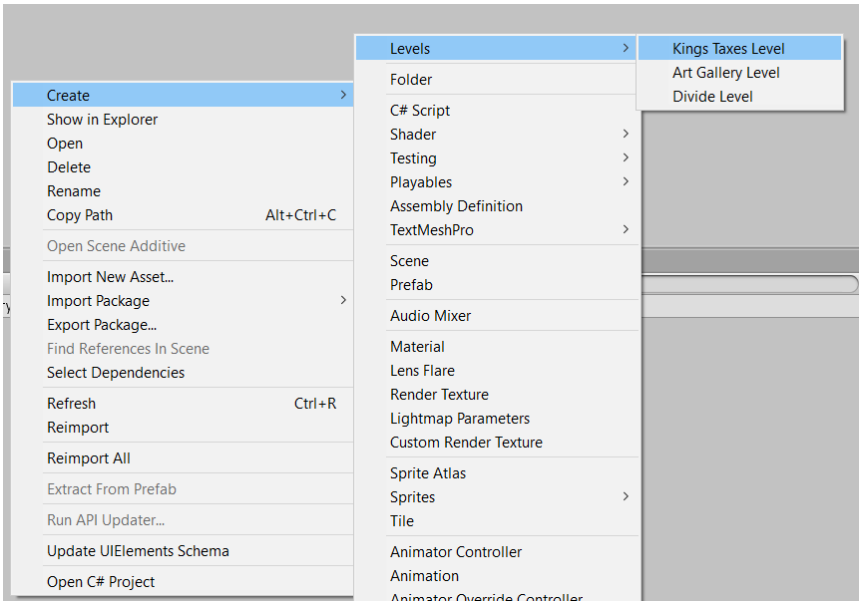
All games come with some pre-made levels to show off the games features. In this section the ways of creating new levels for the pre-existing games will be discussed.

Manual level creation (not recommended)

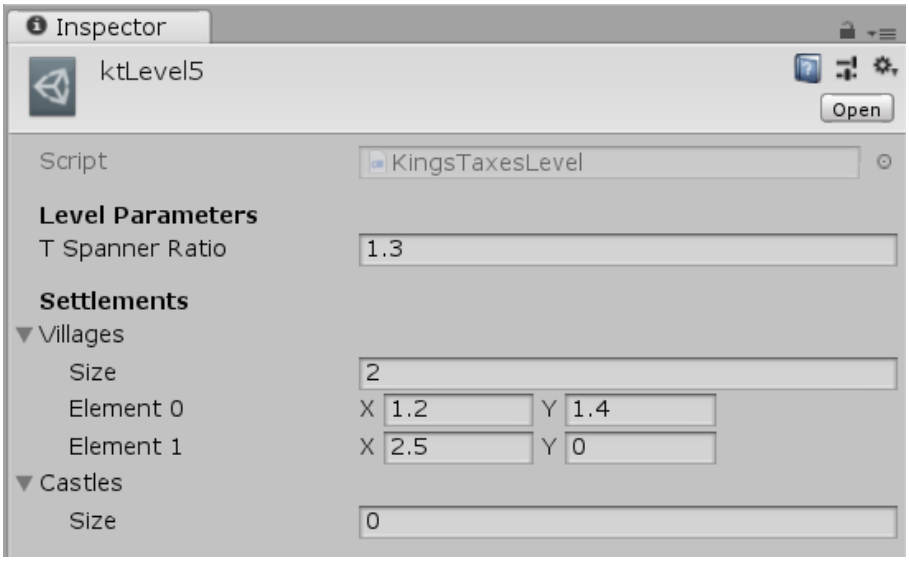
The level object can be created inside the Unity Inspector. This is not recommended since it requires much manual input in order to create a decently large level, since coordinates of points need to be typed in one-by-one. It is also hard to input an interesting level without visual guidance.

To add a level to a game, first move to the corresponding levels folder inside the unity Project view. Note that only levels in this folder will be considered.

Right-click inside the folder, select ‘Create/Levels/’, and then select the corresponding level. This will create a new scriptable object called ‘agLevelNew’/’ktLevelNew’/’divLevelNew’. You can replace “New” with a number corresponding to the order in which you would like the level to appear.



When selecting the new scriptable object, you can use the Inspector to fill in the necessary level data. Note that for filling in arrays in Unity, you first need to enter the size.



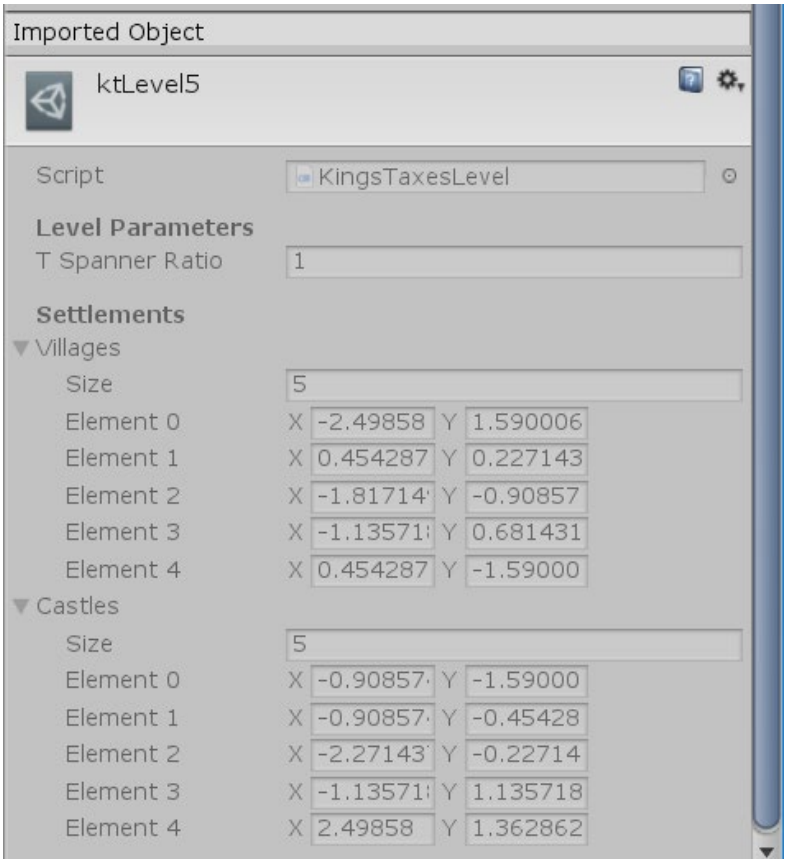
Importing levels

Instead of manually inputting all 2D vector information of points/polygons, one can use the drawing editor IPE (see www.ipe.otfried.org). Inside the drawing editor, you can draw the corresponding level and then import this into Unity. For the currently implemented games, creating a level in IPE will either be drawing a polygon or placing different types of point markers. The exact requirements for the IPE files are described below for each game.

Once an IPE file is created and given a suitable name, one can import it into Unity. Again, make sure to import a level in the corresponding levels folder. Importing a level can be done in two different ways:

- Drag and drop the .ipe file into the Unity Project view. This will automatically import it and create the level object.
- Right click inside Unity Project view, click on “Import New Asset…” and select the .ipe file from your local files.

Once imported, the IPE file will be copied to the selected folder and the associated level object will be created. This level object is generated using the imported IPE file and cannot be edited manually. See below for an example of how the imported level object looks.



If changes need to be made to the level, one can of course remove the old level and import a new IPE file again. However, another (faster) approach is to change the copied IPE file inside the level folder directly and then trigger a reimport. You can edit the IPE file in the level folder by opening the imported object (double clicking or right-click, select “open” on the object in Unity), given that you have IPE as your default program for files with extension .ipe.

To trigger a reimport:

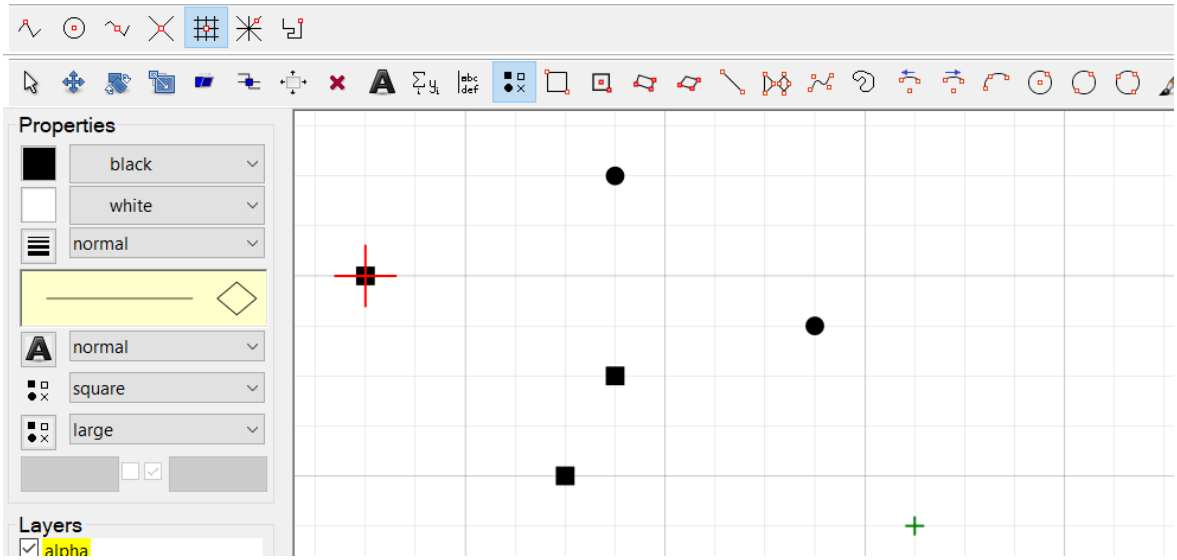
- (automatically) editing a file, saving, then returning to the Unity Editor.
- Right-click asset, select ‘Reimport’.
- In the menu, select ‘Assets/Reimport’

When importing a IPE file (given a recognized naming pattern), the corresponding level object will automatically be created. This uses a scripted importer that is used to support IPE file for importing. The custom importer (called ‘LoadLevelEditor.cs’) is located at ‘Assets/Scripts/Editor’. Read https://docs.unity3d.com/Manual/ScriptedImporters.html for more information on scripted importers and how they work.

Kings Taxes

A kings taxes level consists of a collection of points and potentially a parameter *t* for the *t*-spanner. The points can either be villages or castles, which at the moment only differ visually in the game. The point set can be given in IPE by placing 'marker' points of different types. A marker point can be selected by selecting in the menu or pressing shortcut (M). In the menu on the left one can change the marker shape, by default this is a disk. The Kings Taxes level should thus be created as follows:

1. Create a new IPE file.
2. Place a 'disk' marker point in the plane for each village.
3. Place a 'square' marker point in the plane for each castle.
4. Save IPE file as "ktLevel?.ipe" or "ktLevel?-??.ipe" where *?* is a number used for ordering and *??* is the *t* parameter used for spanners (default 1.0 if absent).
5. Import IPE file into 'Assets/KingsTaxes/Levels' folder in Unity.



Divide

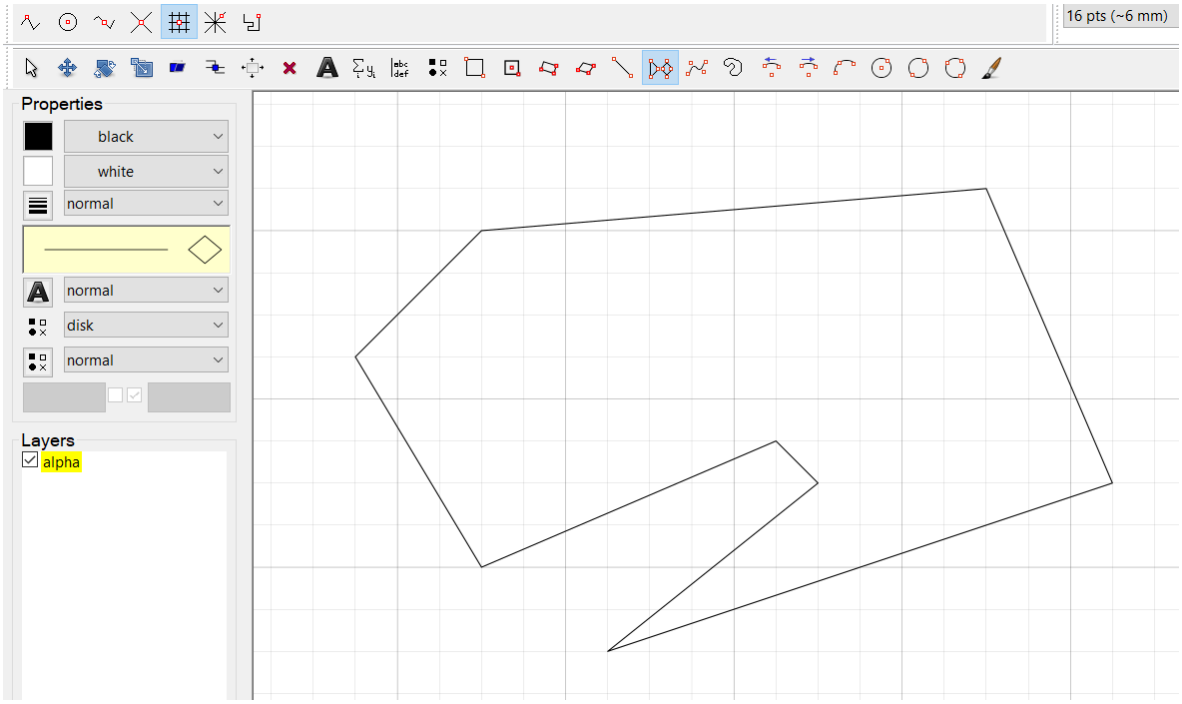
A Divide level consists of a collection of three types of points, that correspond to spearmen, archers, and mages, and a parameter *k* for the maximum number of swaps. Each set should consist of an even number of points. Similar to Kings Taxes, the point set is given by placing 'marker' points in IPE. The types are 'disk', 'square', and 'cross'. Select a marker point by pressing M and change the marker shape in the menu on the left. The level is created as follows:

1. Create a new IPE file.
2. Place a 'disk' marker point in the plane for each spearmen.
3. Place a 'square' marker point in the plane for each archer.
4. Place a 'cross marker point in the plane for each mage.
5. Save IPE file as "divLevel?.ipe" or "divLevel?-??.ipe" where *?* is a number used for ordering and *??* is the *k* parameter for the number of swaps (default 0 if absent).
6. Import IPE file into 'Assets/TheDivide/Levels' folder in Unity.

Art Gallery

An art gallery level consists of a polygon without holes and a parameter *k* for the maximum number of lighthouses. The polygon can be drawn using the "Polygons" tool in the menu, the shortcut is SHIFT-P. The polygon can either be drawn clockwise or counter-clockwise. The level is created as follows:

1. Create a new IPE file.
2. Draw a polygon (press SHIFT-P and add multiple points)
3. Save IPE file as “agLevel?.ipe” or “agLevel?-??.ipe” where *?* is a number used for ordering and *??* is the *k* parameter for the number of lighthouses (default 1 if absent).
4. Import IPE file into ‘Assets/ArtGallery/Levels’ folder in Unity.



Voronoi

The Voronoi game does not consist of levels, thus no level for it can be created.

Generating levels

Game creation

In this section we will shortly describe the steps needed to create a new mini-game and incorporate it into the overall game. To help you get started, you can study the implementation of other mini-games and read the previous sections on Unity and the Game Design. We encourage you to reuse parts of the Unity scenes for other games as well as game art.

During development, the geometric library was separated from the scripts regarding game implementation for easy reuse. Thus, one can already make use of a graph or DCEL implementation. Please study the contents of the library to see what is present and save yourself effort in re-implementing certain structures/algorithms. The library is not complete, however, so it will need to be extended to implement other geometric concepts and algorithms.

Levels

Here we discuss the steps necessary to create level objects and allow a user to import/generate the level. There are many ways to accomplish this, but here we discuss the method used for the other games. To keep in line with the other mini-games, we encourage you to stick to this method.

Scriptable objects

The data for each level is stored inside a scriptable object, which acts as a data container in Unity. For a new type of level, one needs to create a custom class that inherits from 'ScriptableObject'. Inside the class all data is placed in public fields that are necessary to describe the level. These scriptable objects can then be used inside the game controller to initialize a new level. The game controller thus stores a list of all levels in sequence.

```
[CreateAssetMenu(fileName = "ktLevelNew", menuName = "Levels/Kings Taxes Level")]
2 references
public class KingsTaxesLevel : ScriptableObject {

    [Header("Level Parameters")]
    public float TSpannerRatio = 1f;

    [Header("Settlements")]
    public List<Vector2> Villages = new List<Vector2>();
    public List<Vector2> Castles = new List<Vector2>();

}
```

To add the ability for manual creation of a scriptable object inside Unity, you can use the class attribute 'CreateAssetMenu'. This will allow the scriptable object to be listed under the 'Assets/Create' menu. You should give parameters for the default file name of the newly created object and give a name inside the menu for the user to select. See https://docs.unity3d.com/ScriptReference/CreateAssetMenuAttribute.html for more details.

For manual creation of a scriptable object all public fields need to be serializable, meaning that the fields are editable inside the Unity Inspector. Most default datatypes are serializable, though for example 2D arrays are not. One can create a custom data class with class attribute '[Serializable]' to enable editing inside Unity. See 'Vector2Array.cs' in the folder 'Assets/Scripts/General/Model' for an example.

Importing

If possible, it is encouraged to allow the user to import IPE level files instead of manual inserting level data. Importing IPE file is implemented in the 'LoadLevelEditor.cs' which is stored in the folder

‘Assets/Scripts/Editor’. The method ‘OnImportAsset(AssetImportContext ctx)’ is called whenever an .ipe file is being imported. Here the relevant scriptable object is created based on the file name. Here one should insert a call to a new function that creates the level object that was just created.

Th

```
public override void OnImportAsset(AssetImportContext ctx)
{
    var path = ctx.assetPath;
    var name = Path.GetFileNameWithoutExtension(path);

    var fileSelected = XElement.Load(path);

    // switch between which level to generate based on file name
    Object obj;
    if (name.StartsWith("agLevel"))
    {
        obj = LoadArtGalleryLevel(fileSelected, name);
    }
    else if (name.StartsWith("ktLevel"))
    {
        obj = LoadKingsTaxesLevel(fileSelected, name);
    }
    else if (name.StartsWith("divLevel"))
    {
        obj = LoadDivideLevel(fileSelected, name);
    }
    else
    {
        // no file name match
        EditorUtility.DisplayDialog("Error", "Level name not in an expected format", "OK");
        ctx.SetMainObject(null);
        return;
    }

    // add generated level as the main imported file
    ctx.AddObjectToAsset(name, obj);
    ctx.SetMainObject(obj);
}
```

The IPE file is internally stored in XML format. Inside C# we use Linq and XElement to traverse the XML tree. As can be seen in the above screenshot the IPE file is loaded from the asset path and a XElement is created. This is the root element in the tree and can be traversed. You can use either functional or declarative statements to find the elements you need. See https://docs.microsoft.com/nl-nl/dotnet/csharp/programming-guide/concepts/linq/xelement-class-overview for more details.

Generation

Menu

To make your new mini-game reachable from the main menu, you will likely need some button. Upon clicking the button, one should run the ‘SceneLoader.LoadScene’ method with the scene name as a parameter. In the main menu scene, a game object (also called ‘SceneLoader’) is present containing the ‘SceneLoader.cs’ script, which can be used for this purpose.

Also make sure your mini-game scenes are present in the build settings (in the main menu under ‘File/Build Settings…’), otherwise they will not get loaded when building the entire game.