Persistence in games

A solution for easily saving and loading the game state at run-time in the Unity game engine.

Sander A. Verbeek

Little Chicken Game Company, Amsterdam

March 2012

Approved by supervisor:

Student:

Name:	Sander A. Verbeek
Address:	Molenheide 26 A, 6027 PZ Soerendonk
Tel:	04 95 59 35 77 06 20 18 49 71
E-mail:	sanderman@gmail.com
School:	Fontys Hogeschool voor ICT, Eindhoven
	Bachelor HBO-ICT: Software Engineering + Game Design.
Student Nr:	2153619

Internship at:

Employer:	Little Chicken Game Company
Address:	Weteringschans 86, 1017 XS Amsterdam
Tel:	02 06 20 29 70
E-mail:	info@littlechicken.nl

Foreword

This thesis describes my internship at Little Chicken Game Company, a small game studio, where among other tasks I did research and development on the topic of persistence in games. The assignment was to develop a system to aid in implementing functionality like the saving and loading of game progress in Unity games, making this task easier for other developers in the future.

I would thank both of my supervisors at work and at school for their support and advice during this period.

Contents

Summary

For my internship and graduation I chose to work at Little Chicken, where I researched and designed a system that makes it easier for developers to implement persistence functionality in games in the Unity game engine.

Persistence is the ability of an application to remember its current state and later resume execution to the same state. This is very useful for games where player progress needs to be remembered so that the player can resume playing where he left off, with the game world exactly as he left it, instead of having to start over from scratch.

Functionality like this is very common in games but was always a very annoying problem, requiring custom solutions for every game that may or may not work reliably, and which would often result in a lot of maintenance overhead and boring boilerplate code, that we really are not interested in.

In this assignment I had the goal to standardize this process somewhat, and to automate it where possible. This makes this particular problem less of a chore when building a game that requires functionality like saving and loading the game's state.

I chose to do some research first on how other people approach this problem, followed by an iterative process where I developed and analyzed several prototypes followed by the final solution. Among other reasons, I chose this method because this problem is quite abstract and it would be hard to predict where problems would occur. There was also the need for a working prototype at an early stage for the Kenteq project.

The first prototype was well received and integrated into the Kenteq project. Various improvements happened since then. The final solution works well and makes saving and loading the state of a Unity game a lot easier for the developer.

Samenvatting

Gedurende mijn afstudeerstage heb ik gewerkt bij Little Chicken, waar ik heb het probleem van persistentie in games heb onderzocht en een systeem heb ontwikkeld waarmee dit probleem makkelijker op te lossen is voor ontwikkelaars van games in de Unity engine.

Persistentie betekent de eigenschap van een programma om zijn eigen toestand op te slaan en later opnieuw in kan lezen. Dit is zeer nuttig voor games waar de voortgang van de speler onthouden moet worden, zodat de speler later weer verder kan gaan waar hij gebleven was, met de spelwereld exact zoals hij hem heeft achtergelaten, in plaats van opnieuw te beginnen.

Functionaliteit als dit wordt veel gebruikt in games maar was altijd nogal een vervelend probleem. Vaak had elk spel een eigen oplossing op maat nodig, wat leidde tot rommelige code die veel onderhoud vereiste bij veranderingen in het spel en waar eigenlijk niemand in geïnteresseerd was.

In deze opdracht had ik het doel om dit proces te standaardiseren en te automatiseren, waardoor het makkelijker wordt voor de ontwikkelaar om dit soort functionaliteit in een game te maken.

Ik heb ervoor gekozen om eerst onderzoek te doen naar diverse methoden waarop andere mensen dit probleem aanpakken. Vervolgens heb ik in een iteratief proces verschillende prototypes, en de uiteindelijke oplossing ontwikkeld. Ik heb hier onder andere voor gekozen omdat het probleem behoorlijk abstract is en het lastig was om te voorspellen met welke problemen ik te maken zou krijgen. Er was ook al relatief vroeg een werkend prototype nodig voor het Kenteq project.

Het eerste prototype werd goed ontvangen en geïntegreerd in het Kenteq project. Vervolgens vonden er verschillende verbeteringen plaats. De uiteindelijke oplossing werkt goed en maakt het opslaan van de toestand van een Unity game een stuk makkelijker voor de ontwikkelaar.

Glossary

Game Engine	A system designed to assist in the creation and development of video games. Game engines include functionality for rendering visual elements to the screen, playing video and audio, physics simulation, and to offer a foundation upon which the rest of a game can be built. In this project, the Unity game engine is used.
Saved Game	Term used to describe functionality present in many common video games and refers to saving the player's progress to a 'save' in a file or save slot before quitting the game. This is simply referred to as 'saving' the game. The player can later return to the game by choosing the desired save. This is often called 'loading a save.'
Scene	A collection of game objects in game memory, organized in a hierarchical structure. Different scenes can be defined in the editor and loaded at run-time. The contents of the scene can change during game play. Loading a different scene is commonly used to switch environments or game modes.
Game Object	A single object in the scene with a name, position, orientation and scale. Game objects can have numerous components, to provide specific properties or behavior.
Component	A piece of functionality that can be attached to a game object. Other components that can be attached to game objects include: MeshRenderer for visible geometry, AudioSource for emitting sound, and custom scripts.
Game Element	An object in a scene which interacts with the player or other game elements. Examples of game elements include: terrain, obstacles, vehicles, player characters, enemies. In Unity, game elements can be composed of one or more game objects.
Game Entity	A game element which changes state, and wants this state to be persisted either partially or completely. Basically this is any game element that needs to be the same after saving the game and loading it at a later point in time. Examples include: player with amount of health, inventory or in-game currency. Changes like destroying parts of or adding to the environment.
Persistence	The concept of state outliving the process that created it. In our situation, this means we want to store the entire game state to a safe location before the game exits and load it back in later. While saved-game refers to a concept in games, persistence refers to the whole problem domain and how this saving and loading of the game state is achieved, also in other applications.
Serialization	Serialization is the process of converting a data-structure or object in computer memory to a sequential format for storage. The reverse is called deserialization. It is an important problem in persistence. It is not uncommon for data-structures to be very complex, which often makes using an automatic serialization library preferable to doing it manually.
XML	Extensible Markup Language (XML) is a common human-readable format and rules for documents and other data storage used in a variety of applications.
JSON	JavaScript Object Notation (JSON) is a human-readable data-interchange format commonly used on the web.
Binary	Catch-all term for custom binary storage formats used by various applications. Generally refers to writing primitives and simple data-structures directly to a byte stream manually.
Reflection	The process by which a computer program can observe its own structure. This can be combined with code annotations to change behavior at run-time.

Introduction

Most game developers have probably encountered the following situation once upon a time:

You have this idea for a great game, and start work on it. Pretty soon, you need to be able to save the game, so a player can keep his progress and resume playing where he left off at a later moment. You start work on this but quickly become embroiled in boilerplate code for having to write every game object to disk manually, or you encounter problems with serialization, and how to manage it all? Aren't there ways to automate this?

This problem domain is called persistence and at the moment there do not seem to be any standard systems or best practices to assist in solving this problem among game developers. Since platforms, game engines, and the games themselves can be very different, it seems every game needs its own saving and loading routines. The community knowledge on this problem domain seems very fragmented. Software engineers in general have largely solved this with relational databases and fancy abstraction layers, but these are often not a good solution for use in games for various reasons.

During my internship at Little Chicken, a game studio in Amsterdam, I investigated this topic and developed a system for games built in the Unity engine and which assists in providing persistence functionality for games that use it. At the time, Little Chicken was developing a game codenamed Kenteq which would need functionality to save the state of something resembling a roller-coaster track and car built by the player and where a system like this would be very useful.

Before starting development, research was done on the current methods various people use to solve this problem both in the Unity community and outside. Several sub problems and possible solutions were identified. The best solutions were chosen to be used in the system. Over the course of several iterations, prototypes were developed and analyzed. The final system makes adding persistence functionality to a Unity game much easier than before.

Chapter 1 describes the company. Chapter 2 describes the assignment in detail. Chapter 3 tells of the structure of the project and the plan of approach. Chapter 4 relates which research was done before development. Chapter 5 through 7 relate the various development iterations to arrive at the final solution. Chapter 8 concludes on the result of the whole process.

1. The company

Little Chicken Game Company is an independent game studio located in the city of Amsterdam. The core activity is the development of serious games, branded games and entertainment games. Primarily working on small projects for various clients and with publishers in different industries, they have developed over a hundred game projects in the last ten years. Lately they are focusing more on mobile games.

Their most notable project recently was the online game 'Raveleijn' developed for the Efteling theme park, set in a medieval world. Players talk to villagers and vanquish enemies, among other things. Raveleijn is one of the largest free online games released in the Netherlands that year, including over forty quests and many play hours. Since launch in 2011, the game has drawn over three hundred thousand visitors.

They are a small studio with a sociable, informal atmosphere. At the end of every Friday there is a happy hour, playing games and having fun, unless there is a deadline looming.

2. The assignment

This chapter details the exact form and requirements of the assignment, and the background which led to it. These were documented in the Project Initiation Document (PID) at the start of the project.

2.1 Background

2.1.1 Context

As a student at the Fontys Hogeschool for ICT in Eindhoven, for my graduation in 2012, I found an internship position at the Little Chicken Game Company in Amsterdam. Shortly before starting this internship, we had a thorough discussion on the matter of the internship assignment, which, in addition to regular software and game development work, was required to have a significant research component by school and government standards.

During this discussion we considered various subjects of interest to me, including AI, Computer Graphics, Procedural Content, Networking, and Persistence. The latter turned out to be a current issue at Little Chicken.

2.1.2 Concrete reason for the project

At the start of this project, Little Chicken was developing an educational game with the codename Kenteq. This game would contain a component where the player can create their own 'levels' by changing, adding and removing game elements to and from the scene. This game required a feature and mechanism to save the current progress of the player and state of the game, and to load this back in at a later playing session. The chosen development environment, the Unity game engine, does not provide any standard solution for saving or loading the game's state at run-time, which means a custom solution needs to be created.

There was also the wish for a generic solution that can be reused and integrated in games yet to be developed. In the future, this should result in lower efforts needed to implement functionality like this in games. This improves efficiency and allows more resources to be saved or directed to other areas of game development.

2.1.3 Current Situation

For the current project, Little Chicken chose to use the Unity game engine. The people in this company are still gaining familiarity with this software. This means the collections of in-house tools and plug-ins to use with this engine is still quite small, and does not include any solution for handling game saves.

Persistence is largely a solved problem in enterprise applications, where persistence layers are being used to abstract the functionality into reusable systems and store data in databases. Applications are easily separated into domain model and application logic, which makes extracting the data easy.

In game development this is not the case. Various factors contribute to the fact that persisting the game state is often quite a difficult task. Depending on the type of game, the game state might need to be saved to the local filesystem, on remote a web server or somewhere else entirely. In addition to that, games elements include a lot of static data, like meshes and textures, that never change, and as such needs to be separated from the data you want to persist, which is sometimes quite difficult due to the complexity of the scene and the design choices and limitations of the game engine and the way the game is built. Merely saving and loading the persisted data is not enough, because the scene also needs to be reconstructed to the correct state using that data. All these factors mean that a lot of developers will roll custom solutions to suit their specific game and only that game.

2.2 Project Definition

This section describes the goals this project will attempt to achieve and the products that should be delivered. Also included are the exclusions and risks determined for this project. Defining this information aids in making sure the project stays on track and delivers satisfactory results in the allotted time frame. Due to the nature of the assignment, the project goals are necessarily quite abstract and hard to quantify, which needs to be taken into account during evaluation.

2.2.1 Goals

This project will attempt to achieve the following goals, with the intent to simplify the implementation of persistence in games:

1. Research currently used methods and solutions used to handle persistence in games.
2. Realize system to assist implementation of persistence in games with the following properties:
   - Standard: Abstracted method of handling persistence in games
   - Generic: Usable in a wide variety of games
   - Simple: Adding extra persistent game elements is intuitive and involves minimal effort.
   - Flexible: Easily adapted to situations requiring different forms of serialization or data storage.

The final result should be a system that achieves these goals as much as possible.

We hope to learn more about this subject by researching and developing a system to assist in these tasks. We also want to avoid reinventing the wheel for every new game requiring saving and loading of player progress and other parts of the game's state.

2.2.2 Scope

- This project will have significant influence on the development of the Kenteq game which is currently being produced by Little Chicken. Results of this project will be integrated into the Kenteq project.
- Later projects at Little Chicken with the same feature demands might reuse the results of this project.
- Both the company mentor and school mentor for this internship are involved in this project.
- While the Kenteq project itself will end at the end of March, this internship project and assignment will continue until the end of the internship in June.
- Due to the complexity of the subject I will limit myself to a system for use in the Unity game engine. However, the lessons from this endeavor should be applicable in other areas of game development as well.

2.2.3 Products

During the project, the following products are delivered.

- Project Initiation document (PID)
- Prototypes of differing solutions
- Analysis on methods handling persistence in games
- Persistence layer system for games created in Unity and matching documentation
- Thesis detailing the process of this assignment
- Presentation on the assignment
- Blog with frequent updates on the progress of the assignment and internship

2.2.4 Exclusions

While the following areas are closely related to the subject of this assignment, they shall not or rarely be covered in this assignment or thesis:

- Persistence and serialization in relation to the networking problem domain.

- General gameplay and miscellaneous functionality in the Kenteq game project unrelated to this assignment. While I will be working on these parts in addition to this project, I will not cover those in this assignment and thesis.

2.2.5 Risks

The following risks play a part in this project:

- Next to this assignment, I will also work on different projects at Little Chicken. There is a chance that, because of deadlines and other circumstances, I will not spend enough time on my assignment, which will cause problems for graduation. Care needs to be taken to avoid this.

- One of the first prototypes will be integrated into the Kenteq project. If this prototype is not sufficiently successful in achieving its goals, then this will delay the Kenteq project until it has been improved, or an alternative solution developed.

3. Method

This chapter describes the step-by-step plan of approach taken to find the internship position and completing the assignment, and also includes the general planning used for the project.

3.1 Plan of approach

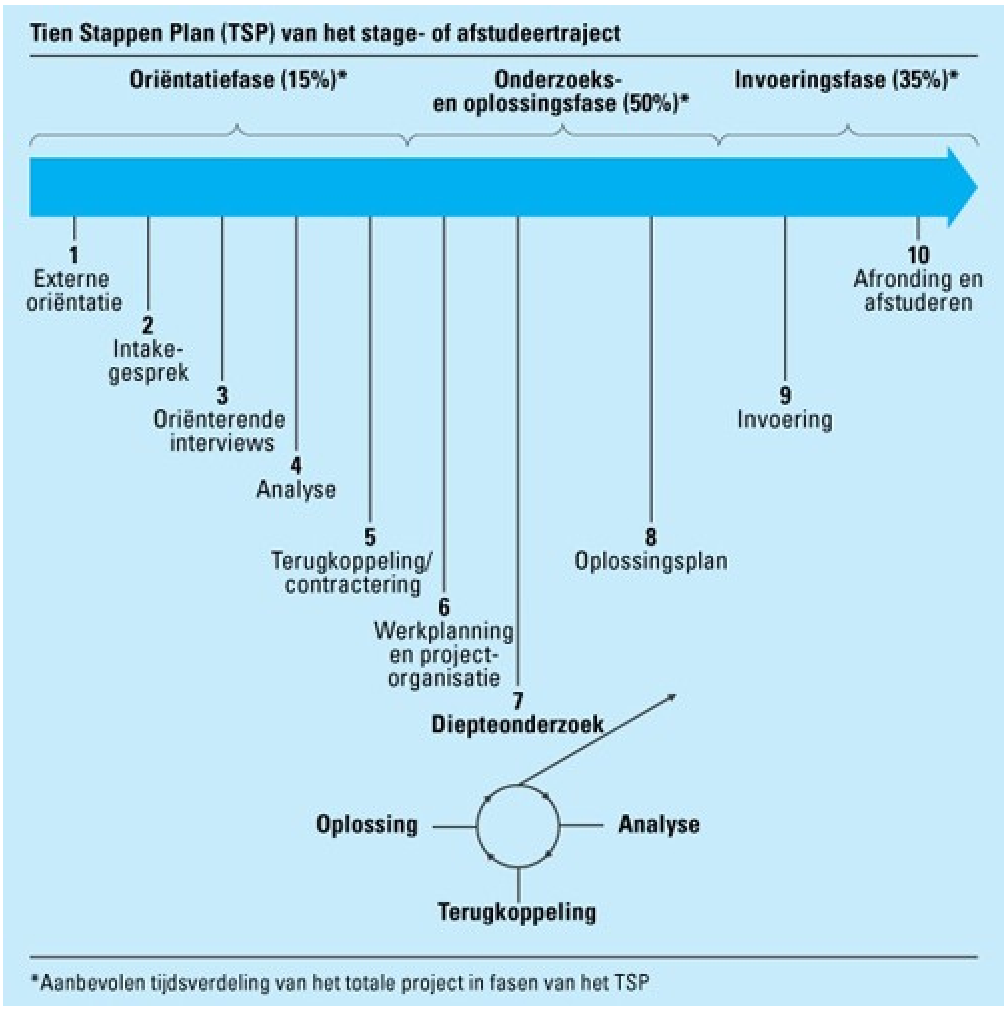

Tien Stappen Plan (TSP) van het stage- of afstudeertraject
Oriëntatiefase (15%)*
Onderzoeks- en oplossingsfase (50%)*
Invoeringsfase (35%)*
1 Externe oriëntatie
2 Intake-gesprek
3 Oriënterende interviews
4 Analyse
5 Terugkoppeling/ contractering
6 Werkplanning en project-organisatie
7 Diepteonderzoek
8 Oplossingsplan
9 Invoering
10 Afronding en afstuderen
Oplossing
Analyse
Terugkoppeling
*Aanbevolen tijdsverdeling van het totale project in fasen van het TSP

Figure 1: Ten-Stap-Plan used to organise this internship and project.

The school required the presence of a significant research component, and advised the use of the Ten-Steps-Plan (TSP) for the final graduation internship.

By the time this plan was presented to us, I had already been accepted for an internship position at Little Chicken, so the first few steps had already been completed.

The following sections describe the entire process in detail.

3.1.1 External Orientation

The first step involved gathering general information about the company of interest. This includes their business activities, motivations and interests, and making an initial estimate of whether the company was a good fit. When all seemed well, it was time to prepare for an interview.

In other careers this preparation would mostly involve updating my resume. Since my desired position was in the game industry, this meant that a portfolio of previous work to show skill was also very important.

3.1.2 Intake / Interview

The interview allowed for a more personal meeting between aspiring intern and the people of the company. This was basically the same as an interview for a regular job. This allowed both the intern and the company to judge more accurately whether the company was a good fit for the intern, and whether the intern will have the right personality and skills to complete the assignment and other tasks given.

3.1.3 Internal Orientation

The internal orientation was needed to determine what the assignment should be. We had a thorough discussion on which projects were currently being worked on, and what functionality they would require. We chose the assignment with this in mind. The rest of this phase was spent getting familiar with the company, their current projects and gathering general information relevant to the assignment and current projects.

3.1.4 Analysis

This step was used to familiarize with the problem domain, and to further define the details of the assignment. During this phase, the PID was written to describe my interpretation of the assignment and the exact goals and limits of the project, along with other details. A plan of approach was included in this PID.

3.1.5 Feedback

The PID was sent to my company coach and school coach for feedback. After integrating this feedback and getting confirmation that I was on the right track, I started work on the project in earnest.

3.1.6 Work planning and project organization

This step normally involves two parts. One of those is organizing the project. One part of this is gathering all the resources and information needed for the project. In this case that merely meant configuring a development environment on my machine and retrieving the main project from a subversion repository. Another important part is to start work on documents like the thesis.

Another is to divide the general planning into a more detailed planning of smaller tasks. This part was skipped because of a few factors. Firstly, the fact that I also worked on other projects means that making a detailed planning for this one would be quite difficult and would have to be adjusted quite frequently. There would be moments where deadlines meant most of my time was taken by the main project or side projects, offset by periods of relative calm where most of my time could be spent on the assignment. Secondly, the assignment was quite abstract in a a number of ways, which made it difficult to estimate the individual tasks and the time they would take. Thirdly, the detailed style of planning does not suit me personally.

Instead, more detailed preliminary research on the subject was done, to prepare for the research and development step. This meant to examine the current state of affairs on this subject in the game development community Unity, specifically among Unity developers, analyzing the various sub problems present and the various methods available to deal with those. Summarizing these helped indicate the most promising leads during the next step, and was the first stage of my research. This stage was mainly secondary research.

3.1.7 Research and Development

For this stage an iterative process was chosen. The analysis from the previous step was used to select the most promising solutions to the various sub problems, for the first prototype. After developing this prototype and integrating it into the game. An analysis was made to evaluate it's merits, shortcomings and other comments or insights. These findings were used in the second prototype.

This could be summarized as a process combining heuristic research and iterative development.

The reason this method was chosen was the abstract nature of the assignment. Though most sub problems and issues were documented during the preliminary research, finding all of them this way would be impossible. Some problems are only encountered in practice and thus should be dealt with in a flexible manner. This makes iteration a better fit for this project, both for development and research. Two prototypes were made and analyzed in an iterative process.

3.1.8 Final Solution

The final solution took one more iteration to build on the previous prototypes. After another evaluation, some of the remaining weaknesses were fixed where practical. The system was also made more robust and easier to use.

3.1.9 Putting it into use

While the system was now finally completed, the first prototype was already put into use on the Kenteq project. In the future, other newly started projects might use it for their functionality. All that remained to be done was to write good documentation, to ensure that people will still know how the system works and be able to use it without problems in my absence.

3.2 Planning

At the start of the project, the following planning was made. In addition to my formal assignment for my thesis and school evaluation, I also did work on different projects. To prepare for any contingencies, a margin of two weeks was planned before the estimated deadline at 22 of June.

The Kenteq project related to this assignment was due for the end of march, which was quite quickly compared to the start and end of my internship, therefore we decided to integrate one of the first prototypes into the game, if it were deemed adequate. This also served as a proof of concept, so this was a convenient solution to the problem.

Activity	Date of delivery
PID	24-02-2012
Preliminary research	09-03-2012
Prototypes	13-04-2012
Final Analysis	20-04-2012
Realization	18-05-2012
Thesis and finalization	08-06-2012

I was later informed that the deadline for the thesis was actually at June 7, quite some time before the end of my internship, so a bit earlier than expected. Luckily I planned in a large margin in the first place.

4. Preliminary Research

The first concrete step in this project was a phase of preliminary research. Using secondary research on websites and forums, the current situation in the field was assessed. The various sub problems in the field and the common methods of solving these problems were found and analyzed.

4.1. Introduction

This assignment covers the subject called persistence, particularly, persistence in the context of games and game engines like Unity. This means we have to deal with a few issues which make persistence quite difficult for us.

Game engines these days are often based on an object-component model and heavily data driven. This paradigm appears very suited to game and other simulation development and is a trend in this industry. This also means less emphasis on the traditional domain model used in other areas of software engineering.

The use of the object-component model means game elements are often composed of many different objects and components. A game object like a person might consist of multiple child objects for the torso, head and limbs. A car might have a body, wheels and other parts. Each of these game objects will have components determining that objects behavior. In Unity, common components are the MeshFilter and Renderer (represents an objects geometry for display), Collider components (determines how objects collide with each other), and components inheriting from MonoBehaviour, which are scripts developers can write themselves for their own functionality.

There are no limits on the number of child objects and components a game object might have. It is not uncommon for game objects to contain huge hierarchies of parts. We will call game elements that should be persisted game entities. All of these game entities might contain data you want to persist when saving the game state and which you want to restore when the player starts up the game the next day and wants to continue playing where he left off. This design makes game development very straightforward, but is problematic for persistence, because the data can not easily be extracted and restored unless this was specially considered and designed for during the start of development. The fact that a lot of data is transient data or data built into the game, which does not need to be saved, complicates matters further, in addition to the fact that different games need different storage methods.

Software engineers used to working on enterprise systems and new to game development will probably recognize that these situations are quite different from the usual persistence scenario often discussed, business information systems where all data is conveniently contained in a collection of domain objects, to be easily persisted into a relational database.

All these factors contribute to the fact that persistence is still quite a difficult problem for a lot of independent game developers using Unity and other game engines.

4.2 Current Situation

The current situation in the field was assessed prior to starting my own research and development efforts. How do other developers in the game design field handle persistence in their games? In addition to the Unity community, other game development environments and communities were also considered. A lot of concepts are applicable in different environments.

The Unity community seems quite fragmented on this issue. On the Unity forums there are frequently questions by developers about saving game state and other forms of persistence. Most of the answers to these questions are very primitive and only focus on small aspects of the bigger problem. Most of the suggested solutions are extremely primitive and once you need more than that, you are effectively on your own. There is almost no deeper discussion on the best design practices for handling persistence problems in game development, other that the basic techniques of serialization and storage. Most third party commercial solutions are quite primitive and often unsuitable for the requirements.

Apart from the extremely limited PlayerPrefs functionality, Unity itself does not provide any standard mechanism for game state persistence at run-time. This means that usually a custom solution is needed. The general consensus of the community is that every developer needs to roll their own, specifically for their own game, but they seem hesitant to offer advice on the design architecture of these systems. Apparently people don't know any good answers to this problem. Advice on serialization or storage is plentiful, but those pieces are just smaller parts of the problem domain, and there is precious little advice on how to fit it all together.

This might be caused by the the fact that this community is mostly comprised of independent game developers, most of which take a more ad hoc approach to game development. Most of them just try something and see what works and what doesn't, without documenting these for future reference for companions in the field. Communities like GameDev.net have more in depth discussion though they seem to suffer from the same issues.

Apart from that, it seems that every single developer has a different definition of what constitutes the problem of persistence. Some will suggest techniques for writing data to a filesystem or PlayerPrefs, while others will point to serialization functionality. Most do not seem to realize that these are just small parts of the bigger problem.

4.3 Sub problems

During this preliminary research it was found that the domain of persistence in games consists of four somewhat related and overlapping problems:

4.3.1 Extraction

This problem is the first problem encountered on implementing persistence, though it is often overlooked because it is quite an abstract concept. In those cases it causes problems with serialization and storage in the following steps because it was not properly thought out from the start.

We will define extraction here as the separating and collecting of the relevant data to save from the transient and other irrelevant data that does not need to be saved. This problem sounds trivial but it rarely is in game development.

Most of the relevant data will, in a lot of situations, be spread around in different parts of the game, for performance reasons and ease of development for both coders and designers. The relevant data might be spread among different game objects, with each a number of components containing data that might need to be persisted. This makes it difficult to gather this data and to separate it from the garbage.

Three approaches to this problem were found. These are summarized in the following:

Separate domain objects

Every game entity maintains its own data object and is responsible for keeping the data in that object synchronized with the current game state. Upon saving, the persistence layer gathers these and all the data inside them is persisted.

Most common outside of game development, but unwieldy in our situation.

Reflection

Every game entity marks the data fields it wants persisted with an annotation telling the persistence layer that that field should be persisted. The persistence layer will ignore all other fields. The persistence layer uses reflection to inspect these annotations.[2]

A bit experimental but potentially quite flexible and powerful.

Manifest

Similar to separate domain objects, with the exception that there is just one manifest object and during saving, all relevant data is attached to this manifest in data objects. The entire object graph, with the manifest object as the root, is then easily serialized to storage. Game entities are again responsible for attaching the right data to the manifest.

This seems to be quite common among games currently, but requires a lot of work and results in a messy, hard to maintain code-base. The concept is sometimes referred to as a manifest.[7][9]

4.3.2 Serialization

Serialization is the most obvious problem most developers will consider when the time comes to implement persistence. It is a tricky domain with a lot of gotchas. While the basics are easy to solve by using standard libraries, there are a few edge cases which will cause problems if one is not careful. Datastructures with circular references will commonly cause problems during serialization. Also some objects simply can not be serialized, such as in Unity the GameObject and objects inheriting from Component. This is why the extraction step is so important.

Depending on your environment and needs, the serialization method to use is usually quite obvious. Usually the heavy lifting of converting data to a sequential format can be done by a standard library, provided a couple of rules are followed in the objects you want to serialize. For Unity and .NET there are quite a few options. The most promising ones are summarized here:

XML

We can use the standard functionality provided by XmlWriter and XmlSerializer to serialize to the XML text format. The advantage of this format is that it is easily human readable, which helps spotting errors and other bugs during developing. However, it is also very verbose, which results in larger file sizes than needed.[2]

JSON

Another text format is JSON. It has advantages in being less verbose than XML but still human readable, which makes it ideal for communication with a server over the web. It is a bit more limited in functionality and support than XML. The third party Json.NET library seems a good fit.[13]

Unfortunately, this and most other third-party serialization libraries did not seem to work on iOS for varying reasons, which means on that platform I was not able to use this, but it still works great for PC.

Binary

We can use the standard functionality provided by BinaryFormatter to serialize straight to a binary stream savable to a file. The advantage of this is that it results in a small filesize. The disadvantage is that it is not human readable and thus difficult to debug if there are problems.[2]

4.3.3 Storage

Storage is probably the most obvious but also the easiest part of the persistence problem. There are a number of options.

Filesystem or Remote Server

The first thing that comes to mind is to save the resulting data from serialization to a file, which can easily be done in our case using FileStream. It is also possible to send it to a remote server for storage using a HTTP Post request or some other protocol.[2]

PlayerPrefs

In the case of a text data stream, it can easily be dumped to a string and stored under a key in PlayerPrefs. This is an easy was to save data using standard Unity functionality, though it will only work on text.

4.3.4 Reconstruction

Reconstruction is in a lot of cases one of the most difficult problems related to persistence. In most cases, the game state consists of a lot more than just the data that was persisted. Merely restoring the saved data is usually not difficult. The loading process is usually very similar to the extracting, serializing and storing process in reverse.

The difficult part comes after the relevant data is loaded. Usually the game state consists of much more than just the loaded data. Objects need to be spawned and positioned. Various things need to be reconnected to each other. Certain functionality will need to be initialized. This means there will always be a large custom aspect specific to the game entities in this problem area. In other words, this problem is quite hard to generalize and automate, so the game developer will need to implement game specific functionality for this. The effort by the game developer needed for this should be as low as possible and will be one measure of how successful this persistence layer is.

The following approaches can help with this problem. These are more like incremental methods of solving this problem in steps. Note that these are not mutually exclusive, and combining these is probably the best course of action.

Entity Interface

Every game entity implements an interface with several methods. These four methods will be called by the persistence layer before saving, after saving, before loading, and after loading. This means game entities receive signals when these events occur and are given the chance to do any preparation or initialization. This approach makes the game entities themselves responsible for restoring the game state. Game entities can use the newly loaded data to restore the old state.

Automatically restoring the scene hierarchy

Restoring the position of objects is a frequently occurring problem, so a shortcut might be wise to implement. It is possible, though complex, to persist the state of the scene hierarchy automatically along with other data. This means the entities are no longer responsible for this aspect of game state persistence.

Automatically Spawn Entities

Some game entities might not be in the scene as defined in the editor, but instead instantiated at run-time. The persistence layer should include functionality to keep track of these dynamic entities, and include a method of automatically instantiating these when needed. After that they can be filled with data and initialized the same way as regular game entities.

5. First Prototype

During the first development iteration, I took the information I gathered during the preliminary research and chose the most promising solutions to implement. My goal was to implement the most important functionality, but still keep the prototype simple enough to be able to develop it in small time frame. An important reason for this approach was the deadline of the Kenteq project, which needed this kind of functionality quite early compared to the length of this assignment.

5.1 Chosen solutions

For the first prototype I chose the following solutions from the earlier analysis.

- Reflection and annotations to solve the extraction problem. Because it seems like the most versatile solution and should be simpler to use, though a little bit more complex to implement in the persistence layer than the other options.

- XML was chosen for serialization, because easy debugging due to human readability is valuable during prototyping and incremental development. Functionality to handle this is also included by default in .NET and widely supported.

- For storage I chose to save the data to the filesystem, because this is the most simple storage method, and again because it is easily to analyze manually in case of bugs. This was later changed to send data to a server.

- For reconstruction the interface with methods for persistence events was chosen. This was simple to implement.

5.2 Design and Implementation

The design of this prototype is pretty simple. It basically consists of four parts:

1. The IEntity interface and game entities implementing it.

2. The PersistenceManager, which manages the system. Outside parts use this to save to persist the game state.

3. Interchangeable IPersisters implementations which handle the details of data storage.

4. Interchangeable IStreamEncoder implementations which serialize data to a format suitable for sequential storage.

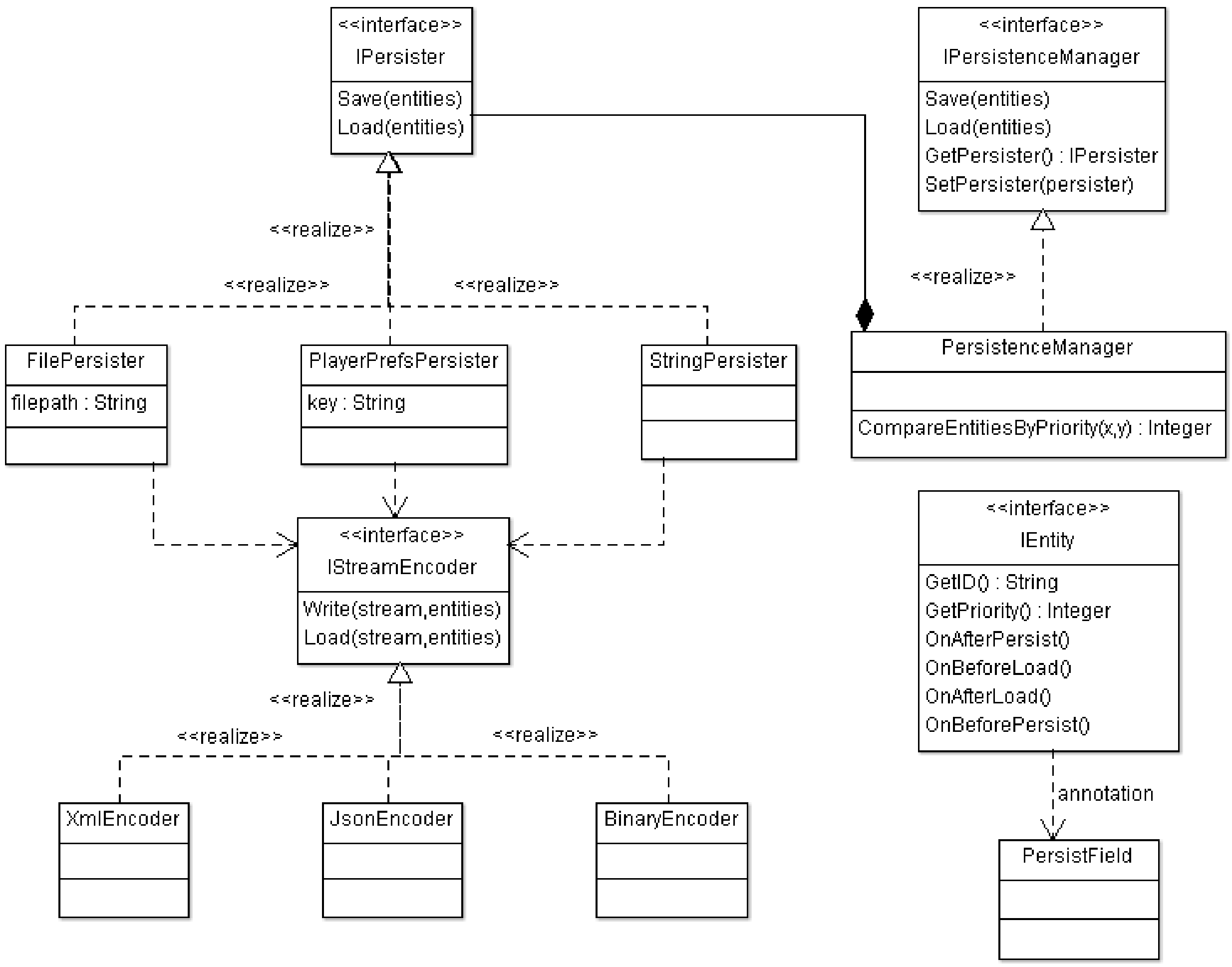


Figure 2: Design of the first prototype

The system was designed to be easily changed and expanded with exchangeable modules, which should make it easy to implement other solutions in further iterations.

5.2 Evaluation

The first prototype was developed and integrated into the Kenteq project. The best ways to test the functionality of a system are testing and using it in a real world situation with a problem it was meant to solve. This way, any defects will show themselves quickly.

The first prototype was adequate for use in the Kenteq project. There are a few defects which mean it is a bit cumbersome in some situations which can be fixed in following iterations. The strengths are also recognized and the overall system works as intended. Integration into the Kenteq was quite straightforward, apart from a few minor issues.

5.2.1 Strengths

- **Separation of responsibility:** Serialization details and storage methods are largely automated by the persistence layer and isolated from the implementation of the game.

- **Ease of use:** Adding new kinds of persistent game entities is easily and quickly done by implementing IEntity and annotating fields to be persisted.

- **Convenience:** Properties can also be persisted which allows for neat tricks making it easy to persist data from fields in related objects.

- **Flexibility:** IPersisters can easily be exchanged for different implementations to change the behavior of the system.

5.2.2 Weaknesses

- **Dynamic entities:** these were not properly supported yet. This should be implemented in the next iteration. Though not ideal, a workaround was to have a manager entity to handle dynamic game elements. (a vehicle track and numerous track pieces)

- **Dependencies:** Some entities turned out to require references to other initialized game entities upon initialization after loading. It was impossible to guarantee the order of loading of entities initially as this problem was not anticipated. The chosen solution was to give each entity a priority. Whether this is the best way to deal with this requires further thought.

- **Integration:** By striving to isolate the persistence details from the rest of the game, the design makes it difficult to to supply persistence options, such as the path of the save file. It made integration with an online saving system a bit more difficult than it should have been. This aspect of the design should be reconsidered.

5.2.3 Other Remarks

- **IEntity:** This interface now contains four methods that need to be implemented, even if they are not really needed for that particular game entity's functionality, which can be annoying to the developer using this system. It might be better to remove these from the interface and instead, call them using Reflection, if present. (This is similar to how Unity calls methods in classes derived from MonoBehaviour, which also uses reflection)

- **Design:** At this time most of the extracting of data and serializing are both done in the encoder, which means all of it has to be duplicated when using a different encoding. It would be better to separate these two issues to different parts of the system.

- **Encoding:** Near the end we decided to switch to JSON because it would be easier to deal with for a server where the saved data would be stored. This switch was relatively easy.

- **Polymorphism:** Near the end there was an issue with subtype polymorphism in serialization. Polymorphism is a concept in object oriented programming which allows similar treatment of different object types through inheritance. Our particular issue involved a list containing various different types of objects describing track pieces. This reminds us that even though the actual serialization mechanism is isolated from the rest of the game using this persistence layer, the developer still needs to be aware of the rules of the serialization method used when marking persistent fields. Fortunately the Json.NET library for serialization was able to handle this with some adjustments.

- **Savable:** This is not really related to the current prototype but more an extra idea to consider. A colleague offered the idea of a savable object which could be attached to an entity and would contain a list of strings describing the fields to save in the entity. This is a more data driven approach to defining what should be persisted compared to the current method, which uses code annotations. Both methods have their pros and cons.

5.2.4 Conclusion

The prototype was a success and used for it's intended purpose, but is still missing key features which it will need to be really helpful in all common use cases. This is to be expected from an initial prototype.

Right now, the highest priorities are support of dynamic entities and restoring the hierarchy.

The prototype was integrated in the Kenteq project, now released as Kenteq Craft. Figure three shows a section of track for a roller-coaster the player has built. My system is used to assist in saving and loading this information, and restoring the scene to its original state.

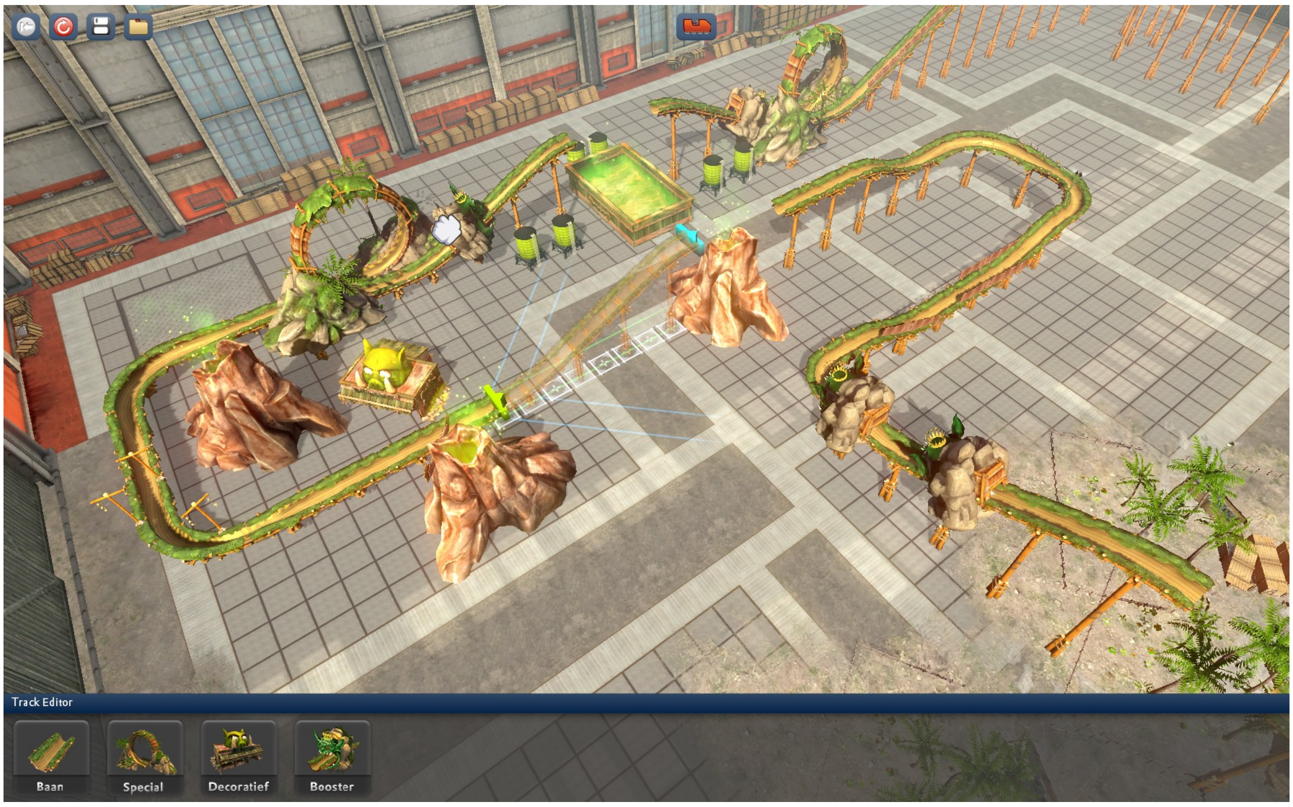


Figure 3: Screenshot of the game Kenteq Craft which includes this system for some of its functionality.

6. Second Prototype

For the second prototype, all the strong and weak points of the first prototype were considered. The lessons learned were used to create the second prototype. The second prototype is in large part simply a continuation of the first prototype, with expanded functionality, and improvements in some key areas.

6.1 Chosen Solutions

The strengths and weaknesses of the previous prototype have been described in the previous analysis. This section describes how the most important problems were solved.

In addition the design was modified with using new insight from the previous prototype, and all three encoders were implemented.

6.1.1 Dynamic Entities Instantiation

The biggest weakness of the original prototype was the fact that it was only able to fill existing entities with persistent data from storage and not able to instantiate new entities in the scene if this was required for the game. Initially, a work around was used to mitigate this problem.

In the new prototype, this problem has been solved by the addition of a Type property and the IEntityFactory interface. Developers using this persistence layer can implement an entity factory and plug it into the system to add enable instantiating dynamic entities from prefabs or using other methods.

This solution was chosen because, depending on the game and the design and implementation of game entities, instantiating an entity is likely to be quite complex. In Unity, game entities are likely to be a component inherited from MonoBehaviour, which means that they can not be instantiated normally using a constructor. Instead, they usually need to be instantiated from a prefab, together with meshes, colliders and other components a game element is composed of.

The type property allows the EntityFactory to select the right prefab, even though the entity classes of different prefabs might be the same. For example, a red car and a blue car might each have their own prefab, but both are a Car entity.

6.1.2 Handling References between Entities

With dynamic entity instantiation, there occurs a related problem, namely the handling of references between entities. Ideally, we want connections between entities to be restored upon loading. Since entities are instantiated elsewhere, we would need to override the serialization behavior to return references to these already existing entities. This is a very useful feature to have when dealing with collections of entities.

An appropriate example would be a Track entity which maintains a list of all TrackPiece entities.

This presented a bit of a problem because these references to entities might be embedded quite deep in the persistent data fields of an entity. This means that the serialization system needs to deal with this somehow. Whatever serialization mechanism is used needs a way to detect the presence of a reference to a game entity, and instead of trying to serialize it, must write the id of this entity, because the entity itself will be handled manually by the rest of the persistence layer. Upon loading, the serialization mechanism will again need to recognize that it should not attempt to deserialize references to entities in the normal way, but instead call back up to the persistence layer to obtain a reference to the entity with that id.

This was accomplished in two of the three implemented encoders. In the JsonEncoder the JsonConverter class could be overridden to change the behaviour of the JsonSerializer.

Likewise, in the BinaryEncoder, the ISerializationSurrogate interface was used in a similiar way.

Unfortunately the XmlSerializer used by XmlEncoder did not support any feature like this and the more modern XmlObjectSerializer is not included in the version of Mono that ships with Unity, so this encoder does not support serialization of references between entities.

6.1.3 Restoring the hierarchy

With dynamic entity instantiation, another problem that crops up is the scene hierarchy. In an variety of situations, it might be needed to restore an entity to its proper location in the hierarchy. This feature was attempted, but abandoned due to several reasons.

Writing the hierarchy data did not prove to be a significant problem. Recursion was used to filter the transforms of all entities and their parents. These were written to the save file and loaded again.

Unfortunately restoring the hierarchy correctly in the scene proved infeasible because of limitations in Unity. This stems from the fact that individual transforms in the scene do not have any unique identifiers. Different transforms can have the same names which can lead to unpredictable behavior when loading since different transforms can not be distinguished from each other. The other properties of a transform are also not unique. Hanford Lemoor[11] had an intriguing approach to this problem, though even this was not guaranteed to work for all situations.

The only way to really implement this feature correctly would be to assign a unique identifier to each transform upon saving and upon loading to load every single one back into the scene. Obviously this does not work in cases where some entities are already defined in the scene editor as they would be duplicated instead of merely filled with the right data. This solution would make saving and loading an all-or-nothing approach and does not fit with Unity's design philosophy in my opinion.

Another, less important reason to scrap this feature was that saving the entire hierarchy included quite a lot of redundant data that might not be needed. Instead, having each entity manage it's own location in the hierarchy if in any way relevant was considered a better option. This way only needed data is saved. This option is slightly less friendly to the developer using this system but offers a lot more flexibility in return. The developer can make his own judgment on how to handle this issue on a case by case basis.

Near the end of the project, I was notified of a GetInstanceID method on components in Unity, which might have allowed partial restoration of the hierarchy. I still can't fathom how I ever overlooked this little thing. Still, this property is read-only, which makes tracking id's and restoring them a very messy ordeal. By the time I noticed restoring the hierarchy was possible, I was nearing the deadline for my thesis and had to prioritize other issues and finalizing my thesis.

6.1.4 Integration

To enable easier integration with other systems, a change in the design was necessary to make the method of storage more accessible, so that parameters can more easily be changed and data can be exchanged. The new design is further described in the following section.

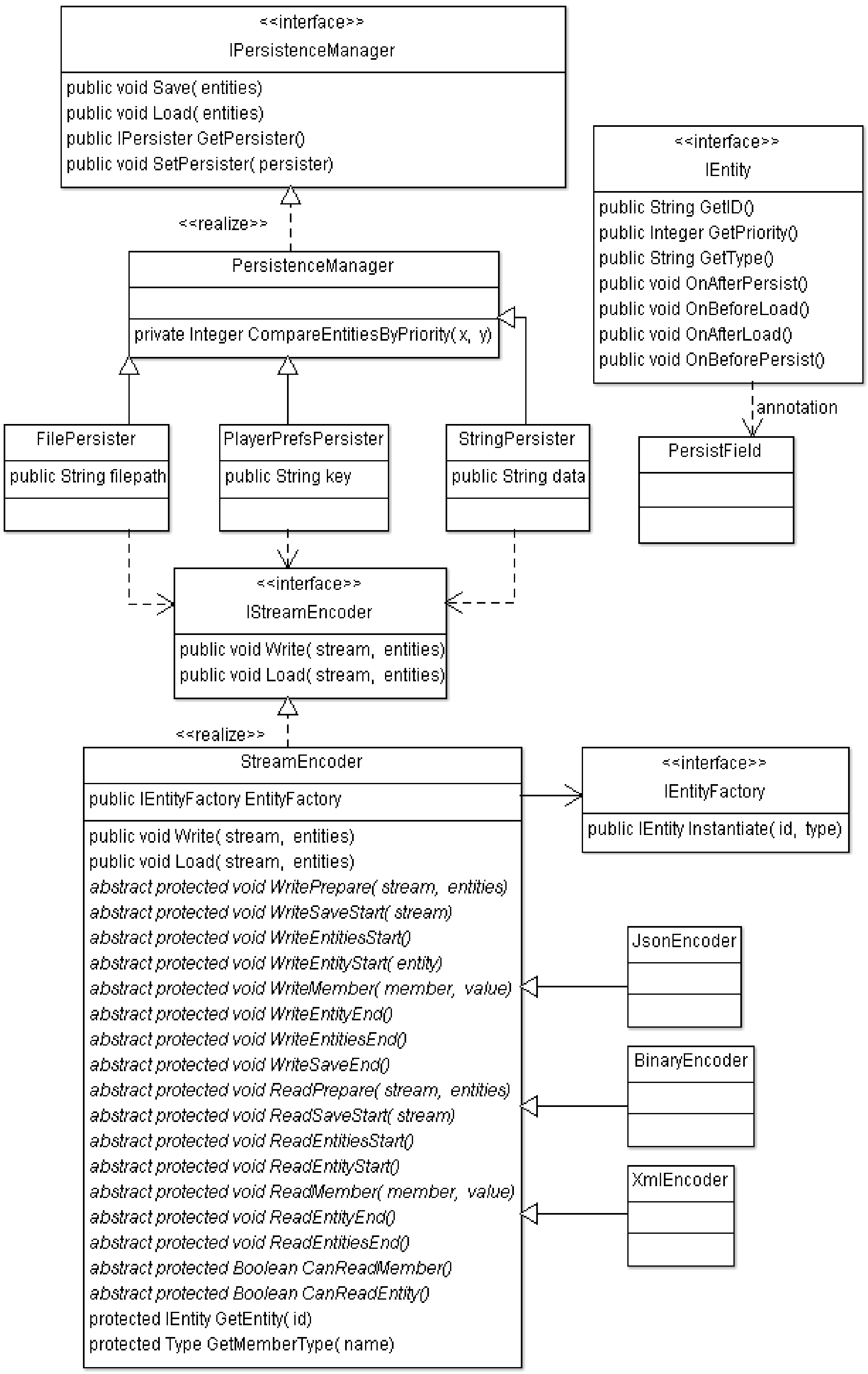
6.2 Design

The design of the system was changed quite a bit compared to the initial design.

The IPersister and IPersister manager and implementing classes have been merged. This makes it easier to access the storage options or data from outside.

In addition, there is now an abstract base class StreamEncoder, which handles extracting the data from entities and instantiating them. Subclasses inheriting from StreamEncoder override abstract methods to implement serialization details and writing and reading all data to and from the stream. This separates the actual extraction and restoration work from the serialization method chosen, even though they are still tightly related.

The additional feature of handling references between objects required a change in the general algorithm for loading entities and restoring the scene. This now happens in two passes, first instantiating all entities, followed by filling all entities with data and reconnecting references between entities.



<<interface>>
IPersistenceManager
public void Save(entities)
public void Load(entities)
public IPersister GetPersister()
public void SetPersister(persister)
<<interface>>
IEntity
public String GetID()
public Integer GetPriority()
public String GetType()
public void OnAfterPersist()
public void OnBeforeLoad()
public void OnAfterLoad()
public void OnBeforePersist()
<<realize>>
PersistenceManager
private Integer CompareEntitiesByPriority(x, y)
annotation
FilePersister
public String filepath
PlayerPrefsPersister
public String key
StringPersister
public String data
PersistField
<<interface>>
IStreamEncoder
public void Write(stream, entities)
public void Load(stream, entities)
<<realize>>
StreamEncoder
public IEntityFactory EntityFactory
public void Write(stream, entities)
public void Load(stream, entities)
abstract protected void WritePrepare(stream, entities)
abstract protected void WriteSaveStart(stream)
abstract protected void WriteEntitiesStart()
abstract protected void WriteEntityStart(entity)
abstract protected void WriteMember(member, value)
abstract protected void WriteEntityEnd()
abstract protected void WriteEntitiesEnd()
abstract protected void WriteSaveEnd()
abstract protected void ReadPrepare(stream, entities)
abstract protected void ReadSaveStart(stream)
abstract protected void ReadEntitiesStart()
abstract protected void ReadEntityStart()
abstract protected void ReadMember(member, value)
abstract protected void ReadEntityEnd()
abstract protected void ReadEntitiesEnd()
abstract protected Boolean CanReadMember()
abstract protected Boolean CanReadEntity()
protected IEntity GetEntity(id)
protected Type GetMemberType(name)
<<interface>>
IEntityFactory
public IEntity Instantiate(id, type)
JsonEncoder
BinaryEncoder
XmlEncoder

Figure 4: Design of the second prototype.

6.3 Evaluation

Since the original Kenteq project is pretty much complete right now, testing these changes is better done somewhere else. Having a smaller project also helps with testing only the functionality of this system.

To test and evaluate this prototype, a new project was created containing a test scene with a collection of entities implemented to cover all functionality during testing. The following features were tested in the following ways:

- Persist static entities. Saving and restoring entities defined in the scene in the editor. Three moving Ball entities were placed into the scene and given persistent properties for position, orientation, scale and velocity.

- Persist dynamic entities. Saving and restoring entities instantiated at run-time. A BallSpawner entity was added to the scene which instantiates smaller ball entities on the press of a button.

- References. Saving and restoring persistent references between entities. The BallSpawner was given a list with references to all balls it spawned. Balls were given a reference to the last Ball they collided with. All of these references were rendered by drawing lines in the scene for easy visual confirmation.

Together this test scene and it's contents covered all features I designed into this system. Static Balls are correctly restored to the right position on load. Spawned balls are instantiated correctly and restored to the the right position as well. All references among entities are restored correctly.

6.3.1 Strengths

All strengths of the first prototype, plus the following:

- Support for dynamic entities

- Support for persistent references between entities

- A more flexible design with less code duplication

6.3.2 Weaknesses

- The game object hierarchy is not restored automatically. In situations where this is needed, the developer will need to take care of that himself, usually by implementing entities' OnAfterLoad methods.
  Note that transform members like position can still be easily persisted by wrapping them into a property. Restoring relevant parts of the hierarchy could be as simple as merely attaching the entity's transform to a parent entity.

- Some serialization mechanisms can not support references between entities and other features like polymorphism. At the moment this is the case with the XmlEncoder. Unfortunately, this encoder can not support this because the XmlSerializer is simply not flexible enough to allow this.

- Special care still needs to be taken to ensure the value types of persistent fields and properties are serializable in the chosen serialization mechanism. Every serializer demands its own rules and constraints for serializable types. This really can not be avoided, though there might be some way to alert the developer of any issues at an early stage, instead of run-time exceptions.

- Recently another use case was discovered which, in hindsight, should have been considered. That is the case where game entities are defined in the scene and removed during run-time. An example of this would be pickups that vanish when the player touches them. This should be handled in the final version but is not a difficult problem.

6.4 Conclusion

The current system is usable for most of the use cases it is likely going to deal with. One exception is the use case for removed entities, which was not initially considered, but handling this should not be difficult so it will likely work in the final version.

7. Final Solution

The second prototype was already very satisfactory, with very few weaknesses. Unfortunately some of those weaknesses are impossible to solve entirely, but the most important ones were fixed in the final solution. This time the design did not undergo any significant changes.

7.1 Chosen Solutions

Some of the weaknesses of the second prototype were fixed using the following solutions.

7.1.1 Removing Dynamic Entities

This was more a leftover edge-case from the previous prototype than an entirely new problem, and as such was quite easy to solve. The second prototype introduced spawning of entities into the scene if they were in the save but not in the scene. The reverse should also of course be possible. That is the situation where the scene contains entities which have already been picked up or otherwise removed in the saved game.

To solve this I introduced a new method in the IEntity interface, named RemoveSelf, which the developer can use to do any cleanup that might be necessary before removing the game entity from the scene. The loading algorithm now compares the current entities in the scene with those in the save file and removes any that are not present in the save from the scene.

7.1.2 Robust Exception handling

A problem of the previous prototype was that it seemed quite fragile. A small mistake in implementing an entity (like trying to persist a type that cannot be serialized) would result in a largely incomprehensible stack trace with little information on where the problem occurred. These situations are hard to avoid due to the fact that there will always be bugs during development of a game. Garbage in, garbage out. What we can do is make the problem more transparent. In the final version this was solved by adding re-throws to insert this information in the stack trace for debugging. The developer can now easily distinguish where a problem occurred and which entity or field it relates to.

7.2 Evaluation

This time we did a more thorough evaluation of the entire system than for the two prototypes.

7.2.1 Strengths

- Dynamic entities can now be added and removed and persisted correctly
- Errors are more informative. The developer using this system can more easily fix his mistakes.

7.2.2 Weaknesses

- Having to assign a unique identifier to every instantiated entity is a bit of a bother. It also seems to be prone to bugs. Would it not be nice if every entity would be assigned his own identifier automatically?

- Having multiple Entity components on one game object will not work correctly if these are intended to be dynamic entities. This is impossible due to the way the system is designed. This use case seems nonsensical, since multiple entity components on a single game object can not be separated which makes them effectively a single game entity anyway. There might be valid use cases for this but they are rare and can also be solved in other ways.

- The system seems to work only partially on iOS devices. The Json.NET library binary is incompatible and building it from source for iOS seems complex if it is even possible at all. The BinaryFormatter also seems to crash on run-time, because of a restriction in iOS. Apparently the BinaryFormatter uses Just-In-Time (JIT) compilation under the hood, which is not allowed on that platform.
  XmlEncoder has been tested and found to work on iOS, provided certain conditions are met. The system has yet to be tested on the android platform.

7.3 Final Changes

After the evaluation, we still noticed some easy to fix weaknesses.

The first mentioned weakness, involving assigning unique identifiers, was easily fixed by a simple change in the handling of identifiers. If an identifier is required, but not set, then one will be generated based on the name of the game object and the result of GetInstanceID.

The other weaknesses were more complex. Especially the fact that the system did not work on iOS with all features was quite a disappointment. This was not a requirement from the start, since the focus was to get it working in a game for PC. In hindsight it should have been considered that there would be a desire to use the system on mobile platforms like iOS and Android. Making the system compatible with these turns out to be much more problematic than expected. Serialization techniques are a veritable minefield on these platforms due to various limitations. The lack of JIT compilation on iOS means a lot of serialization libraries simply will not work on iOS. At least the basic functionality supported by the XmlEncoder still works.

Additionally, to make the system more maintainable and easier to use, the inline documentation and the Readme document were improved. Separate documentation files were also generated using Doxygen.

7.4 Conclusion

The overall system works well for its intended purpose. As long as certain concepts and guideline are observed, it can make development of a game with saving/loading or other functionality requiring persistence of the game state much easier than it was before. In addition, the system is quite flexible. Developers have quite a lot of freedom to decide how to integrate the system into the game and how to implement game entities.

Unfortunately the system also requires some familiarity with the chosen persistence library used with this system, to ensure that types that should be persisted are actually serializable without problems. This is something that can not really be avoided as long a serialization is used and every serialization mechanism works differently.

8. Conclusion

For this internship, me and my employer chose the assignment to research and develop a system for use in Unity games with the goal of making the task of implementing persistence functionality in games easier for developers. A lot of games use persistence in the form of saved games, and there are no best practices on how to handle this in the field of game design that I know of. This is likely the case because every game and platform is different. The software engineering field in general seems to focus more on transactional databases which are often not the best solution in games. Writing to files is more common, but there is no standard way to approach this. This means that a lot of custom solutions are developed, which are hard to reuse. The intention of this project was to develop a system to handle this problem in a standard way in the Unity game engine.

Before starting work on the project, a plan of approach and planning was made and documented along with other project details in a project initiation document. I then approached the problem by first doing research on the current methods people use to tackle this problem, both in the Unity game engine, and other environments. The problem domain was analyzed and the different sub problems identified. These were; extraction, serialization, storage, and restoration. After careful consideration of the options, several prototypes were used to iteratively develop a system solving these problems.

The first prototype was already largely functional and usable. It was integrated into the Kenteq project and performed quite well. During the next iterations, various flaws were identified and fixed until the final system was robust and had features to handle various use cases encountered in development of games. Unfortunately, some convenience features, like restoring the hierarchy automatically, had to be scrapped, because of their complexity while they were not strictly needed in the system. The system was designed to be flexible so that the game developer can use his own judgment to decide how to handle these edge cases.

Evaluation

Assignment

I personally feel that I did a good job on this project and other projects at Little Chicken. Of course there is always room for improvement in some areas.

One of my first learning moments was after analyzing the first prototype and it's integration into a game currently being developed. I noticed I had made the design too simple, with just one point of access to the system. While this seems easy to use, it actually reduced the system's flexibility and thus made it more difficult to integrate with some other parts of the game, namely the part dealing with storing the saves over the internet. The second prototype was designed differently as a result.

The second problem was while I was trying to persist parts of the scene hierarchy. This was a 'nice to have' convenience feature which unfortunately was scrapped, because I could not find any way to implement this reliably. While solving a different problem later, I encountered an aspect of Unity that would allow this after all. By that time, the deadline for this thesis was close so I no longer had time to try and implement it again. Perhaps I did not study this particular problem thoroughly enough.

The fact that the more advanced parts of the system do not work on iOS was quite a shock to me. I suspected that some encoder would have problems, which is one reason why I made the system modular, but I did not suspect that all of them would have issues. The fact that a lot of serialization mechanisms do not work on iOS due to the lack of JIT was a real wake-up call. This means the more advanced features will not be available on iOS and there is little to be done about it.

On another note: The project where the first prototype was integrated has now been partially completed and released to the client as a demo. The system I developed is being used to manage the saving and loading of user-editable tracks, with different track pieces in various configurations. This serious game is named Craft Mechatronica and is used for education.[1]

Internship

I am quite content with the environment and the support I received during my internship. The working environment has a nice atmosphere and the people are generally friendly. In the case of a problem, my mentor or others were usually happy to help or provide advice.

Getting used to working here took quite a bit of effort. I was not used to normal work days and had a lengthy commute, but I eventually grew somewhat used to this. I feel I did quite well on tasks asked of me. Sometimes things at work could get somewhat chaotic, and I had to be more assertive and show more initiative. These things I still find quite difficult.

There was also a misunderstanding about the requirement for a colleague to be present at my graduation hearing, which caused me some discomfort. I probably should have been more clear in communicating exactly what was required of the company, when I started here. Fortunately this problem was eventually solved to everyone's satisfaction.

The fact that my time was split between my own assignment and other projects meant I sometimes had difficulty focusing on a problem or planning when a task would be completed. I believe I might have a personal preference for working on tasks in sequence rather than in parallel, though I realize this is not always possible. Next to my normal assignment, I also integrated the system I developed into the Kenteq Craft game and worked on other aspects of this game, like game play, UI, and communication with a back-end server, most of which I found quite interesting. I also worked on some other projects which are still in development.

Overall I had an interesting time and I feel I learned quite a bit.

Sources

The following game project made use of the results of this project.

Kenteq Craft

A serious game developed by Little Chicken during my internship.

[1] http://www.kenteq.nl//craft

The following sources were used for reference and inspiration:

Environment Documentation

.NET documentation in the MSDN library:

[2] http://msdn.microsoft.com/en-us/library/gg145045

Discussions on the Unity forums:

On persistence in Unity in general and the (in)feasibility of out-of-the-box support:

[3] http://forum.Unity3d.com/threads/15766-Save-Game-made-easier

Several approaches to developing a game saving/loading solution:

[4] http://forum.Unity3d.com/threads/44559-Saving-Game-beyond-PlayerPrefs-XML-ISerializable-etc.

Advice on game saving/loading:

[5] http://forum.Unity3d.com/threads/6432-Javascript-serialization-examples

Discussions on UnityAnswers:

General advice on game saving/loading:

[6] http://answers.Unity3d.com/questions/8480/how-to-scrip-a-saveload-game-option.html

On a persistent manifest:

[7] http://answers.Unity3d.com/questions/163092/suggestion-for-persisting-level-state-between-load.html

Discussions on GameDev.net

On custom serialization using BinaryFormatter:

[8] http://www.gamedev.net/topic/400787-object-serialization-for-saving/

Another variant of the manifest idea:

[9] http://www.gamedev.net/topic/76487-problematic-thinking-saving-and-loading-a-game/

Discussions on StackOverflow

On XML Serialization:

[10] http://stackoverflow.com/questions/1075860/c-sharp-custom-xml-serialization

Hanford Lemoore

Interesting approach to identifying unique elements in the scene hierarchy:

[11] http://hanfordlemoore.com/v/Unity-simple-state-save-for-pick-up-objects-in-games

Silverback Productions

A design for saving game state in a systematic way:

[12] http://silverbackgames.com/blog/?p=27

Json.NET

A JSON framework for .NET:

[13] http://json.codeplex.com/

Easy Save middleware

System aiding game saving in Unity:

[14] http://u3d.as/content/moodkie/easy-save/1Sg

EZ Game Saver middleware

System aiding game saving in Unity:

[15] http://www.anbsoft.com/middleware/ezs/

Addendum I. IEntity.cs

Game entities implement IEntity allowing them to be persisted.

```
/// <summary>
/// Represents an entity in the game with a number of persistable fields.
/// </summary>
public interface IEntity
{
    /// <summary>
    /// A unique identifier for this entity
    /// </summary>
    string ID
    {
        get;
        set;
    }

    /// <summary>
    /// A string representation of the type of this entity. This can be a class type, but does
not need to be.
    /// An IEntity factory will construct dynamic entities using this property.
    ///
    /// Example: A Unity game has prefabs of different cars. Each car has the same components,
but different settings.
    /// Using the Type property, the right prefab is chosen from a list and instantiated.
    /// </summary>
    string Type
    {
        get;
    }

    /// <summary>
    /// Determines in which order entities are loaded/saved.
    /// A lower value means higher priority and thus earlier loading than entities with a higher
value and thus lower priority.
    /// </summary>
    int Priority
    {
        get;
    }

    /// <summary>
    /// Called before this entity is persisted. Use this to prepare for persisting by collecting
relevant data.
    /// </summary>
    void OnBeforePersist();

    /// <summary>
    /// Called after this entity is persisted. Use this to perform any cleanup or other stuff
after persisting.
    /// </summary>
    void OnAfterPersist();

    /// <summary>
    /// Called before this entity is loaded. Use this to do any preparation before data is
loaded into this entity.
    /// </summary>
    void OnBeforeLoad();

    /// <summary>
    /// Called after this entity is loaded. Use this to restore the scene with loaded data.
    /// </summary>
    void OnAfterLoad();

    /// <summary>
    /// Called during loading if the entity is defined in the scene but not in the save. This
means it was at some point removed during gameplay.
    /// Entities implementing this method should do one of two things:
    /// option 1: Remove themselves from the scene and return true.
    /// option 2: If the entity does not want to be removed. return false.
    /// This behaviour can be used to override the system is certain situations.
    /// </summary>
    /// <returns>true if the entity was removed, false if the entity should not be
removed.</returns>
    bool RemoveSelf();
}
```

Addendum II. Simple game entity example

Simple example of a game entity:

```
[Serializable]
class Item
{
    public int itemid;
    public float condition;
}

class Player : MonoBehaviour, IEntity
{
    // transient
    public int maxHealth = 100;

    // persisted
    [PersistField]
    public int health;
    [PersistField]
    public List<Item> inventory;

    public string id = "Player";
    public string ID
    {
        get { return id; }
    }

    string Type
    {
        get { return "Player"; }
    }

    int Priority
    {
        get { return 0; }
    }

    public void OnBeforePersist()
    {
        // Maybe remove a consumable item needed for saving the game?
    }

    public void OnAfterPersist()
    {
    }

    public void OnBeforeLoad()
    {
        // Perhaps remove any held/equipped items?
    }

    public void OnAfterLoad()
    {
        // Do something related to showing newly held/equiped items?
    }

    bool RemoveSelf()
    {
        return false; // the player will never be removed
    }
}
```

Addendum III. StreamEncoder.cs

This abstract class is an important part of this system. Different types of encoders use this as a base. Implementation details were omitted.

```
/// <summary>
/// Writes to and reads from streams in a certain encoding and format. The encoding mechanism is
determined by the implementation of the subclass.
/// Subtypes should implement the various abstract methods to handle the details of
serialization and formatting the datastream.
/// </summary>
public abstract class StreamEncoder : IStreamEncoder
{
    /// <summary>
    /// Attach an EntityFactory to handle dynamic entities.
    /// Dynamic entities are entities which may not be present in the scene when loading.
    /// These will need to be instiated. A factory takes care of that to handle any special
operations that need to be done to instantiate.
    /// </summary>
    public IEntityFactory EntityFactory
    {
        get;
        set;
    }

    /// <summary>
    /// Determines which fields/properties, public/private/static/instance/inherited are
considered for persisting.
    /// </summary>
    private const BindingFlags bindingFlags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;

    /// <summary>
    /// The entity we are currently working. Accessed by subclasses during writing and reading.
    /// </summary>
    protected IEntity CurrentEntity
    {
        get;
        private set;
    }

    /// <summary>
    /// Position of the stream at the start of the write/read operation.
    /// </summary>
    protected long StartPosition
    {
        get;
        private set;
    }

    /// <summary>
    /// All entities currently being written/read accessible by keys.
    /// </summary>
    private IDictionary<string, IEntity> _entities;

    /// <summary>
    /// Writes all persistent data in entities to the stream.
    ///
    /// The following general algorithm is used:
    /// 1. For every IEntity:
    /// 2. Gather all members that are fields and properties.
    /// 3. filter members annotated with the PersistField attribute.
    /// 4. Write entity metadata to stream. (ID, Type)
    /// 5. Write entity members to stream.
    /// </summary>
    /// <param name="stream">storage destination</param>
    /// <param name="entities">entities to be written to storage</param>
    /// <exception cref="SaveException">A problem occured during saving</exception>
    public void Write(Stream stream, IEnumerable<IEntity> entities)
    {
        // omitted
    }

    /// <summary>
    /// Reads all persistent data into entities from the stream. Instantiates entities when
needed.
    /// Removes entities if they are not present in the save.
    ///
    /// The following general algorithm is used:
    /// 1. First pass:
    /// 2. For every entity in the stream:
    ///     3. Read meta data
    ///     4. Instantiate entity using assigned EntityFactory
    /// 5. Second pass:
    /// 6. For every entity in the stream:
    ///     7. Gather all members that are fields and properties
    ///     8. filter members annotated with the PersistField attribute
    ///     9. read entity members from stream
    ///
    /// Two passes are used to allow for connecting references between entities in the second
pass.
    /// </summary>
    /// <param name="stream">storage source</param>
    /// <param name="entities">entities currently in existence</param>
    /// <exception cref="LoadException">A problem occured during loading</exception>
    public void Read(Stream stream, ICollection<IEntity> entities)
    {
        // omitted
    }

    /// <summary>
    /// Get the value type of the member with the specified name of the CurrentEntity.
    /// </summary>
    /// <param name="name">member name</param>
    /// <returns>type of the member</returns>
    protected Type GetMemberType(string name)
    {
        // omitted
    }

    /// <summary>
    /// Get entity with the specified id if it exists.
    /// </summary>
    /// <param name="id">entity id</param>
    /// <returns>entity with specified id, otherwise null</returns>
    protected IEntity GetEntity(string id)
    {
        // omitted
    }

    /// <summary>
    /// Prepare for saving in any way needed.
    /// </summary>
    /// <param name="entities"></param>
    protected virtual void WritePrepare(Stream stream, IEnumerable<IEntity> entities) { }

    /// <summary>
    /// Begin writing the save file.
    /// </summary>
    /// <param name="stream"></param>
    protected abstract void WriteSaveStart(Stream stream);

    /// <summary>
    /// Begin writing entities.
    /// </summary>
    protected abstract void WriteEntitiesStart();

    /// <summary>
    /// Write the start of an entity.
    /// </summary>
    /// <param name="entity"></param>
    protected abstract void WriteEntityStart(IEntity entity);

    /// <summary>
    /// Write entity member.
    /// </summary>
    /// <param name="membername">name of the field or property</param>
    /// <param name="value">value of this field/property in the current entity</param>
    protected abstract void WriteMember(string membername, object value);

    /// <summary>
    /// Write the end of an entity.
    /// </summary>
    protected abstract void WriteEntityEnd();

    /// <summary>
    /// Write the end of all entities.
    /// </summary>
    protected abstract void WriteEntitiesEnd();

    /// <summary>
    /// Write the end of this save and finish writing.
    /// </summary>
    protected abstract void WriteSaveEnd();

    /// <summary>
    /// Prepare for reading in any way needed.
    /// </summary>
    /// <param name="entities"></param>
    protected virtual void ReadPrepare(Stream stream, ICollection<IEntity> entities) { }

    /// <summary>
    /// Read the start of the save file.
    /// </summary>
    /// <param name="stream"></param>
    protected abstract void ReadSaveStart(Stream stream);

    /// <summary>
    /// Read the start of the entities.
    /// </summary>
    protected abstract void ReadEntitiesStart();

    /// <summary>
    /// Can we read an entity?
    /// </summary>
    protected abstract bool CanReadEntity { get; }

    /// <summary>
    /// Can we read a member?
    /// </summary>
    protected abstract bool CanReadMember { get; }

    /// <summary>
    /// Read start of an entity.
    /// </summary>
    /// <param name="id"></param>
    /// <param name="type"></param>
    protected abstract void ReadEntityStart(out string id, out string type);

    /// <summary>
    /// Read entity member.
    /// </summary>
    /// <param name="membername">returns member name</param>
    /// <param name="value">returns member value</param>
    protected abstract void ReadMember(out string membername, out object value);

    /// <summary>
    /// Read end of entity
    /// </summary>
    protected abstract void ReadEntityEnd();

    /// <summary>
    /// Read end of all entities.
    /// </summary>
    protected abstract void ReadEntitiesEnd();

    /// <summary>
    /// Read end of save and finish reading.
    /// </summary>
    protected abstract void ReadSaveEnd();
}
```