

SERIALIZATION AND TOOLING FOR A USER-FACING LEVEL EDITOR

Sievers, Nathan

Master's thesis / Specijalistički diplomski stručni

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Algebra University College / Visoko učilište Algebra**

Permanent link / Trajna poveznica: <https://um.nsk.hr/um:nbn:hr:225:898953>

Rights / Prava: [In copyright](#) / [Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2024-10-06**



Repository / Repozitorij:

[Algebra University - Repository of Algebra University](#)



ALGEBRA UNIVERSITY COLLEGE

GRADUATION THESIS

**Serialization and Tooling for a User-Facing
Level Editor**

Nathan Sievers

Zagreb, September 2022

Student must sign the Graduation Thesis on the page preceding Acknowledgements with a date, location of the graduation thesis and a following note:

By submitting this Proposal, I hereby acknowledge and accept the **Declaration on authenticity and academic ethics** that reads:

“By submission of this application, I hereby acknowledge and accept that the Master’s Thesis which will be prepared as a result of this application will be my own individual work, prepared in accordance with advices and consultations received from my mentor and other teachers / experts I might consult, and using literature and sources referenced in my Thesis.

Furthermore, I acknowledge that no part of my Thesis will be taken from other uncited authors, and that I will not breach the rights of any author.

While preparing my Thesis I will follow best academic practices in respect to authorship and proper citing and referencing work of others.

If breach of stated principles is proven through an independent check of the referenced sources any time in the future, I am fully willing to bare corresponding formal consequences, including the possibility of my diploma and academic degree achieved as a result of this Thesis to be revoked.”

Zagreb, April 2023

Acknowledgements

To my friends and teachers at Algebra, Igor Margan, Igor Porkolab, Victor Popa, and my advisor Goran Dambic. Thank you for your help, patience and guidance. The paper Acknowledges Lidija, my mom and dad Patricia and Robert, and by girlfriend Ariaah. I would not be here without your support.

Abstract

This paper covers the architectural decisions and programming contributions created for Impulse, a fast-paced 2D platforming game created with Unity. Impulse was designed with extremely high player performance in mind, and so requires specific design and technical challenges to be met. Additionally, Impulse places heavy emphasis on user authored content, with a custom level editor prioritized as one of the selling points. There were many design challenges in building Impulse, but this paper will focus on three of them: quicksaves, object serialization, and parsing level data.

Key words: game architecture, slotmap, serialization, level editor.

Table of Contents

1. Introduction	1
2. Background.....	3
2.1. Classical Unity Architecture.....	3
2.2. Limitations of Unity Builtin Editor	5
2.3. Limitations of Unity Serialization	5
2.4. Common Serialization Implementations: Scriptable Objects.....	6
2.5. Common Serialization Implementations: Entity Component System.....	7
2.6. The Slotmap Data Structure	8
3. Methodology.....	11
3.1. High Level System Overview.....	11
3.2. Unity Component Serialization	11
3.3. Aggregating Serialized Components into GameObjects	14
3.4. Saving and Loading	16
3.5. Unity Prefab Serialization	19
3.6. Parsing Level Geometry	22
3.7. partitioning a rectangular array into subarrays of constant value.....	23
3.7.1. Implementation:.....	27
4. Results	32
4.1. Save State	32
4.2. Level Editor	35
5. Discussion.....	38
Conclusion.....	39
List of Abbreviations.....	40

List of Figures.....	41
List of Codes.....	42
References	44
Appendix	45

Table of Contents is created automatically through the following options: **References – Insert Table of Figures.**

1. Introduction

Unity traditionally uses a component-based architecture. Each game world is organized as a tree of game objects, and each game object is organized as a list of components. Components contain both data and code, and the tree is traversed calling specific functions on each in order to synchronize updates. This system is logically straightforward to use but atomizes the code along difficult-to-serialize boundaries. Sections 2.1-2.5 provide background on this system, as well as architectural alternatives and serialization methods.

These standard alternatives to Unity components are powerful but often difficult to design with. In the case of a 2D platformer, they are too much machinery to be worthwhile. Section 2.6 provides background on the slotmap, an obscure data structure known mainly by game developers. The paper then proceeds to introduce an alternative architecture based on metaprogramming and SlotMaps.

The first part of the original work revolves around this topic. A metaprogram generates an interface for each component that allows for data to be serialized into the slotmap. Special care is taken to preserve references between „logical“ objects even if the actual objects are created and destroyed between saves and loads. This work is described in sections 3.1-3.4

The second part of the original work revolves around the integration of part 1 with a user facing level editor. A user-made level format is introduced which is structured around reading images for geometry data, visual data, and placement of game objects (a.k.a. „sprites“) into a game scene. A third-party JSON library specifies serialization of structs, but a custom layer is autogenerated on top of this to translate between developer and user-levels of granularity and to integrate with the slotmap. This is handled by a 2nd order metaprogram which gathers specific configurations of game objects („prefabs“) and creates a secondary plain-old-data layer that the JSON can work with. This 2nd order metaprogram then also generates a metaprogram which links all relevant references together so that the slotmap can rebuild „default“ versions of each object only from their type. This work is described in section 3.5.

The third part of the original work deals with parsing an image file and turning it into axis aligned boxes that can be used for level geometry. This is a well-studied problem in computer

science [7] which can give an exact solution, however, the complexity of this solution is not needed, since we can get an approximate answer without incurring slowdown in game. Section 3.7 provides an overview of this work, and 3.7.1. presents an original solution which gives an approximate answer with less overhead.

Section 4 shows the simplicity of working with such a system, both for saving and loading as well as using the level editor. Section 5 describes alternative ways of designing a similar system as well as future improvements.

2. Background

2.1. Classical Unity Architecture

The Unity Engine classically uses what is called a component-based architecture. Content is organized into components, which are just classes that control a single behavior on a single object in the game world. Examples of these are built-in components such as Transform (position and rotational data) and MeshRenderer (code for drawing a 3D model). Most components in Unity however are custom-made for each game. Examples could include a PlayerController for updating Transform Data from velocity and input, or a HealthController for managing an object's health. Architecturally, a Unity Scene is a tree of GameObjects and a GameObject is a list of components (called MonoBehaviors in code). [1]

The flow control of this system works as follows: each MonoBehaviour has built-in callback functions called Unity Events which execute across all MonoBehaviors in the scene that implements them.

For example, any MonoBehaviour which implements the Start() event will fire on the first frame of the objects' lifecycle. Any MonoBehaviour which implements Update() will run Update() once per frame. All Update() events will fire before moving on to the next event. Figure 2.1.1. shows the full execution order in detail.

What is of interest in this system is the production pipeline for the developer and the way in which code and data is logically grouped. In this setup, programming work in Unity consists of creating new components, assigning them to GameObjects, placing them in a scene, and configuring the initial values as needed for gameplay. One component can be used on many different GameObjects. This will mean a coarser level of granularity will need to be artificially introduced for serialization.

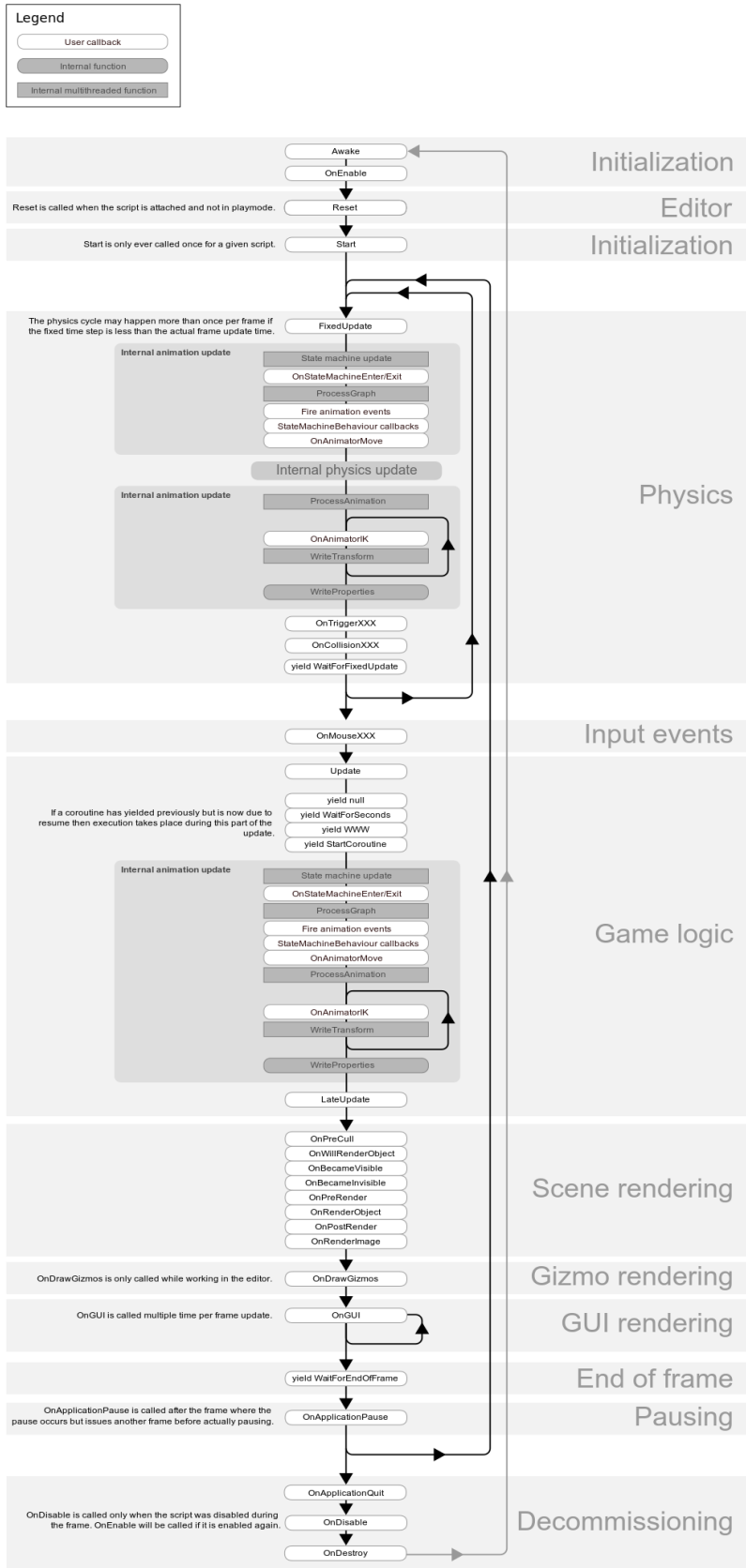


Figure 2.1.1 Event Execution Order, retrieved from [2]

2.2. Limitations of Unity Built-in Editor

While the Unity Editor is very convenient for developers, it is not practical as a user-facing level editor. In order for players to make levels using Unity, they would need to have access to the unpacked assets from the game and would be able to publish and distribute raw executables with arbitrary code modifications. This is less than desirable because it requires shipping large amounts of unpacked data, allows users to copy content into unrelated projects, and ship (potentially malicious) copies of the game.

A partial solution would be to bundle all scripts into an opaque assembly, or to implement some form of DRM within the editor itself. This strategy is done for some Unity plugins such as Wikitude [8], but it does not solve the problem of redistributing builds or injecting malicious code. In the case of a Unity plugin, it is desired that new executables be created, where in this case it is not.

This does not help much, since that same opaque assembly could be moved into other projects, and any functionality available for making levels could be used for making other projects as well. It also does not account for art assets, which would be usable in other contexts without an extreme amount of obfuscation.

Even if all of these problems were somehow solved, it is also not feasible to ask players to learn a complicated, general-purpose editor like Unity to make content for a specific game. The flexibility that Unity offers to a developer becomes a hinderance when aimed at players, as the majority of Unity's functionality is not applicable to any single project.

For all of these reasons, it is necessary to develop a separate editor to aim at the public.

2.3. Limitations of Unity Serialization

In order for a level editor to work, information must somehow be transferred from it to the main application. Thus, there must be some serialization mechanism to encode assets and level data that can be read at runtime. Since Impulse also implements a quicksave and quickload feature, it becomes important for serialization and deserialization to be as fast as possible.

Unity's built-in mechanisms are not equipped to handle this problem. There is very good support for serializing assets for use within the editor, but no convenient methods exist for interoperating with an external program.

The problem comes from the fact that most of the tooling is built for the editor version of the application, and once the standalone game is created, the serialization system cannot be used in the same way. Customizing the Unity Editor is possible by creating scripts in a special "Editor" folder, but this code does not get bundled into the game.

Since the only thing which players have access to is the game application, one is left with the same problem as in the previous section: to expose the developer-facing features of Unity to the player the internals of the game must be exposed. For the same reasons as before Impulse must therefore implement its own serialization system.

Unity's serialization system will still be useful for bootstrapping into a custom version. Unity has the ability to create serialized configurations of GameObjects which can be referenced in custom editor code. This feature, which Unity calls "Prefabs", will be used to index the assets in the game and automate the custom serialization process [3]. Other Unity features such as ScriptableObjects will also see use cases, with assistance from some obscure data structures and programming paradigms. All these features are described in the subsequent sections.

2.4. Common Serialization Implementations: Scriptable Objects

A common method for object serialization that is specific to Unity is the idea of scriptable objects. A scriptable object is a struct that can be serialized by Unity's internal API and can be initialized with values from inside the editor. This idea should not be confused with prefabs: a scriptable object is not a GameObject, merely data which is flagged to be referenceable by the rest of the editor [4].

Although Impulse will not follow this setup, the typical use for scriptable objects and serialization is as follows: First, create a corresponding ScriptableObject for every GameObject. This scriptable object will just serve as a struct containing gameplay-relevant data. Next, create an instance of the ScriptableObject for each possible set of values the GameObject could be initialized with. Finally, load in the desired configurations from these prefabs as needed.

This is a nice system, and one Impulse shall make use of, but it does not solve the hard parts of the serialization problem for us, because it assumes that the set of configurations for each GameObject is very small. Impulse requires that the player be able to save and load at any point, and the game contains many floating-point values. Moreover, some of the fields that need to be serialized will be container types, so the set of configurations will not even be finite.

Another part of the serialization problem which ScriptableObjects do not solve is the transience of entities within the game. Components may be added or removed from a

GameObject at runtime, and objects may be created or destroyed. The hard part of serialization is keeping track of which objects need to be reconstructed and what data needs to be populated. Once that information is understood serialization is easy, so ScriptableObjects are more of a convenience than anything.

2.5. Common Serialization Implementations: Entity Component System

An old but increasingly popular way of serializing transient objects in a game engine is through an architecture pattern known as Entity Component System. Entity component system has been around since at least the early 90s but has recently seen a surge in interest as well as native support within the Unity Editor.

First, a note on terminology: both Unity and Entity Component System use the word “component” to mean separate things. A “Unity Component” is a synonym for a MonoBehaviour. An “ECS Component” is an item in an array of homogeneous data, as shall soon be explained. Care is taken in this section to always use the word “ECS Component” when referring to the later and shall not be needed in later sections. For the remainder of the paper, a plain “component” may be understood to mean “MonoBehavior”.

The basic idea of Entity Component System (commonly abbreviated to ECS), is that gameplay data and code be divided along similar functionality rather than along which "object" the data refers to. This is best understood through an example.

Consider a 2D projectile. This projectile will have an object to world transform, a sprite texture that it needs to render, a physics component, a particle system, and some other custom scripts to explode on contact and damage certain objects.

As discussed earlier, a typical Unity implementation of this would see this GameObject as a list of MonoBehaviors. These MonoBehaviors each run their functionality independently and developer-made code is run through callbacks functions like FixedUpdate, OnCollisionEnter, and so on.

This is a nice mental division of the data but from the standpoint of the computer it makes very little sense. From both a data and a code standpoint, similar MonoBehaviors on different game objects are more alike than different MonoBehaviors on the same GameObject. Different types of MonoBehaviors have essentially nothing to do with each other, and it would make more sense to aggregate all the physics data across all GameObjects, all of the mesh rendering data across all components, and so on. Taking this one step further, one can aggregate only the data and then have a single instance of code per "system" which loops through these homogeneous collections.

Entity Component System is essentially this setup. The notion of a GameObject becomes an index, also known as an Entity. MonoBehaviors break into two separate parts. The data gets

aggregated into arrays called “ECS Components”, which are indexed by their corresponding entity. Code is then separated into Systems, which operate on "all relevant components" and "all relevant entities" at once. For more information on ECS architecture see [5].

People usually advertise ECS from a performance standpoint. By aggregating the data into homogeneous arrays, one expects much better cache performance. However, it is also a win from a serialization standpoint. The reason for this is that one has all the game data located in one convenient place. One could easily decide which Components are gameplay relevant and write this to some backup array when needed. The same could be done with the entity list in principle, provided one solves some of the problems described below.

The difficulty with ECS is not in the architectural complexity. That part is fairly simple. The complexity comes from fitting one’s game code into this model and therefore consistently visualizing things in these terms. It is a nonzero amount of work to think about your code from a data-oriented standpoint and may not be worth the trouble. It is worth noting that Unity offers much of this out of the box through a hybrid system, but the author finds that it is much harder to think about code that is organized this way and would prefer to work with ECS Components and Systems as a single logical object. This is especially true when a complex serialization setup is required for both quicksaves and a level editor, and so any black box that Unity provides would have to be opened up anyway. Finally, Unity’s built-in ECS is still early enough in development that it is officially recommended against use for commercial products.

It should also be emphasized that none of the above constitutes criticism of ECS in general. It is much more performant than what it proposed here. However, for a 2D sprite-based game this level of hardcore performance is not a huge factor in making architectural decisions, and so it is desirable to opt for something both serializable and easy to understand.

2.6. The Slotmap Data Structure

The previous section alluded to the fact that there may be some more nuance to the serialization of entities. This shall now be discussed in more detail.

Suppose one follows an Entity-like system in which GameObjects correspond a unique index in some array. Suppose that in the first frame, index 0 corresponds to object A and index 1 corresponds to object B, and that object B references object A. Suppose that the player saves, then object A is destroyed, and then the player loads. The reconstructed object A is now a different object in memory, so B's reference is no longer valid.

One could try to fix this by storing the index of the referenced entity rather than a reference to the GameObject. Suppose this setup is used, but now object A is destroyed, a new object C is created in slot 0 on the next frame. B's reference now points to a valid index, but the contents of that index have been changed without B's knowledge.

What is needed is a data structure with array-like element storage but dictionary-like insert lookup and erase. These operations should be constant time and the keys should detect reused slots as previously described. This is the purpose of a slotmap.

The idea of a slotmap is extremely simple. A single level of indirection with the indices is introduced by storing two arrays instead of one. The first array is an ordinary array of data. The second is an array of SlotKeys, which are just integer indices into the data array together with an integer counter. When a user calls insert, lookup or erase, they do so with an index into the SlotKey array and a generation number. If the generation number matches the counter in the SlotKey, the slotmap operates on the data index corresponding to whatever value was in the SlotKey index. When data is inserted into an old slot, the corresponding counter in the SlotKey array is incremented. This solves the previous problem, as index information has access to both location and version number.

To maintain constant insert, lookup, and erase, unused entries in the SlotKey array point to each other in a list. The SlotMap maintains an index to the head and tail of this list for constant access.

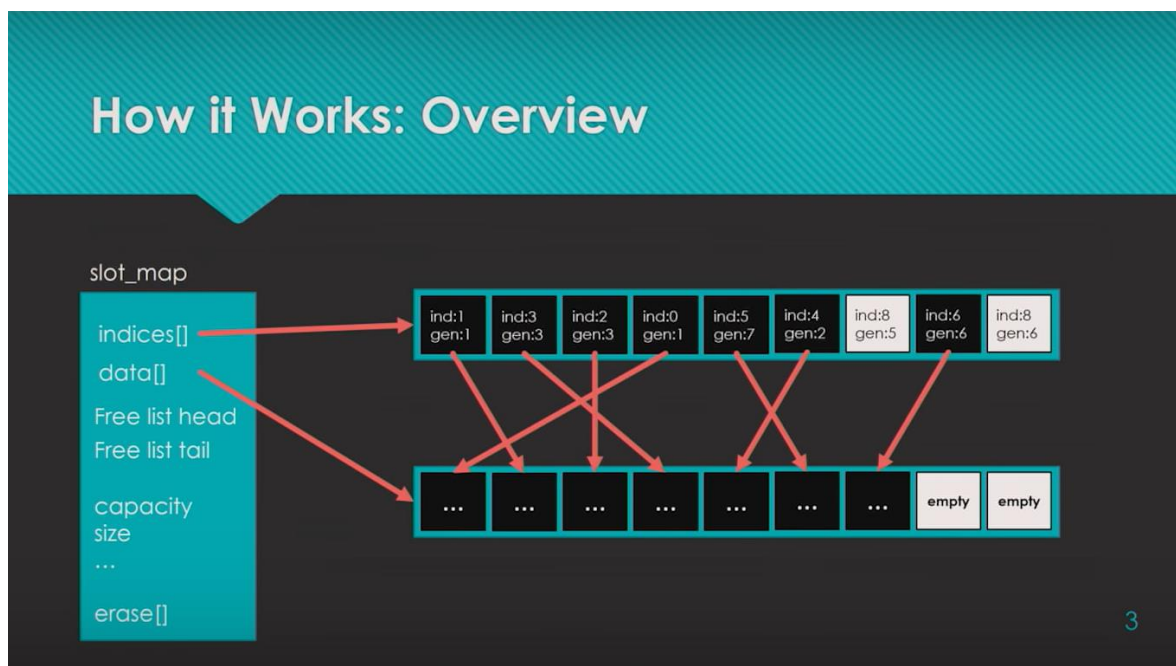


Figure 2.6.1 Diagram of a SlotMap, retrieved from [6]

SlotMaps support constant time lookup, insert, and delete. The above figure shows a diagrammatic view of the data structure in a typical state.

SlotMaps also have several other benefits which will be made use of later. First of all, if one merely wants to iterate through all of the contents of the data, one can access this as a raw array. Furthermore, if one wishes to reconstruct a game scene, as will be done in the loading system, one can separate the problem into two steps: serialization of GameObjects can be

handled “one at a time” in a more or less standard way. serialization of the scene (including inter-object references) can be done simply by remembering the status of all of the SlotKeys and a template object of what they point to. If one can reconstruct the SlotMap in the same order with the same generation numbers, references will remain intact.

For more information on the SlotMap architecture, see [6]

3. Methodology

3.1. High Level System Overview

The serialization system in Impulse uses an Entity-like system for keeping track of GameObjects, but while maintaining a standard Unity architecture. Rather than construct everything around ECS, GameObjects and need only register themselves into a SlotMap. Serialization of data takes place through a bit of metaprogramming which will be described in detail, but the important idea is that this is essentially Entities without the Components or the Systems.

Everything in this chapter is original work with the exception of the SlotMap container and JSON parsing library. Therefore, the attributes and mechanisms described are not available out of the box.

3.2. Unity Component Serialization

The most granular level of serialization takes place on components. When designing a new Unity Component, the programmer specifies the class as partial and marks it with a custom attribute called `GenerateSaveLoadAttribute`. Next, any gameplay-related fields which should be serialized are marked with another custom attribute, `SaveLoadAttribute`.

Code 3.2.1 shows an example use case of this setup with a simple component class. The `bounds` field is tagged for serialization. Transform data and component type are automatically serialized.

```
[GenerateSaveLoad]
public partial class CameraArea : MonoBehaviour
{
    [SaveLoad]
    public Vector2 bounds = new Vector2(40, 30);

    public Vector2 min
    {get {return (Vector2)transform.position - 0.5f * bounds;}}
    public Vector2 max
    {get{ return (Vector2)transform.position + 0.5f * bounds;}}
}
```

Code 3.2.1 A simplified `CameraArea` class tagged for serialization

These attributes allow an offline metaprogram to find any class marked with `GenerateSaveLoadAttribute` and extend it with another partial definition. This definition

automatically implements a common `ISaveLoadable` interface for the serialization system to access.

The first step of the metaprogram is to identify all the fields marked with `SaveLoadAttribute` or `SaveLoadOverrideAttribute` and list them in an enum called `SaveLoadFields`. It then creates two functions within the common interface: one which writes to a `Dictionary<int, object>` and another which reads from a `Dictionary<int, object>`. The keys are simply values in the enum cast to integers, and the values are the corresponding values of the tagged fields.

Code 3.2.2. shows the code autogenerated for the user-made example from 3.1.1. Some of code is omitted for clarity, see appendix.

```
public partial class CameraArea : MonoBehaviour, ISaveLoadable
{
    public Dictionary<int, object> data = new Dictionary<int, object>();
    public SaveLoadableType type { get; private set; } =
        SaveLoadableType.cameraarea;
    private enum SaveLoadFields
    {
        type = 0,
        save_load_id,
        transform_position,
        transform_scale,
        bounds
    }

    public void load_data(Dictionary<int, object> data)
    {
        if (data.ContainsKey((int)SaveLoadFields.type))
            type = (SaveLoadableType)data[(int)SaveLoadFields.type];
        if (data.ContainsKey((int)SaveLoadFields.save_load_id))
            save_load_id = (SaveLoadKey)data[(int)SaveLoadFields.save_load_id];
        if (data.ContainsKey((int)SaveLoadFields.transform_position))
            transform.position =
                (Vector3)data[(int)SaveLoadFields.transform_position];
        if (data.ContainsKey((int)SaveLoadFields.transform_scale))
            transform.localScale =
                (Vector3)data[(int)SaveLoadFields.transform_scale];
        if (data.ContainsKey((int)SaveLoadFields.bounds))
            bounds = (UnityEngine.Vector2)data[(int)SaveLoadFields.bounds];
    }

    public Dictionary<int, object> save_data()
```

```

{
    data.Clear();
    data.Add((int) SaveLoadFields.type, type);
    data.Add((int) SaveLoadFields.save_load_id, save_load_id);
    data.Add((int) SaveLoadFields.transform_position, transform.position);
    data.Add((int) SaveLoadFields.transform_scale, transform.localScale);
    data.Add((int) SaveLoadFields.bounds, bounds);
    return data;
}
}

```

Code 3.2.2 Autogenerated Code for example 3.1.1.

There are two complications with this system: The first complication of this system is serializing references to other Unity Components. The solution to this requires a deeper understanding of the system and so it is delayed until the next section. The second complication is serializing container types.

In the case of container types, the solution is essentially to autogenerate the entry in the enum, but provide a hook into the `load_data` and `save_data` functions which the programmer could implement themselves. It is necessary for the programmer to implement containers themselves because such a container could be arbitrarily complex and have special requirements that a metaprogram could not possibly know about.

An example of this API is given below in the `ImpulsePlayerGun` class: The programmer marks the field with a `SaveLoadOverrideAttribute` instead of a `SaveLoadAttribute`. This creates helper functions `load_data_()`, and `save_data_()`, which generate the same functions as above, skipping the container types.

The programmer must then complete the `ISaveLoadable` by implementing `load_data` and `save_data` themselves.

Code 3.2.3. shows a class which makes use of serializing a container.

```

[GenerateSaveLoadOverride]
public partial class ImpulsePlayerGun: MonoBehaviour
{
    public GameObject[] missile_types;
    //List type for serialization
    [SerializeField]
    [SaveLoadAttribute]
    public List<MissileType> ammo;
    [HideInInspector]
    [SaveLoadAttribute]
    public bool missile_check = true;
}

```

```

[SaveLoadAttribute(false)]
bool has_previously_loaded = false;

//gameplay code omitted, see appendix

public virtual void load_data(Dictionary<int, object> data)
{
    load_data_(data);
    UIManager.clear_ammo();
    for (int i = ammo.Count - 1; 0 <= i; --i)
    {
        UIManager.add_ammo(ammo[i]);
    }
}

public virtual Dictionary<int, object> save_data()
{
    return save_data_();
}
}

```

Code 3.2.3 The ImpulsePlayerGun class, which uses custom saving and loading on a field and a container

In the case of this class, there are two notable elements. The `has_previously_loaded` flag is added to allow the loader to tell when an object is truly being instantiated from the players perspective, and when it is simply being reloaded. (The “false” option in `SaveLoadAttribute` will be explained later). Ammo must be loaded manually because it is a container type and also because the game’s UI must update when this happens, since it clears all data on reload.

Autogenerated code for this example is analogous to the 3.1.2, and can be found in the appendix.

3.3. Aggregating Serialized Components into GameObjects

Unity GameObjects have lists of components, not just one. In order for serialization to work, the components on a GameObject must be aggregated. To accomplish this, there is a single component, `SlotMapReference`, on every GameObject whose responsibility it is to call save and load on each `ISaveLoadable` component, and aggregate the results into a `List<Dictionary<int, object>>`. The order of the list is the same as the order of the `ISaveLoadable` components on the GameObject, so it is not necessary to keep track of which is which. Hence, GameObjects are serialized as Lists of Dictionaries from ints to objects, which are placed into a `SlotMap`.

What follows is the `SlotMapReference` code as well as the relevant parts `SaveLoadManager` needed for understanding (Code 3.3.1, Code 3.3.2)

```

[System.Serializable]
public struct SaveLoadKey
{
    public SlotKey object_id;
}

```

```

public int component_id;

public static SaveLoadKey mk_null()
{
    return new SaveLoadKey() {component_id = -1};
}

public bool is_null() { return component_id == -1;}
}

[System.Serializable]
public class SaveLoadValue
{
    public ISaveLoadable[] component_map = null;
    public PrefabType type_to_construct = (PrefabType) (-1);
    public GameObject object_reference = null;
}
public class SaveLoadManager
{
    // "live" gameobjects that have registered
    // these objects will be saved when a snapshot is created
    public static SlotMap<SaveLoadValue> sprite_refs =
        new SlotMap<SaveLoadValue>();
    public static SlotMap<PrefabType> sprite_refs_indices =
        new SlotMap<PrefabType>();

    // a local save created
    // by messaging each ISaveLoadable registered in the save_data list
    private static Dictionary<SaveLoadKey, Dictionary<int, object>>
        snapshot = new Dictionary<SaveLoadKey, Dictionary<int, object>>();

    // remainder of class to be explained later
}

```

Code 3.3.1 Basic SaveLoadManager Structure

```

[GenerateSaveLoadAttribute]
public partial class SlotMapReference: MonoBehaviour
{
    public PrefabType prefab_type;

    [SaveLoadAttribute(false)]
    [HideInInspector]
    public bool already_registered = false;

    public bool decrement_slotmap = true;

    void Start()
    {
        if (already_registered) return;

        ISaveLoadable[] s = GetComponents<ISaveLoadable>();
        SlotKey k = SaveLoadManager.sprite_refs.add(new SaveLoadValue()
        {
            type_to_construct = prefab_type,
            object_reference = gameObject,
            component_map = s
        });
        for (int i = 0; i < s.Length; ++i)
        {
            s[i].save_load_id = new SaveLoadKey()

```

```

        {
            object_id = k,
            component_id = i
        };
    }

    already_registered = true;
}

void OnDestroy()
{
    if (decrement_slotmap)
        SaveLoadManager.sprite_refs.try_remove(save_load_id.object_id);
}
}

```

Code 3.3.2 Definition and implementation of the bootstrapper for component serialization.

On the first run, all objects are serialized and added to the SlotMap as described.

3.4. Saving and Loading

As GameObjects with ISaveLoadable components are spawned into the scene, their serialization component registers the object into the SlotMap. When the player hits save, all of the registered GameObjects have their lists of dictionaries saved into an additional SlotMap which contains the same indices and generations.

When the player presses load, all GameObjects marked for serialization are destroyed (or returned to a pool). The SlotMap indices, generations, and lists of GameObjects from the time of saving are used to reconstruct objects with the same components, and the SlotMap is restored to its previous state.

Because the generations and indices are restored, any key into the SlotMap that was valid at the time of saving is valid after a load. The standard SlotMap behavior of incrementing the generation is not used here, because the aim is to recreate the references as they were at the time of saving.

This answers the question of how references to other components are serialized: They are kept as pairs of keys into the SlotMap together with indices into the list of dictionaries. Since the list of dictionaries has the same structure as the list of ISaveLoadable components on a GameObject, and SlotMap keys were arranged to persist after loading, this structure will be stable after a save and a load, even though the objects they refer to may be at different locations in memory.

It is now possible to review the full listing of the SaveLoadManager (Code 3.4.1,2).

```

public class SaveLoadManager
{
    //"live" gameobjects that have registered
    //these objects will be saved when a snapshot is created

```



```

public static SlotMap<SaveLoadValue> sprite_refs =
    new SlotMap<SaveLoadValue>();
public static SlotMap<PrefabType> sprite_refs_indices =
    new SlotMap<PrefabType>();

//a local save created
//by messaging each ISaveLoadable registered in the save_data list
private static Dictionary<SaveLoadKey, Dictionary<int, object>>
    snapshot = new Dictionary<SaveLoadKey, Dictionary<int, object>>();

public static ISaveLoadable get_component(SaveLoadKey k)
{
    SaveLoadValue v = new SaveLoadValue();
    if(!sprite_refs.try_get(k.object_id, out v))
        Debug.Log($"invalid object_id");
    return v.component_map[k.component_id];
}
//...

```

Code 3.4.1 Fields and helper functions for SaveLoadManager

```

//..continued from 3.4.1
public static void create_snapshot()
{
    snapshot.Clear();

    //part 1
    for (int i = 0; i < sprite_refs.count; ++i)
    {
        for (int j = 0; j < sprite_refs[i].component_map.Length; ++j)
        {
            ISaveLoadable s = sprite_refs[i].component_map[j];
            snapshot.Add(s.save_load_id, s.save_data());
        }
    }

    //part 2
    sprite_refs_indices.copy_indices(sprite_refs);
    for (int i = 0; i < sprite_refs.count; ++i)
    {
        sprite_refs_indices[i] = sprite_refs[i].type_to_construct;
    }
}
//..

```

Code 3.4.2 SaveLoadMangeer.create_snapshot()

For creating the snapshot, first the function clears any old data from a previous quicksave. This could be replaced with a list of saves if multiple save slots are desired, but a single save suffices to show the technology.

In part 1, the function loops through every object reference (`sprite_ref`) and grabs its list of components. It then runs through the `ISaveLoadable` interface on each component and aggregates the resulting (`field_name`, `field_value`) dictionaries into a list. This constitutes a

representation of that object at that time. This is then stored in a larger dictionary of type (save_load_id -> object state).

In part 2, the function creates a memory of the SlotMap state, with its generational indices and internal structure. When the objects are recreated, the internal representation of the SlotMap should be the same, so that references between recreated objects can indirect through the SlotMap and not point at old data. In addition to remembering the index data, the code also remembers what kind of object to construct, as we need to know what to attach to the GameObject and which pool to grab it from if pools are used.

Code 3.4.2 shows the listing for applying a snapshot, with comments dividing the main parts of the function. This works analogously:

```
//...continued from 3.4.2
public static void apply_snapshot()
{
    //handle load before save
    if (sprite_refs_indices.count == 0) return;

    //clear delegates in physics system
    //(on initialization all physics objects will reregister)
    PhysicsHandler.refresh_delegates();

    //part 1
    for (int i = 0; i < sprite_refs.count; ++i)
    {
        SaveLoadableAssetProvider.return_to_pool
        (
            sprite_refs[i].object_reference,
            sprite_refs[i].type_to_construct
        );
    }

    sprite_refs.copy_indices(sprite_refs_indices);

    //part 2
    for (int i = 0; i < sprite_refs.count; ++i)
    {
        PrefabType asset_type = sprite_refs_indices[i];
        GameObject game_object =
            SaveLoadableAssetProvider.create_from_pool(asset_type);
        sprite_refs[i] = new SaveLoadValue()
        {
            type_to_construct = asset_type,
            object_reference = game_object,
            component_map = game_object.GetComponents<ISaveLoadable>()
        };
    }

    //part 3
    foreach (var s in snapshot)
    {
        try
        {
            sprite_refs[s.Key.object_id]
```

```

        .component_map[s.Key.component_id].load_data(s.Value);
    }
    catch (System.Exception e)
    {
        Debug.Log
        (
            $"unable to load type
            {sprite_refs[s.Key.object_id].type_to_construct} in
            {sprite_refs[s.Key.object_id].object_reference.name} with
            {sprite_refs[s.Key.object_id]
                .component_map[s.Key.component_id]}
            for slotmap reconstruction : {e.Message}, {e.StackTrace}"
        );
    }
}
}
}
}

```

Code 4.3.3 SaveLoadManager.apply_snapshot()

Part 1 destroys or returns all ISaveLoadables to a pool, as the code will be recreating them from the save data. Part 2 gets the ISaveLoadables from the pool, in the correct type, in the correct slot for the SlotMap. At this point, the objects are created but are not initialized with the right variables. Part 3 then goes into each component and runs load_data.

3.5. Unity Prefab Serialization

For users to make their own levels, it will be necessary to work at a different level of granularity. Rather than expose components user-facing code must work with complete prefabs and share only those fields which they should be able to edit. Furthermore, many of the fields must be translated into human-readable names.

For this purpose, the ISaveLoadGenerator metaprogram also creates a function called GetParamNameMapping. All this does is create a dictionary between the user-facing and developer-facing names of relevant fields on the component to be saved. It also provides for free the correct ordering as it appears in the corresponding generated enum.

To accomplish this, define a class ParamNameMapping as follows:

```

public class ParamNameMapping //constructor omitted for brevity
{
    public string component_field_name;
    public string component_field_type;
    public string prefab_field_name;
    public int component_field_index;
}

```

Code 3.5.1 Definition of ParamNameMapping

The following method is then added to the ISaveLoadable interface. (the CameraArea class from Code 3.2.1 is used as an example)

```
public List<ParamNameMapping> get_param_name_mappings()
{
    List<ParamNameMapping> param_name_mappings =
        new List<ParamNameMapping>(1);
    param_name_mappings.Add
    (
        new ParamNameMapping
        (
            "bounds",
            "UnityEngine.Vector2",
            "bounds",
            4
        )
    );
    return param_name_mappings;
}
```

Code 3.5.2 Example of autogenerated get_param_name_mappings() function for the CameraArea class (Code 3.2.1)

It is also necessary to translate between the SlotMap serialization of Unity GameObjects (i.e. lists of components) and the user-facing serialization format that will be read in by a level editor. For this the code uses a third-party JSON serializer and a metaprogram to translate between the two formats.

The following metaprogram searches every prefab in the project and creates a corresponding ScriptableObject which implements the interface IPrefabSerializer. The usage of this class is essentially the same as described in the section on Scriptable Objects, it acts merely as a bridge between incoming JSON data and the object which it mirrors. Each IPrefabSerializer is easily encodable and decodable as it is just plain old data and can be plugged into a third-party JSON serialization library.

```
public class ShootOnMissileSerializer : IPrefabSerializer
{
    public PrefabType get_prefab_type()
    {
        return PrefabType.shoot_on_missile;
    }
    UnityEngine.GameObject missile_to_launch;

    public Dictionary<int,object>[] get_component_snapshots()
    {
        Dictionary<int,object>[] component_snapshots =
            new Dictionary<int,object>[2];
        component_snapshots[0] =
            new Dictionary<int,object>();
        if (missile_to_launch != null)
            component_snapshots[0][4] = missile_to_launch;
        component_snapshots[1] = new Dictionary<int,object>();
    }
}
```

```
        return component_snapshots;
    }
}
```

Code 3.5.3 Example of an IPrefabSerializer generated by the metaprogram. The naming convention for generated code is to append the Serializer suffix.

Note this corresponds to an entire prefab: first indices in `component_snapshot` refer to the components and the second indices refer to the serialized fields. Which fields to load come from the metaprogram through the attribute system.

It is also now possible to describe the purpose of the inputs for `SaveLoadAttribute`. A `SaveLoadAttribute` marked `false` (as seen with `has_already_loaded`) designates a variable which is serialized but which should not be seen by the level editor. That is, users should not be able to configure an object from the level editor with that field modified. Therefore, the optional Boolean parameter to `SaveLoadAttribute` tells this metaprogram not to include it in the translation to JSON. Likewise, the optional string parameter sets the user-facing name seen in the JSON file.

Since both the `IPrefabSerializer` and the `ISaveLoadable` encode information as a `Dictionary<int, object>`, once data is read from a level file and interpreted from a JSON file, it can simply be placed inside a `SlotMap`, and the internals of the serialization system works the same way as loading state.

The only thing that then remains then is a mechanism by which the level loader can populate a `GameObject` “from scratch” into the `SlotMap`. So far, the save load system only gets lists of components from `SlotMapReference`, which means such an object must have already existed in the scene for the serializer to know about it. What is needed is a way of putting every possible prefab into a list that the `SaveLoadManager` can access. When loading from JSON and a level file, the serializer can then opt to grab the default object from a list of prefabs instead of from a save.

To make this work, Impulse created a metaprogram `IPrefabSerializerGenerator`. The output of this program consists of two things. First, a giant scriptable object which serves as a reference container for every possible prefab. Second, another autogenerated metaprogram, which will populate the references when run and reloads the assembly.

One may ask why it is necessary to have two layers of metaprograms (a metaprogram generating another metaprogram). Would it not be possible just to build the `ScriptableObject` and populate it in one pass? It turns out that a two-step pass is necessary because until the first metaprogram is done running, the `ScriptableObject` class does not yet exist in code. The project must be recompiled in order for it to exist, which means a second pass referencing the `ScriptableObject` code is needed. This can be autogenerated, hence the additional layer of code generation. Code 3.5.4 shows an example output:

```

public static class PrefabInfoFieldFiller
{
    [MenuItem("Generate/PrefabSerializerLinks")]
    public static void generate_prefab_serializer_links()
    {
        List<GameObject> prefabs = new List<GameObject>();
        string[] guid = AssetDatabase.FindAssets
        (
            "t: Prefab",
            new string[1] { "Assets/MyAssets/Prefabs" }
        );
        foreach (string id in guid)
        {
            string asset_path = AssetDatabase.GUIDToAssetPath(id);
            GameObject prefab = (GameObject)AssetDatabase.LoadAssetAtPath
            (
                asset_path,
                typeof(GameObject)
            );
            prefabs.Add(prefab);
        }
        PrefabInfo prefab_info_instance =
            ScriptableObject.CreateInstance<PrefabInfo>();
        prefab_info_instance.first_prefab_name = prefabs[1];
        prefab_info_instance.second_prefab_name = prefabs[2];
        //...code omitted here. where, nth_class_name is the
        //verbatim name of the unity prefab being serialized

        AssetDatabase.CreateAsset
        (
            prefab_info_instance,
            "Assets/MyAssets/Resources/PrefabInfoAsset.asset"
        );
        AssetDatabase.Refresh();
    }
}

```

Code 3.5.4 Reference filling metaprogram. This metaprogram is not user-written, and is generated at the same time as the PrefabInfo object by IPrefabSerializerGenerator

With these considerations covered, serialization now functions within saves and loads, across new asset creation, and between live game data and level loading via JSON files.

3.6. Parsing Level Geometry

With serialization taken care of, the setup for the level editor is quite straightforward. Users will specify four bitmap files and two JSON files. The first two bitmaps will contain the levels visuals. The third bitmap will contain "sprites" (enemies, items, etc.). The fourth bitmap will contain level geometry.

The format is as follows: for a level of size $N \cdot M$ the corresponding bitmap files will be of size $kN \cdot kM$. For geometry files, $k = 1$, as tiles must be placed on integer coordinates. For sprite files, k can be any positive integer, and this determines the granularity of possible positions in the level.

For the “geometry” and “sprite” bitmaps, colors map to level tiles and configurations of prefabs respectively. This is determined in the JSON file. For “visuals” bitmaps, the image is loaded directly into the game.

The geometry file is parsed first. As the section title suggests, most of the subtlety comes from this section, which will be addressed momentarily.

After the geometry file is parsed, the code compares the pixel ratio of each sprite image to the geometry image in order to determine k (the granularity of object placement). Sprite placement is essentially trivial, because every pixel corresponds to a unique object in the game world, so the density of "filled in" pixels will always be very low.

Finally, the visuals file is loaded directly into the game. This may raise some eyebrows due to the potential texture size. However, this is also not an issue for game-specific reasons: A typical level is no bigger than 100x100 units, and a 16-bit art style is used for official levels. So, if almost all of the visuals of the level occupy a 1600x1600 texture, there is really nothing to be concerned about. If need be, the ratio of the geometry image to the visuals image could be used to calculate pixels-per-unit, and the visuals texture could then be sliced into 1-unit pieces and scanned for duplicates with a simple hash function. So far this has not been necessary even on very large levels.

3.7. Partitioning a rectangular array into subarrays of constant value

With the visual file taken care of, one can address the most complicated issue of level loading. In the case of the level geometry file, one does not want to be spawning a unique GameObject for every "solid" 1x1 pixel in the bitmap, as level geometry will generally be made up of large contiguous regions. There are a small number of potential ground tiles since visuals and gameplay have been handled separately. Due to the mechanics of the game, most solid regions will also consist of large squares or rectangles. For this reason, such rectangles ought to be detected and treated as a single object. This section provides exposition on existing research on the “rectangular partition problem” (see [7] for reference). The follow-up section, 3.7.1, provides a new simplified solution.

Mathematically, the problem is straightforward. The bitmap A can be thought of as an $N \times M$ matrix $A_{i,j}$ with entries in some small finite set $\{1, \dots, n\}$ (one for every type of ground tile). One would like a "fast" way of partitioning this matrix into blocks which have constant value, such that the number of blocks is minimized, or at least kept reasonably small.

First consider the 2-color case. One can visualize the problem by imagining incrementally adding "cuts" until all the resulting shapes are rectangles, as in figure 3.7.1. The question then becomes selecting cuts judiciously one at a time.

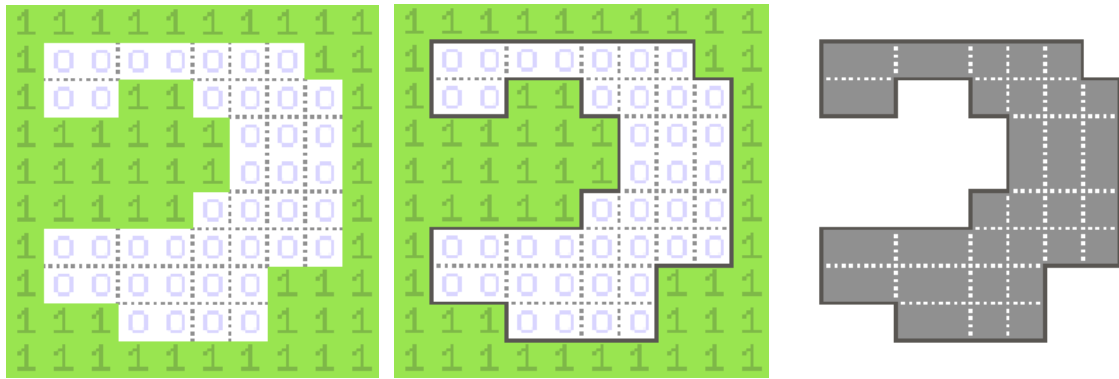


Figure 3.7.1 Cutting an axis-aligned shape into rectangles. The dotted lines represent possible cuts.

For the moment let us ignore the subtlety of intersecting cuts. In that case, there are two ways a partition could fail to be minimal, shown in figure 3.7.2. The first is adding cuts that are not necessary. That would mean that a cut did not meet any corners. The second way a partition could fail to be minimal is if one selects a corner and cuts along an axis without a matching corner, when the other axis allowed a cut that did have a matching corner. That is to say, one selected a cut which met one corner, when one could have selected a cut which matched two.

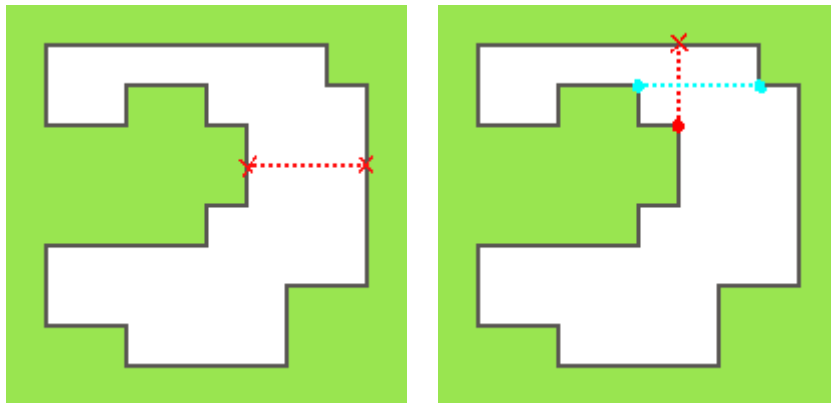


Figure 3.7.2 Maximizing number of corners per edge

All of the complexity of the problem comes into play because of intersections between cuts. Consider the following configuration.

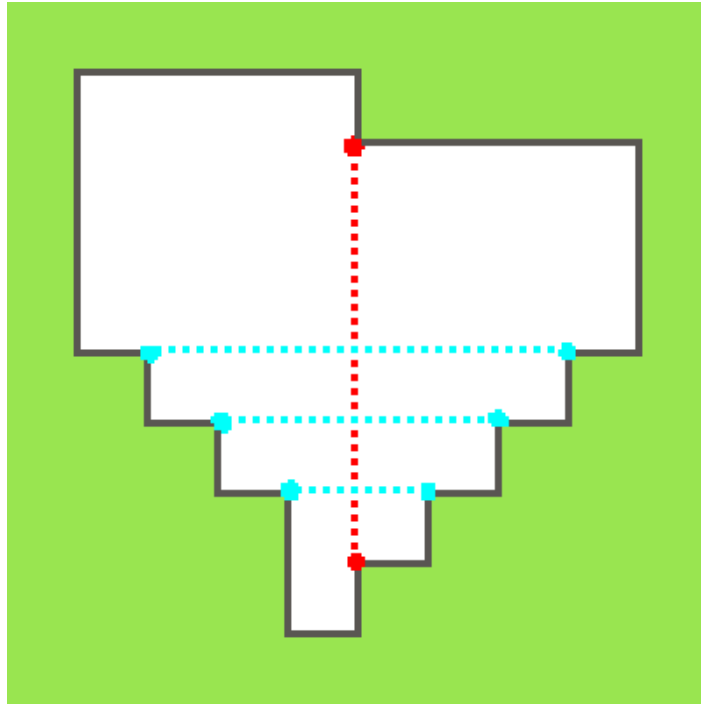


Figure 3.7.3 Nonlocality of Squares

Even though the vertical cut connects two corners, it is suboptimal, because it "obstructs" other pairs of corners. So, decisions about which direction to choose for a cut from any corner are extremely non-local. They depend on all other corners which are "visible" from a candidate cut, which is in turn dependent on all previous cuts. Here, cutting only horizontally would result in 6 squares, but including the cut in red would result in at least 8.

This is a well-known problem in computational geometry, and an exact solution exists [7]. The basic idea is to find as many two-corner cuts that "don't see each other" as possible, make cuts between them, then draw cuts arbitrarily from the remaining corners. The hard part of the problem is then making as many two-corner cuts as possible such that no two-corner cuts intersect.

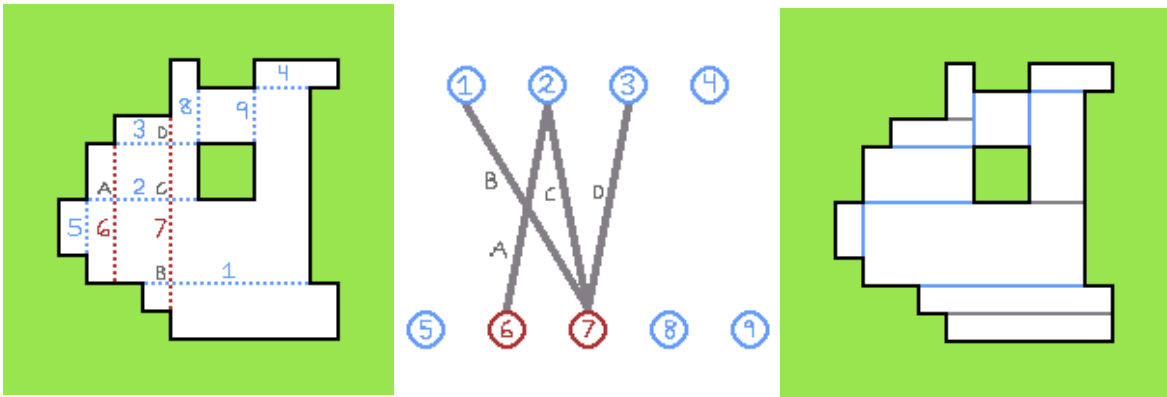


Figure 3.7.4 Solution after finding maximum independent set

Finding edges that “don’t see each other” works as follows: First, mark each possible cut which meets at two corners. Next, define the intersection graph of this configuration by creating a vertex for each cut and an edge between two vertices if the corresponding cuts intersect. The problem of avoiding “visible cuts” may then be restated as finding a maximum independent set of this graph: that is, a maximal set of vertices such that no two vertices are joined by an edge.

Figure 3.7.4 shows a typical example. The key insight here is that the intersection graph is bipartite, since horizontal and vertical cuts will never intersect. Finding a vertex cover for a generic graph is NP hard, but for bipartite graphs, polynomial time algorithms exist.

Finding a generic solution which is better than this would be very hard, as the problem is well studied, even more so intersection graphs. In fact, it is a theorem that every planar bipartite graph corresponds to a crossing of horizontal and vertical line segments in exactly this way.

There is good news. This approach is unnecessarily complicated for the stated use case, because having a strictly optimal solution is not necessary. The feature of the level geometry which causes the graph to grow is large numbers of axis aligned corners, and from a gameplay perspective one does not expect to see a high density of corners within a level. Furthermore, the cost of introducing a few extra ground tiles into the level does not warrant the conceptual overhead of this approach.

Given this information, Impulse opts for the following solution: scan along a single axis (say, left to right, top to bottom) and make a cut every time an edge is found. Always cut along the scanning direction. It is guaranteed that this is at least a local minimum, in the sense that removing a cut would result in nonrectangular shapes.

3.7.1. Implementation

This section presents an original algorithm for solving this problem, which takes $\theta(NM \log(M))$ time and $\theta(M)$ space, where all the constants involved are very small.

Without loss of generality, one can always operate on bitmaps for which $M \leq N$, in which case the algorithm runs in $\theta(n \log(\sqrt{n}))$ time and $\theta(\sqrt{n})$ space respectively, where n is the number of pixels. Pixels correspond to square units in the level, so one can think of n as level size.

The general idea is to follow the same methodology that a human might, given this task. The array is parsed like a CRT monitor, "scanline by scanline", creating as wide of rectangles as possible. If a constant interval on the row below matches up with an interval on the row above, these are combined. This is done greedily, not worrying about more than two successive scanlines at once.

First, define a RangeInfo as a struct containing a leftmost index, a rightmost index, an upmost index, and a value. The purpose of this struct is to keep track of candidate rectangles, and a list of these structs is kept for each of the two scanlines being considered, with the left and right fields creating a partition of the scanline for each list.

Start by initializing the two arrays of scanlines, prev and current. These two arrays are never longer than M since they serve to partition the scanline, but will often be shorter. `p1` and `c1` will be used to denote the "true" length of each array.

```
public static void to_squares(int[,] A, ref List<Rectangle>[] rectangles)
{
    //"prev length", number of initialized entries in the prev array
    int p1 = 0;
    //partition of previous "scanline" by horizontal axis into RangeInfos
    //prev[0], ..., prev[p1]
    RangeInfo[] prev = new RangeInfo[A.GetLength(1)];
    //partition of current "scanline" by horizontal axis into RangeInfos
    //current[0], ..., current[c1]
    RangeInfo[] current = new RangeInfo[A.GetLength(1)];
}
```

Code 3.7.1.1 to_squares, part 1

Next, the topmost scanline is processed into prev. This is trivial: just grow the current RangeInfo rightward if the next index has the same value, otherwise create another RangeInfo. None of these RangeInfos is marked as added since they could continue to grow downwards.

```
prev[0] = new RangeInfo()
{
    l = 0, r = 0, u = 0, v = A[0, 0], added = false
};
for (int j = 1; j < A.GetLength(1); ++j)
{
```

```

if (A[0, j] == A[0, j - 1])
{
    ++prev[p1].r;
}
else
{
    prev[++p1] = new RangeInfo()
    {
        l = j, r = j, u = 0, v = A[0, j], added = false
    };
}
}

```

Code 3.7.1.2 to_squares, part 2

From the second scanline onward, the code begins analyzing scanlines in pairs, checking what is above to detect finalized RangeInfos. At the beginning of each new line the code sets `cl = 0` and starts using `current` as if it were empty.

```

for (int i = 1; i < A.GetLength(0); ++i)
{
    // "current length", number of initialized entries in the current array
    int cl = 0;
    current[0] = new RangeInfo() { l = 0, r = 0, u = i, v = A[i, 0] };
}

```

Code 3.7.1.3 to_squares, part 3

In the leftmost entry of each line `A[i, 0]`, the code always creates a new `RangeInfo` into `current`. For joining with the previous scanline only need to check if `prev[0].v == current[0].v` since the above `RangeInfo` cannot extend further leftwards and must be at index 0. If the values do match, processing continues. If they do not, the previous `RangeInfo` is finalized, and the code can add it to the list of `Rectangles`.

```

if (prev[0].v == current[0].v && prev[0].l == current[0].l)
{
    current[0].u = prev[0].u;
}
else
{
    rectangles[prev[0].v].Add
    (
        new Rectangle()
        {
            l = prev[0].l, r = prev[0].r, u = prev[0].u, d = i - 1
        }
    );
    prev[0].added = true;
}

```

Code 3.7.1.4 to_squares, part 4

For entries `A[i, j]` in the interior of the matrix, processing is slightly more complicated. First the code must determine which index into `prev` contains the `RangeInfo` for which `l <= j <= r`. That is, which `RangeInfo` is "above" `A[i, j]`. This essentially just amounts to a binary search.

```

for (int j = 1; j < A.GetLength(1); ++j)
{
    int above = entry_containing(prev, pl + 1, j);

```

Code 3.7.1.5 to_squares part 5

```

static int entry_containing(RangeInfo[] A, int pl, int j)
{
    int l = 0;
    int r = pl;
    int i = (r + 1) / 2;
    while (A[i].l != j && l < r)
    {
        if (j < A[i].l) r = i;
        else l = i + 1;
        i = (r + 1) / 2;
    }
    if (A[i].l != j) --i;
    return i;
}

```

Code 3.7.1.6 helper function, entry_containing

Next the code grows the range rightward if the values on the current scanline do not change just as before. However, if they differ the code must check for the following configuration.

$$\begin{bmatrix} x & x \\ x & y \end{bmatrix}$$

In this case, the RangeInfo to the left is distinct from the RangeInfo above, and so its *u* value must be reset to *i*.

```

if (A[i, j] == A[i, j - 1])
{
    ++current[cl].r;
}
else
{
    //discovered shape
    // 1a 1a
    // 1b 0c, mark 1b as separate square, so u = i
    // then create new rangeinfo on this scanline for 0c
    if (prev[above].v == A[i, j - 1]) current[cl].u = i;
    current[++cl] = new RangeInfo()
    {
        l = j, r = j, u = i, v = A[i, j]
    };
}

```

Code 3.7.1.7 to_squares part 6

Now the code must process the RangeInfo above us if it has not yet been finalized. The code first checks for the following shape.

$$\begin{bmatrix} x & \dots & x & \dots & x \\ x & \dots & x & ? & ? \end{bmatrix}$$

where, as far as the code can tell, the RangeInfo at this position corresponds to the same Rectangle as the RangeInfo above. In this case, set `u` equal to the RangeInfo above us and do not mark either as finalized. It may later be the case that the previous pattern is discovered, in which case `u` will be corrected.

```

if (!prev[above].added)
{
  if (prev[above].v == current[cl].v
      && prev[above].l == current[cl].l)
  {
    //discovered shape
    // ... | 1a ...      ... 1a
    // ... | 1b ... 1b ... ?
    //as far as we can tell this is the same square,
    //so treat it as such.
    //if we find out later on this scanline that we have
    // 1a 1a
    // 1b 0c we can correct it
    current[cl].u = prev[above].u;
  }
}

```

Code 3.7.1.8 to_squares, part 7

If this shape is not discovered, the above RangeInfo is known to be distinct and can be finalized. However, there is one more special case to check against

$$\begin{bmatrix} x & y \\ x & x \end{bmatrix}$$

In this case, also finalize the RangeInfo diagonally above and to the left. Furthermore, set the current RangeInfo's up index to `i`.

```

else
{
  //since the above range has a different value or leftmost index
  //it must be part of a distinct square from this one
  //so we can add it and mark it as finalized
  rectangles[prev[above].v].Add(
    new Rectangle()
    {
      l = prev[above].l, r = prev[above].r,
      u = prev[above].u, d = i - 1
    }
  );
  prev[above].added = true;

  //added as a special case for the following scenario
  //1a 0b
  //1c 1c
  //need to complete the upper left square.
  //the square to complete will always be prev[above-1]
  //and if the first two conditions are true,
  //prev[above-1] will always exist
  if (A[i - 1, j - 1] == current[cl].v &&
      prev[above].v != current[cl].v && !prev[above - 1].added)
  {

```


4. Results

4.1. Save State

The following dropdown option in Unity generates the necessary interfaces discussed in this paper.

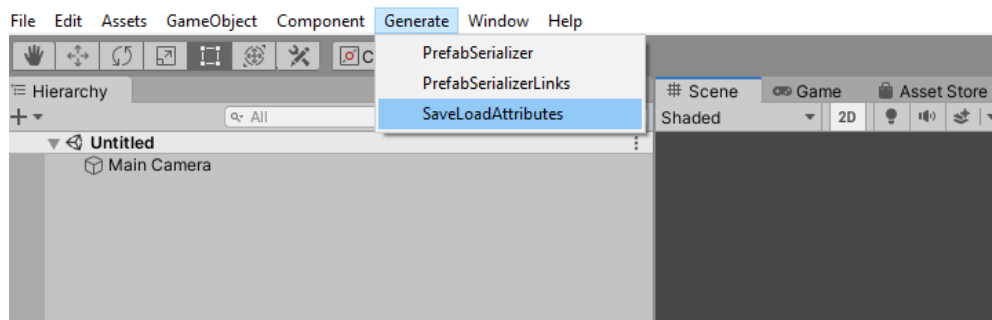


Figure 4.1.1 Code Generation Options

These three options are to be run in sequence. The first creates the `ISaveLoadable` interface for the `SaveLoadAttributes`. Figure 4.1.2. shows a code snippet of the generated program.

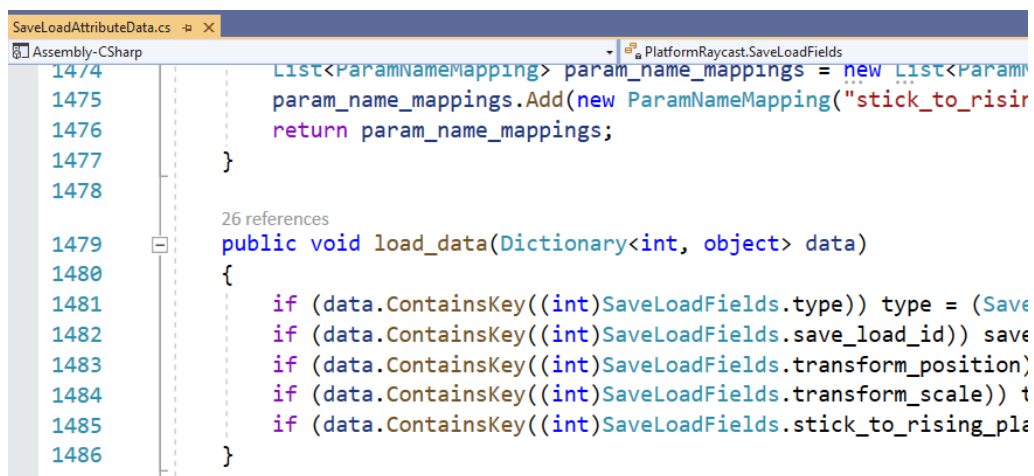


Figure 4.1.2. Output of SaveLoadAttributes Generator

The `PrefabSerializer` option creates the necessary links for prefab serialization to disc and for reconstructing a generic version into the `SlotMap`. It also creates the metaprogram for the third option. Figure 4.1.3. shows a code snippet.


```
PrefabSerializerData.cs x
Assembly-CSharp PrefabType
52 references
961 public Dictionary<int,object>[] get_component_snapshots()
962 {
963     Dictionary<int,object>[] component_snapshots = new Dictionary<
964     component_snapshots[0] = new Dictionary<int,object>();
965     if (follow_x != null) component_snapshots[0][4] = follow_x;
966     if (follow_y != null) component_snapshots[0][5] = follow_y;
967     if (screen_x != null) component_snapshots[0][6] = screen_x;
968     if (screen_y != null) component_snapshots[0][7] = screen_y;
969     if (camera_distance != null) component_snapshots[0][8] = camera_distance;
```

Figure 4.1.3. Output of PrefabSerializer Generator

The third option simply builds the ScriptableObject with a reference to each prefab asset and links the references together. Code for this is not shown since its output is dependent on a metaprogram which is itself generated in the same file as figure 4.1.3.

The save and load system works as one would expect within the game. The player presses save at some point in the level, and then presses load. This process works even if the player restarts the level. This is shown in figures 4.1.4 through 4.1.6.

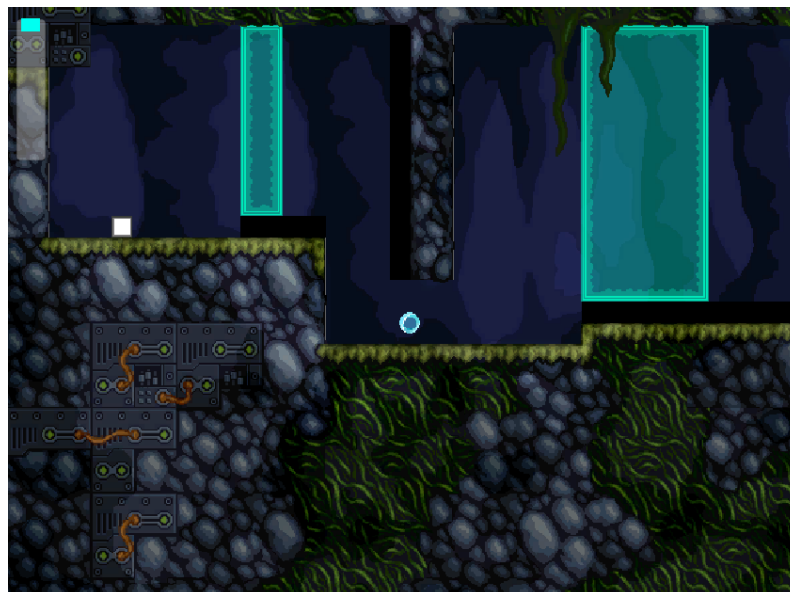


Figure 4.1.4 Player hits save



Figure 4.1.5. Player moves through level and creates new objects

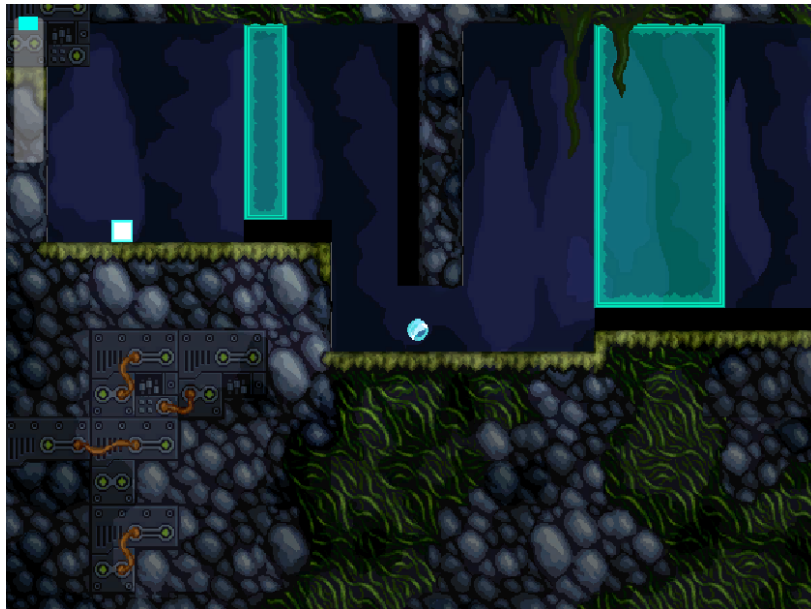


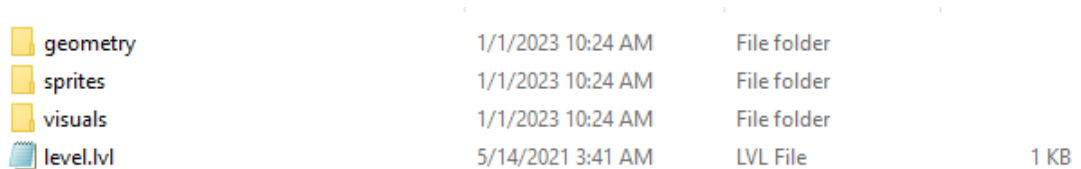
Figure 4.1.6 Player loads successfully

Take note that this is merely a demonstration of the core functionality. It does not handle, for example, saving on one level and then loading on another. These are all extremely simple cases to handle with gameplay logic, and not part of the serialization and deserialization system. Similarly, for multiple save slots, one would simply need to keep track of more than one configuration of SlotMap indices and set of dictionaries. Finally, this form of save is only transient as long as the application is open. This is the intended gameplay mechanic, since saves are just meant to practice in a given room, not to hold all player progress. To

keep saves between sessions, one would need to additionally write out the relevant data to disc.

4.2. Level Editor

Levels can be loaded in as follows. First a folder is constructed for each level as in figure 4.2.1. Each level consists of a .lvl file and a geometry, sprites, and visuals subfolder.



geometry	1/1/2023 10:24 AM	File folder	
sprites	1/1/2023 10:24 AM	File folder	
visuals	1/1/2023 10:24 AM	File folder	
level.lvl	5/14/2021 3:41 AM	LVL File	1 KB

Figure 4.2.1 folder structure for a level

The geometry folder contains png files, all of the same size, named layer#.png. These contain data for where the bounds for interaction should be. Every pixel corresponds to one 1x1 unit in the game. Figure 4.2.2. gives a simple example.



Figure 4.2.2. Contents of geometry file

The sprites folder contains .png files of a size which is a multiple of the resolution of the geometry files, as described in section 3. The files are named according to the same layer# convention. Any typical example is hard to see in a figure since single pixels of color correspond to , and hence most of the image will be blank. Figure 4.2.3. shows an example with a single pixel filled in for the player location.

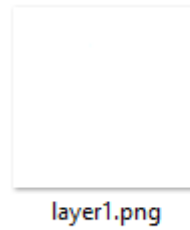


Figure 4.2.3. Contents of sprites file. Note there is a single pixel colored in for the player

The visuals folder contains .png files, all of the same size, all multiples of the resolution of the files in the geometry folder. The naming convention is the same, and the contents of these files are drawn directly as sprites onto the screen with no physical interaction.



Figure 4.2.4. Contents of the geometry folder

The .lvl file contains the information which is used to parse the above data. Its basic structure is shown in figure 4.2.5. spr_colors and spr_prefabs are in one to one correspondence, as are obj_colors and obj_prefabs. These color values determine which rgb color in the image correspond to what object in the game. The JSON format allows for serialization of the various properties exposed to the user via the metaprograms described in this paper.

It is important to note that a template .lvl file will be made publicly available upon shipment of the game, since this would not need to be changed very much. Moreover, the setup of this level data exists in such a way that other tools could be made to generate the .lvl file and interoperate with a given image editor.

```

level.lv1 - Notepad
File Edit Format View Help
{
    "version":0,
    "layer_names":["layer1"],
    "layer_params":[{"position":{"x":1.0,"y":1.0,"z":1.0}}],
    "spr_colors":[{"r":0,"g":255,"b":255,"a":255}],
    "spr_prefabs":[{"$type":"ImpulsePlayerSerializer"}],
    "obj_colors":[{"r":0,"g":0,"b":0,"a":255}],
    "obj_prefabs":[{"$type":"Ground1Serializer"}]
}

```

Figure 4.2.5. Contents of the lv1 file. A more comprehensive file would be provided to users so that they do not need to change the JSON unless they wanted to.

Using this data, Impulse can load the information in from a blank Unity scene containing only the list of Prefabs and the level loading script. Figures 4.2.6 and 4.2.7 show the results from the examples given so far.

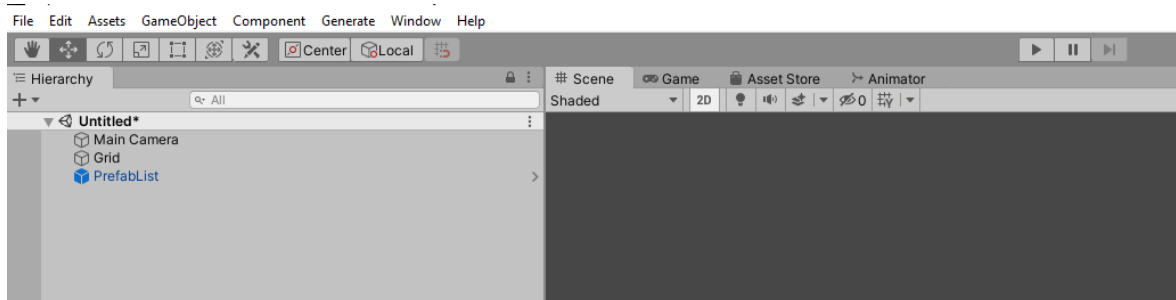


Figure 4.2.6. Minimum contents to load a scene. With a camera included in the sprites file even the main camera would not be needed.

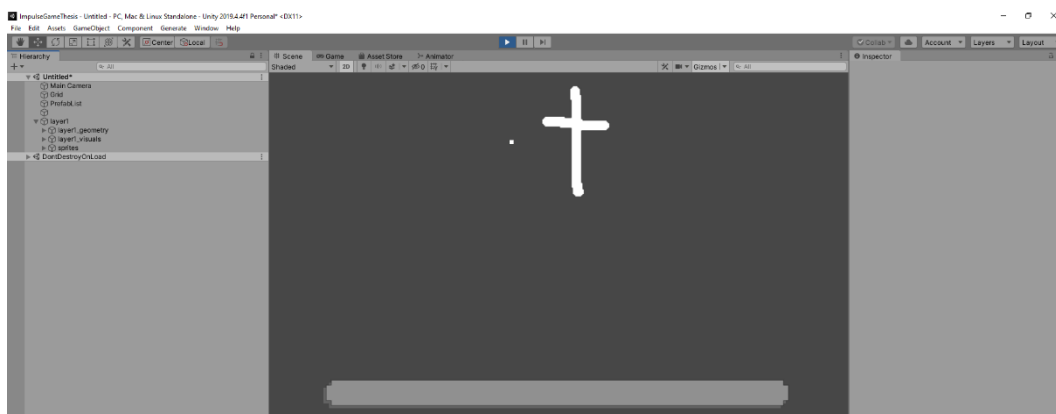


Figure 4.2.7. Pressing play builds all geometry, sprites, and levels directly from the images.

5. Discussion

There is a lot of conceptual overhead in setting up such an architecture, and with ECS being immediately available in Unity, one may ask what the point is. From a conceptual point of view, this system has proven easier to handle once it is up and running, because all future code can be written in a standard Unity style without worrying about data-oriented concerns. This is also true of Unity's hybrid ECS system, which converts Unity code to ECS automatically, but one then has to merely trust Unity to do all of the serialization correctly. If there is a serialization problem, it would be much more opaque.

The system described has a few drawbacks that one should be aware of. The first is that this is obviously less performant than Unity ECS. ECS is built with high level performance in mind and as such the Components are arranged in memory for maximum cache coherency. On the other hand, this system uses a large number of dictionaries and other such objects, so the access of data is not as linear. For a more high-tech game, one would have to weigh their options more, at least if the amount of serialized data was expected to be large. Because Impulse is a 2D sprite-based game, the cost of using dictionaries was not meaningful enough to impact performance.

For future iterations, it should be noted that the list of components system causes the bulk of the complexity. The code goes through the effort of working at the component level of granularity when designing prefabs, but only exposes a prefab level of granularity when saving, loading, or drawing from the editor. Building prefabs from components is convenient but may be more work than it is worth.

A cleaner solution using the same setup would be to insist that GameObjects may only have one custom MonoBehaviour and to disregard encapsulation principles by always working at the level of GameObjects. If code duplication is a serious concern, one could easily run a script that inserts common code whenever a given tag is seen (basically just at #define for C#). That way, custom MonoBehaviors and Prefabs would be in one-to-one correspondence, minimizing a lot of the conceptual and computational overhead. The general setup described in this paper would still be of use to such a system using essentially the same structure.

Architecture aside, there are also several quality-of-life elements one may wish to implement. First, a mechanism to have a single field for `has_previously_loaded` would obviously be preferable to doing this for each MonoBehaviour. This is easily achievable through e.g. a custom Unity event. However, this is a small detail not directly relevant to the main work, and so has not been included in the discussion.

One also has to be very careful about the SlotMapReference's enum containing the prefab to use as a base for reloading. This enum is merely a number, so changes in the ordering of the PrefabInfo ScriptableObject can invalidate these enums unless one arranges for this not to be the case. Other issues can arise from aliasing in field names and so on, but they are very easily detectable at compile time, since the generated code is intentionally very flat and simple.

Conclusion

Impulse was architected using three passes of metaprograms. The first serializes user-authored components and extends them to a saving and loading interface. The second serializes prefabs for instantiation, noting the components that they have on them, and builds the third autogenerated metaprogram. The third metaprogram attaches references from the prefabs to a ScriptableObject which allows the content to be instantiated in the saving and loading interface. All of this is supported using a SlotMap to keep track of the transience of objects.

The benefit of this setup is that save data can be automatically aggregated and stored merely by tagging gameplay relevant data with an attribute. Some care is needed in the case of containers or startup logic, and leniency is given in this case using the SaveLoadOverride attribute. Standard unity architecture does not allow for this level of control with serialization, and ECS (while powerful) introduces too much code footprint and conceptual overhead by comparison.

The primary limitation of this system is the disconnect between the granularity of components and prefabs. Because all objects must be instantiated from a prefab anyway, the utility of serializing multiple components on a GameObject may be unnecessary. An interesting future direction to explore would be a more procedural style in which all GameObjects have only one custom component, and code reuse is accomplished by some other method than lists of components.

The architecture described connected well with a level editor, using the same principles to serialize data from disc via a third-party JSON parser. The primary work consisted of translating from the JSON into the dictionary format understood by the SlotMap, and this required a translation layer for each prefab. Reading level geometry from an image was its own computer science problem, which was solved in a more simplistic way than the standard approach, but with great speed and reasonable efficiency. The user-facing benefits of the level design system allow anyone to simply draw geometry and sprite data in with an image editor.

As with any programming architecture, the goal is to reduce complexity and not add it, and any value added must be measured by the quality of the final product. The combination of these serialization techniques has allowed impulse the ability to quickly save and load data at high speeds. As the feature set of the game expands (e.g. through the addition of multiple gameplay layers), the system is able to easily adapt. Serializing new gameplay elements is a straightforward process. For both user and developer, the system meets its goal.

List of Abbreviations

ECS *Entity Component System*

List of Figures

Figure 2.1.1 Event Execution Order.....	3
Figure 2.6.1 Diagram of a Slotmap	8
Figure 3.7.1 Cutting an axis-aligned shape into rectangles.....	24
Figure 3.7.2 Maximizing number of corners per edge	24
Figure 3.7.3 Nonlocality of Squares.....	25
Figure 3.7.4 Solution after finding maximum independent set	26
Figure 4.1.1 Code Generation Options	32
Figure 4.1.2. Output of SaveLoadAttributes Generator.....	33
Figure 4.1.3. Output of PrefabSerializer Generator	33
Figure 4.1.4 Player hits save	33
Figure 4.1.5. Player moves through level and creates new objects.....	34
Figure 4.1.6 Player loads sucessfully	34
Figure 4.2.1 folder structure for a level	35
Figure 4.2.2. Contentes of geometry file	35
Figure 4.2.3. Contents of sprites file. Note there is a single pixel colored in for the player.....	36
Figure 4.2.4. Contents of the visuals file	36
Figure 4.2.5. Contents of the lvl file	37
Figure 4.2.6. Minimum contents to load a scene	37
Figure 4.2.7. Pressing play builds all geometry, sprites, and levels directly from the images.....	37

List of Codes

Code 3.2.1 A simplified CamerArea class tagged for serialization	11
Code 3.2.2 Autogenerated Code for example 3.1.1.	13
Code 3.2.3 The ImpulsePlayerGun class, using custom saving and loading on a container.....	14
Code 3.3.1 Basic SaveLoadManager Structure	15
Code 3.3.2 Definition and implementation of the bootstrapper for component serialization	16
Code 3.4.1 Fields and helper functions for SaveLoadManager.....	17
Code 3.4.2 SaveLoadMangeer.create_snapshot().....	17
Code 4.3.3 SaveLoadManager.apply_snapshot()	19
Code 3.5.1 Definition of ParamNameMapping	19
Code 3.5.2 Example of autogenerated get_param_name_mappings() function for the CameraArea class (Code 3.2.1).....	20
Code 3.5.3 Example of an IPrefabSerializer generated by the metaprogram. The naming convention for generated code is to append the Serializer suffix	21
Code 3.5.4 Reference filling metaprogram. This metaprogram is not user-written, and is generated at the same time as the PrefabInfo object by IPrefabSerializerGenerator	22
Code 3.7.1.1 to_squares, part 1	27
Code 3.7.1.2 to_squares, part 2.....	28
Code 3.7.1.3 to_squares, part 3.....	28
Code 3.7.1.4 to_squares, part 4.....	28
Code 3.7.1.5 to_squares part 5	29
Code 3.7.1.6 helper function, entry_containing.....	29
Code 3.7.1.7 to_squares part 6	29
Code 3.7.1.8 to_squares, part 7.....	30
Code 3.7.1.9 to_squares part 8	31

Code 3.7.1.10 to_squares part 9.....	31
Code 3.7.1.11 to_squares part 10.....	31

References

- [1] UNITY TECHNOLOGIES. *GameObjects*. Unity Manual. Retrieved January 19, 2023, from <https://docs.unity3d.com/Manual/GameObjects.html>
- [2] UNITY TECHNOLOGIES. *Order of execution for event functions*. Unity Manual. Retrieved January 19, 2023, from <https://docs.unity3d.com/Manual/ExecutionOrder.html>
- [3] UNITY TECHNOLOGIES. *Prefabs*. Unity Manual. Retrieved January 19, 2023, from <https://docs.unity3d.com/Manual/Prefabs.html>
- [4] UNITY TECHNOLOGIES. *ScriptableObject*. Unity Manual. Retrieved January 19, 2023, from <https://docs.unity3d.com/Manual/class-ScriptableObject.html>
- [5] ACTON, M. (n.d.). A Data Oriented Approach to Using Component Systems. Unity at GDC, Retrieved January 19, 2023, from <https://www.youtube.com/watch?v=p65Yt20pw0g>
- [6] DEUTSCH, A. (n.d.). The SlotMap Data Structure. C++ Now 2017, lightning talks. Retrieved January 19, 2023, from <https://www.youtube.com/watch?v=SHaAR7XPtNU>
- [7] EPPSTEIN, D. (2010). Graph-theoretic solutions to computational geometry problems. *Graph-Theoretic Concepts in Computer Science*, 3–5. https://doi.org/10.1007/978-3-642-11409-0_1
- [8] WIKITUDE. *License Key Wikitude SDK Unity*. Wikitude Documentation. Retrieved January 19, 2023, from <https://wikitude.com/external/doc/documentation/latest/unity/triallicense.html#license-key>

Appendix

This appendix contains the code for two of the three metaprograms described in section 3. The third metaprogram is generated automatically by the second metaprogram, and so its code is dependent on whatever assets are in the project.

```

using UnityEngine;
using UnityEditor;
using System.IO;
using System.Reflection;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public static class SaveLoadGenerator
{
    [MenuItem("Generate/SaveLoadAttributes")]
    public static void generate_save_load_attributes()
    {
        List<string> class_types = new List<string>();

        StreamWriter sw = new StreamWriter(
            $"{Application.dataPath}/MyAssets/Scripts/SaveLoad/SaveLoadAttributeData.cs");
        sw.WriteLine("using System.Collections;");
        sw.WriteLine("using System.Collections.Generic;");
        sw.WriteLine("using UnityEngine;");
        foreach (Assembly assembly in System.AppDomain.CurrentDomain.GetAssemblies())
        {
            if (assembly.FullName.Contains("CSharp") && !assembly.FullName.Contains("Editor"))
            {
                foreach (System.Type classtype in assembly.GetTypes())
                {
                    if (Attribute.GetCustomAttribute(classtype, typeof(GenerateSaveLoadOverrideAttribute), false)
                        != null || Attribute.GetCustomAttribute(classtype, typeof(GenerateSaveLoadAttribute),
                        false) != null)
                    {
                        //define partial class for each ISaveLoadable
                        string class_type = classtype.ToString().Replace('+', '.');
                        class_types.Add(class_type.ToLower());

                        sw.WriteLine($"public partial class {class_type} : MonoBehaviour, ISaveLoadable");
                        sw.WriteLine("{");
                    }
                }
            }
        }
    }
}

```

```

//define save_load_id, type and data implementations,
//generating the object id and the dictionary of fields to be saved
sw.WriteLine("\tpublic SaveLoadKey save_load_id { get; set; }");
sw.WriteLine("\tpublic GameObject game_object() { return gameObject; }");
sw.WriteLine($" \tpublic SaveLoadableType type {{ get; private set; }} =
                SaveLoadableType.{class_type.ToLower()};");
sw.WriteLine("\tpublic Dictionary<int, object> data = new Dictionary<int, object>();");

if (classtype.GetMethod("activate") == null)
{
    sw.WriteLine("\tpublic void activate() {}");
}

//get all fields with [SaveLoadAttribute] for registration into SaveLoadManager system
FieldInfo[] fields = classtype.GetFields(
    BindingFlags.NonPublic | BindingFlags.Public | BindingFlags.Instance);
FieldInfo[] save_load_fields_full = Array.FindAll(fields, field =>
    Attribute.GetCustomAttribute(field, typeof(SaveLoadOverrideAttribute), true) != null
    || Attribute.GetCustomAttribute(field, typeof(SaveLoadAttribute), true) != null);
FieldInfo[] save_load_fields = Array.FindAll(save_load_fields_full, field =>
    Attribute.GetCustomAttribute(field, typeof(SaveLoadAttribute), true) != null);

//define an enum mapping each [SaveLoadAttribute] and [SaveLoadOverrideAttribute] field name
// to a unique integer, along with the default fields of type, save_load_id, position,
// rotation, and scale
//[SaveLoadOverrideAttributes] will only be added to the enum in code generation. They must
// be manually added to load_data and save_data
//assert: there are less than 2^16 - 3 fields with this attribute
sw.WriteLine($" \tprivate enum SaveLoadFields");
sw.WriteLine("\t{");
string s = (save_load_fields_full.Length == 0) ? "" : ",";
sw.WriteLine("\t\ttype = 0,");
sw.WriteLine("\t\tsave_load_id,");
sw.WriteLine("\t\ttransform_position,");
sw.WriteLine($" \t\ttransform_scale{s}");
for (int i = 0; i < save_load_fields_full.Length; ++i)

```

```

        {
            s = (i + 1 == save_load_fields_full.Length) ? "" : ",";
            string r = (save_load_fields_full[i].FieldType != typeof(ISaveLoadable) &&
                save_load_fields_full[i].FieldType.GetInterface(typeof(ISaveLoadable)) ==
                null) ? "" : "_ref";
            sw.WriteLine($"{save_load_fields_full[i].Name}{r}{s}");
        }
        sw.WriteLine("\t");
sw.WriteLine("\t");

//define a mapping from field names to their corresponding position in the enum
//this is needed because parse_data reads in user generated arguments for how to fill out
// the sprite data
//and since new fields may be added with new game updates, these arguments cant be
// positional, so named arguments are required
sw.WriteLine("\tprivate static Dictionary<string, int> save_load_field_names =
    Util.array_inverse(System.Enum.GetNames(typeof(SaveLoadFields)));");
sw.WriteLine("\t");

//define a correspondance between [SaveLoadAttribute]s and [SaveLoadOverrideAttribute]s
// fieldnames, types, and SaveLoadFields indices in the ISaveLoadable
//and the corresponding name in the PrefabSerializer (provided one is to be generated as per
// the [SaveLoadAttribute] and [SaveLoadOverrideAttribute] constructors)
//if a field name for the PrefabSerializer is passed into the constructor,
// that is what will be generated when the PrefabSerializers are built
//otherwise the ISaveLoadable field name will be used for the PrefabSerializer as well
sw.WriteLine("\tpublic List<ParamNameMapping> get_param_name_mappings()");
sw.WriteLine("\t");
sw.WriteLine($"{save_load_fields_full.Length}");
sw.WriteLine("\tList<ParamNameMapping> param_name_mappings = new
    List<ParamNameMapping>({save_load_fields_full.Length});");
for (int i = 0; i < save_load_fields_full.Length; ++i)
{
    string field_name = save_load_fields_full[i].Name;
    string field_type = Regex.Replace(
        save_load_fields_full[i].FieldType.ToString().Replace('+', '.'),
        @"1\[([a-zA-Z0-9]*)\]", "<$1>");
    string prefab_field_name = "";

```



```

SaveLoadAttribute sl1 = (SaveLoadAttribute)Attribute.GetCustomAttribute(
    save_load_fields_full[i], typeof(SaveLoadAttribute));
SaveLoadOverrideAttribute sl2 = (SaveLoadOverrideAttribute)Attribute.GetCustomAttribute(
    save_load_fields_full[i], typeof(SaveLoadOverrideAttribute));
if (sl1 != null && sl1.generate_asset_parameter)
    prefab_field_name = (sl1.asset_parameter_name == "") ?
        field_name : sl1.asset_parameter_name;
if (sl2 != null && sl2.generate_asset_parameter)
    prefab_field_name = (sl2.asset_parameter_name == "") ?
        field_name : sl2.asset_parameter_name;
if (prefab_field_name != "")
    sw.WriteLine($"{t\tparam_name_mappings.Add(new ParamNameMapping(\"{field_name}\",
        \"{field_type}\", \"{prefab_field_name}\", {i + 4}));");
}
sw.WriteLine("\t\treturn param_name_mappings;");
sw.WriteLine("\t}");
sw.WriteLine("\t");

//for a [GenerateSaveLoadAttribute], define load_data automatically
//for a [GenerateSaveLoadOverrideAttribute],
//define a load_data_function which does all the automated work
//required by ISaveLoadable's load_data
bool helper_function = Attribute.GetCustomAttribute(
    classtype, typeof(GenerateSaveLoadOverrideAttribute), false) != null;
sw.WriteLine($"{t\t{(helper_function ? "private" : "public")} void load_data{(helper_function
    ? "_" : "")}(Dictionary<int, object> data)");
sw.WriteLine("\t{");
sw.WriteLine("\t\tif (data.ContainsKey((int) SaveLoadFields.type)) type =
    (SaveLoadableType) data[(int) SaveLoadFields.type];");
sw.WriteLine("\t\tif (data.ContainsKey((int) SaveLoadFields.save_load_id)) save_load_id =
    (SaveLoadKey) data[(int) SaveLoadFields.save_load_id];");
sw.WriteLine("\t\tif (data.ContainsKey((int) SaveLoadFields.transform_position))
    transform.position = (Vector3) data[(int) SaveLoadFields.transform_position];");
sw.WriteLine("\t\tif (data.ContainsKey((int) SaveLoadFields.transform_scale))
    transform.localScale = (Vector3) data[(int) SaveLoadFields.transform_scale];");
foreach (FieldInfo field in save_load_fields)
{

```

```

string field_name = field.Name;
string field_type = Regex.Replace(field.FieldType.ToString().Replace('+', '.'),
    @"\1\\\[([a-zA-Z0-9]*)\\]", "<$1>");
if (field.FieldType.GetInterface(typeof(IEnumerable)) != null)
{
    sw.WriteLine($"\\t\\tif (data.ContainsKey((int)SaveLoadFields.{field_name}))
        {field_name} = new {field_type}({{field_type}}data[(int)SaveLoadFields.
            {field_name}]);");
}
else
{
    if (field.FieldType == typeof(ISaveLoadable) ||
        field.FieldType.GetInterface(typeof(ISaveLoadable)) != null)
    {
        sw.WriteLine($"\\t\\tif (data.ContainsKey((int)SaveLoadFields.{field_name}_ref
            ))");
        sw.WriteLine("\\t\\t{");
        sw.WriteLine($"\\t\\t\\tSaveLoadKey {field_name}_ref = (SaveLoadKey)data[(int)
            SaveLoadFields.{field_name}_ref];");
        sw.WriteLine($"\\t\\t\\t{field_name} = ({{field_name}_ref.is_null()) ? null :
            ({{field_type}})SaveLoadManager.get_component({field_name}_ref);");
        sw.WriteLine("\\t\\t}");
    }
    else
    {
        sw.WriteLine($"\\t\\tif (data.ContainsKey((int)SaveLoadFields.{field_name}))
            {field_name} = ({{field_type}})data[(int)SaveLoadFields.
                {field_name}]);");
    }
}
}
sw.WriteLine("\\t");
sw.WriteLine("\\t");

//for a [GenerateSaveLoadAttribute], define save_data automatically
//for a [GenerateSaveLoadOverrideAttribute],
//define a save_data_function which does all the work required by ISaveLoadable's save_data

```

```

sw.WriteLine($"\\t{(helper_function ? "private" : "public")} Dictionary<int, object>
    save_data{(helper_function ? "_" : "")}());");
sw.WriteLine("\\t{");
sw.WriteLine("\\t\\tdata.Clear();");
sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.type, type);");
sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.save_load_id, save_load_id);");
sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.transform_position, transform.position);");
sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.transform_scale, transform.localScale);");
foreach (FieldInfo field in save_load_fields)
{
    string field_name = field.Name;
    string field_type = Regex.Replace(field.FieldType.ToString().Replace('+', '.'),
        "\\1\\[([a-zA-Z0-9]*)\\]", "<$1>");
    if (field.FieldType.GetInterface(nameof(IEnumerable)) != null)
    {
        sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.{field_name},
            new {field_type}({field_name}));");
    }
    else
    {
        if (field.FieldType == typeof(ISaveLoadable) ||
            field.FieldType.GetInterface(nameof(ISaveLoadable)) != null)
        {
            sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.{field_name}_ref, ({field_name}
                == null) ? SaveLoadKey.mk_null() : {field_name}.save_load_id);");
        }
        else
        {
            sw.WriteLine($"\\t\\tdata.Add((int)SaveLoadFields.{field_name}, {field_name});");
        }
    }
}
sw.WriteLine("\\t\\treturn data;");
sw.WriteLine("\\t}");
sw.WriteLine("\\t");
sw.WriteLine("");
}

```

```

        }
    }
}

sw.WriteLine("public enum SaveLoadableType");
sw.WriteLine("{}");
for (int i = 0; i + 1 < class_types.Count; ++i)
{
    sw.WriteLine($"{class_types[i]},");
}
if (class_types.Count > 0)
{
    sw.WriteLine($"{class_types[class_types.Count - 1]}");
}
sw.WriteLine("{}");

sw.Close();
AssetDatabase.Refresh();
}
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
using System.IO;

public static class PrefabGenerator
{
    const string prefab_folder = "Assets/MyAssets/Prefabs";
    const string prefab_info_folder = "Assets/MyAssets/Resources";

    [MenuItem("Generate/PrefabSerializer")]
    public static void generate_prefab_serializers()
    {
        //Find all prefabs in folder :
        List<GameObject> prefabs = new List<GameObject>();
    }
}

```

```

string[] guid = AssetDatabase.FindAssets("t:Prefab", new string[1] { prefab_folder });
foreach (string id in guid)
{
    string asset_path = AssetDatabase.GUIDToAssetPath(id);
    prefabs.Add((GameObject)AssetDatabase.LoadAssetAtPath(asset_path, typeof(GameObject)));
}

//Write PrefabSerializer classes for each prefab
StreamWriter sw = new StreamWriter(
    $"{Application.dataPath}/MyAssets/Scripts/SaveLoad/PrefabSerializerData.cs");
sw.WriteLine("using System.Collections;");
sw.WriteLine("using System.Collections.Generic;");
sw.WriteLine("using UnityEngine;\n");
sw.WriteLine("public interface IPrefabSerializer { PrefabType get_prefab_type(); Dictionary<int, object>[] get_component_snapshots(); }\n");

string[] value_names = new string[prefabs.Count];

for (int i = 0; i < prefabs.Count; ++i)
{
    string class_name = remove_whitespace_and_parens(prefabs[i].name);
    value_names[i] = to_value_name(class_name);

    ISaveLoadable[] prefab_components = prefabs[i].GetComponents<ISaveLoadable>();
    List<ParamNameMapping>[] prefab_fields = new List<ParamNameMapping>[prefab_components.Length];
    for (int j = 0; j < prefab_components.Length; ++j)
    {
        //generate param_name_mappings at the same time as ISaveLoadables
        //so we only have to search the assembly once
        prefab_fields[j] = prefab_components[j].get_param_name_mappings();
    }

    sw.WriteLine($"public class {class_name}Serializer : IPrefabSerializer");
    sw.WriteLine("{");
    sw.WriteLine($"\tpublic PrefabType get_prefab_type() {{ return PrefabType.{value_names[i]}; }}");

    //every [SaveLoadAttribute("paramname")] in a component of this prefab

```

```

//gets a corresponding field named paramname
//this is so all of the fields can be loaded in from a file specified at the Prefab level,
//serialized, and then loaded into components
for (int j = 0; j < prefab_fields.Length; ++j)
{
    for (int k = 0; k < prefab_fields[j].Count; ++k)
    {
        sw.WriteLine($"\\t{prefab_fields[j][k].component_field_type}
                    {prefab_fields[j][k].prefab_field_name};");
    }
    sw.WriteLine("");
}

//get_component_snapshots translates from prefab fields to component fields
//when each SaveLoadAttribute is read by the SaveLoadGenerator, a ParamNameMapping is constructed,
//which remembers the snapshot index, the field name for the component,
//the field name for the prefab, and the type
//that info gets read here, building snapshot Dictionaries for each ISaveLoadable component on the
//prefab, so that the ISaveLoadables can read the field values in with load_data
sw.WriteLine("\\tpublic Dictionary<int,object>[] get_component_snapshots()");
sw.WriteLine("\\t{");
sw.WriteLine($"\\t\\tDictionary<int,object>[] component_snapshots = new Dictionary<int,object>
                [{{prefab_fields.Length}}];");
for (int j = 0; j < prefab_fields.Length; ++j)
{
    sw.WriteLine($"\\t\\tcomponent_snapshots[{{j}}] = new Dictionary<int,object>();");
    for (int k = 0; k < prefab_fields[j].Count; ++k)
    {
        sw.WriteLine($"\\t\\tif ({{prefab_fields[j][k].prefab_field_name}} != null)
                    component_snapshots[{{j}}][{{prefab_fields[j][k].component_field_index}}] =
                    {{prefab_fields[j][k].prefab_field_name}};");
    }
}
sw.WriteLine("\\t\\treturn component_snapshots;");
sw.WriteLine("\\t}");

sw.WriteLine("}\\n");

```

```

}

//enumerate all the prefabs in the PrefabType enum
sw.WriteLine("public enum PrefabType");
sw.WriteLine("{");
for (int i = 0; i < value_names.Length; ++i)
{
    string comma = (i + 1 == value_names.Length) ? "" : ",";
    sw.WriteLine($"    {value_names[i]}{comma}");
}
sw.WriteLine("}");

//build a ScriptableObject which can translate from the PrefabType enum to a reference
//to the corresponding Prefab
sw.WriteLine("public class PrefabInfo : ScriptableObject");
sw.WriteLine("{");
for (int i = 0; i < value_names.Length; ++i)
{
    sw.WriteLine($"    public GameObject {value_names[i]}");
}
sw.WriteLine("");
sw.WriteLine("public static GameObject get_prefab(PrefabType prefab_type) { return
    ResourceManager.sharedInstance.prefabInfo.get_prefab_(prefab_type); }\n");
sw.WriteLine("    public GameObject get_prefab_(PrefabType prefab_type)");
sw.WriteLine("    {");
sw.WriteLine("        switch(prefab_type)");
sw.WriteLine("        {");
for (int i = 0; i < value_names.Length; ++i)
{
    sw.WriteLine($"            case PrefabType.{value_names[i]}: return {value_names[i]}");
}
sw.WriteLine("        default: return null;");
sw.WriteLine("    }");
sw.WriteLine("}");
sw.WriteLine("}");

sw.Close();

```

```

//create another metaprogram which creates and fills out the fields of the PrefabInfo ScriptableObject
//once code has compiled (and PrefabType has been updated to include any new prefabs)
StreamWriter sw2 = new StreamWriter(
    $"{Application.dataPath}/MyAssets/Scripts/SaveLoad/Editor/PrefabInfoFieldFiller.cs");
sw2.WriteLine("using System.Collections;");
sw2.WriteLine("using System.Collections.Generic;");
sw2.WriteLine("using UnityEditor;");
sw2.WriteLine("using UnityEngine;\n");

sw2.WriteLine("public static class PrefabInfoFieldFiller");
sw2.WriteLine("{");
sw2.WriteLine("\t[MenuItem(\"Generate/PrefabSerializerLinks\")]");
sw2.WriteLine("\tpublic static void generate_prefab_serializer_links()");
sw2.WriteLine("\t{");
sw2.WriteLine("\t\tList<GameObject> prefabs = new List<GameObject>();");
sw2.WriteLine($"\t\tstring[] guid = AssetDatabase.FindAssets(\"t: Prefab\", new string[1] {{");
sw2.WriteLine("\t\t\t\"{prefab_folder}\" });");
sw2.WriteLine("\t\tforeach (string id in guid)");
sw2.WriteLine("\t\t{");
sw2.WriteLine("\t\t\tstring asset_path = AssetDatabase.GUIDToAssetPath(id);");
sw2.WriteLine("\t\t\tGameObject prefab = (GameObject)AssetDatabase.LoadAssetAtPath(asset_path,");
sw2.WriteLine("\t\t\t\ttypeof(GameObject));");

sw2.WriteLine("\t\t\tprefabs.Add(prefab);");
sw2.WriteLine("\t\t}");
sw2.WriteLine("\tPrefabInfo prefab_info_instance = ScriptableObject.CreateInstance<PrefabInfo>());");
for (int i = 0; i < value_names.Length; ++i)
{
    //assumes prefabs have not been added or moved between
    //this script running and the generated script running! i.e. prefabs == "prefabs".
    //Run the generated script immediately after this one.
    sw2.WriteLine($"\t\tprefab_info_instance.{value_names[i]} = prefabs[{i}];");
}
sw2.WriteLine($"\tAssetDatabase.CreateAsset(prefab_info_instance, \"{prefab_info_folder");
sw2.WriteLine("\t\t}/PrefabInfoAsset.asset\");");
sw2.WriteLine("\tAssetDatabase.Refresh();");

```



```

        sw2.WriteLine("\t");
        sw2.WriteLine("");
        sw2.Close();
        AssetDatabase.Refresh();
    }

public static string to_value_name(string class_name)
{
    StringWriter s = new StringWriter();
    bool first_letter = true;
    for (int i = 0; i < class_name.Length; ++i)
    {
        if (char.IsUpper(class_name[i]))
        {
            if (!first_letter) s.Write('_');
            s.Write(char.ToLower(class_name[i]));
        }
        else
        {
            s.Write((class_name[i] == ' ') ? '_' : class_name[i]);
        }
        if (char.IsLetter(class_name[i]))
        {
            first_letter = false;
        }
    }
    return s.ToString();
}

public static string remove_whitespace_and_parens(string s)
{
    StringWriter sw = new StringWriter();
    for (int i = 0; i < s.Length; ++i)
    {
        if (!char.IsWhiteSpace(s[i]) && s[i] != '(' && s[i] != ')') sw.Write(s[i]);
    }
    return sw.ToString();
}

```

} }