
# “Snapshot”

## Μία ολοκληρωμένη και open-source λύση σειριοποίησης δεδομένων για βιντεοπαιχνίδια σε C++

# “Snapshot”

## A complete, full-fledged and open-source video game data serialization solution in C++

**Τμήμα**: BSc (Hons) in Games Programming / GA7M5 21T2
**Κωδικός Μαθήματος:** Major Project (BSc) - CMN6302 24S1 [ATH]
**Κωδικός Εργασίας**: CMN6302.S1
**Όνομα & Κωδικός σπουδαστή:** Διαμαντής Μιχαήλ-Ευάγγελος, 13-13246
**Ημ/νία Υποβολής**: 19/08/24
**Σελίδες:** -
**Αριθμός λέξεων:** -

## Δήλωση

Με το παρόν δηλώνω ότι έγραψα την πτυχιακή εργασία μόνος μου και χωρίς τη χρήση άλλων πηγών και βοηθημάτων εκτός αυτών που αναφέρονται στο κείμενο, ως παραπομπές, αυτολεξεί ή μη, και στη βιβλιογραφική λίστα. Επίσης, η εργασία αυτή δεν έχει κατατεθεί στην παρούσα ή σε παραπλήσια μορφή για παρόμοιο σκοπό σε άλλο εκπαιδευτικό ίδρυμα ή εξεταστική επιτροπή.

19 Αυγούστου 2024, Ελλάδα, Αθήνα, Ηλιούπολη

Διαμαντής Μιχαήλ-Ευάγγελος

## Περίληψη

Η ιστορία της αποθήκευσης και τροποποίησης δεδομένων στα βιντεοπαιχνίδια είναι ενδιαφέρουσα, καθώς ακολουθεί την εξέλιξη της τεχνολογίας των παιχνιδιών και των προσδοκιών των παικτών. Με την πρόοδο της τεχνολογίας, την δημιουργία διαφόρων μηχανισμών αποθήκευσης και την ανάπτυξη των αντικειμενοστραφών γλωσσών προγραμματισμού, προέκυψε ένα σχετικά πιο σύνθετο πρόβλημα στον τομέα των βιντεοπαιχνιδιών: η αποθήκευση των σχέσεων μνήμης μεταξύ των instances των κλάσεων που δημιουργήθηκαν από τον εκάστοτε προγραμματιστή στο runtime μίας εφαρμογής. Με την πάροδο των ετών, έχουν χρησιμοποιηθεί διάφορες υλοποιήσεις και τεχνικές για την καταγραφή αυτών των σχέσεων, συμπεριλαμβανομένων απλών αρχείων JSON/XML, βάσεων δεδομένων και δυαδικών μορφών. Δεν υπήρξε όμως ένας αποτελεσματικός ή πλήρης τρόπος σειριοποίησης των δεδομένων για τη διαδικασία αυτή.

Σε αυτό το πλαίσιο, στη παρούσα διατριβή δημιουργήθηκε μια βιβλιοθήκη μεθόδων C++, η οποία, σε συνδυασμό με ένα Foreign Function Interface που την πλαισιώνει, επιτρέπει σε έναν προγραμματιστή να την ενσωματώσει εύκολα στο παιχνίδι ή το πρόγραμμά του, ανεξάρτητα από τη γλώσσα προγραμματισμού που χρησιμοποιείται. Για την επίδειξη των λειτουργιών της, αναπτύχθηκε ένας C# wrapper για χρήση στη μηχανή παιχνιδιών Unity, και ως αποτέλεσμα του σχεδιασμού γεννήθηκε μια αρχιτεκτονική για την αξιοποίηση των λειτουργιών της.

Τέλος, ο στόχος της παρούσας διπλωματικής εργασίας είναι να θέσει τις βάσεις για τη μελλοντική ανάπτυξη ενός πλήρους και πλήρως αξιόπιστου μηχανισμού αποθήκευσης, σειριοποίησης και διατήρησης των object references σε βιντεοπαιχνίδια και άλλες εφαρμογές.



## Abstract

The history of data storage and modification in video games is interesting, as it follows the evolution of game technology and player expectations. With advances in technology, the creation of various saving mechanisms, and the development of object-oriented programming languages, a relatively more complex problem has arisen in the video game domain: storing the memory relationships between instances of classes created by the developer in the runtime of an application. Over the years, various implementations and techniques have been used to serialize these relationships, including simple JSON/XML files, databases, and binary formats. However, there has not been an efficient or complete way to serialize the data for this process.

In this context, this thesis has created a C++ method library, which, combined with a Foreign Function Interface that frames it, allows a programmer to easily integrate it into their game or program, regardless of the programming language used. To demonstrate its functions, a C# wrapper was developed for use in the Unity game engine, and as a result of the design an architecture was born to exploit its functions.

Finally, the goal of this thesis is to lay the groundwork for the future development of a complete and fully reliable mechanism for storing, serializing and preserving object references in video games and other applications.

## Αναγνωρίσεις

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Γεώργιο Αναστασάκη, για την πολύτιμη καθοδήγηση και την βοήθεια που μου παρείχε καθ' όλη τη διάρκεια εκπόνησης της πτυχιακής μου εργασίας.

Επιπλέον, θα ήθελα να ευχαριστήσω εγκάρδια, φίλους και συναδέλφους για την πολύτιμη τους υποστήριξη και καθοδήγηση.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου για την πολύτιμη στήριξη που μου παρείχαν διότι χωρίς αυτή δεν θα μπορούσα να φέρω εις πέρας την ακόλουθη διατριβή.

# Πίνακας Περιεχομένων

# Πίνακας εικόνων

## Πίνακας καταχωρήσεων κώδικα

# Κεφάλαιο 1: Εισαγωγή

## 1.1 Πρόλογος

Η παρούσα διατριβή έχει ως στόχο τη δημιουργία ενός open-source μηχανισμού αποθήκευσης με βασικό στόχο την χρήση του σε βιντεοπαιχνίδια, ο οποίος δεν θα περιορίζεται από την μηχανή στην οποία λειτουργεί ή την γλώσσα την οποία χρησιμοποιεί η εκάστοτε μηχανή παιχνιδιών για κύρια χρήση της. Αυτό έχει ως απώτερο σκοπό την βοήθεια των προγραμματιστών να διαχειρίζονται εύκολα και γρήγορα τα δεδομένα προς αποθήκευση που έχουν στο παιχνίδι τους αλλά φυσικά και να μπορούν με μία φαινομενικά απλή αρχιτεκτονική να φορτώσουν αυτά τα δεδομένα πίσω στο παιχνίδι τους.

Στα πλαίσια της έρευνας αυτής, δημιουργήθηκε μία βιβλιοθήκη μεθόδων γραμμένη σε C++ 17, η οποία πλαισιώθηκε από ένα Foreign Function Interface ώστε να μπορεί να χρησιμοποιηθεί από διαφορετικές γλώσσες όπως είναι η C#, η Java ή και η Python. Μαζί με το ολοκληρωμένο πακέτο της λύσης, δημιουργήθηκε και ο C# wrapper που εγκαθιστά δυνατή τη χρήση της βιβλιοθήκης στη μηχανή παιχνιδιών Unity σε συνεργασία με μία προτεινόμενη αρχιτεκτονική χρήσης της.

Τέλος, σε μακροπρόθεσμο επίπεδο, η διατριβή αυτή αποσκοπεί στο να θέσει τα θεμέλια για τη μελλοντική δημιουργία ενός universal μηχανισμού αποθήκευσης με το δυνατότερα χαμηλό αντίκτυπο στο σύστημα του παίκτη και της μνήμης του.

## 1.2 Τρόπος προσέγγισης

Ο παρών μηχανισμός δημιουργήθηκε με γνώμονα το μικρό αντίκτυπο στο μηχάνημα που θα τρέχει η εκάστοτε εφαρμογή αλλά και με γνώμονα τον προγραμματιστή που θα κληθεί να κάνει χρήση της βιβλιοθήκης αυτής, ώστε να μπορεί εύκολα να την επεκτείνει. Για να επιτευχθεί αυτό, έγινε μία ενδελεχείς και εξονυχιστική συλλογή όλων των απαραίτητων πληροφοριών, θεωριών και μετρήσεων που χρειάζονταν για να δημιουργηθεί αυτή η βιβλιοθήκη.

Επιπλέον, εξετάστηκαν διαφορετικοί τύποι serialization δεδομένων ως προς την αποτελεσματικότητα τους και το δυνατότητα τους να διαβαστούν από τον άνθρωπο ώστε να επιλεχθεί ο πιο κατάλληλος. Μέσω αυτών των αναλύσεων παράχθηκε μία αρκετά ικανοποιητική βιβλιοθήκη, με όλα τα απαραίτητα εργαλεία για τη δημιουργία και τη διαχείριση αρχείων αποθήκευσης για παιχνίδια και όχι μόνο.

Τέλος, ένα μεγάλο προτέρημα της βιβλιοθήκης που θα παρουσιαστεί μετέπειτα, είναι η δυνατότητα της να συγκρατεί τις σχέσεις των object references που υπήρχαν κατά τη διάρκεια του serialization, μέσω ενός έξυπνου συστήματος μοναδικών κωδικών.

## 1.3 Δομή πτυχιακής εργασίας

Αρχικά, το παρόν κεφάλαιο αποτελεί την εισαγωγή και έχει ως σκοπό την παρουσίαση του ερευνητικού στόχου και την ένταξη του αναγνώστη στο θέμα. Έπειτα, ακολουθεί το κεφάλαιο του θεωρητικού πλαισίου, όπου θα γίνει η αποσαφήνιση όλων των εννοιών που ο αναγνώστης θα συναντήσει στο κείμενο αλλά παρουσιάζεται και ολόκληρη η θεωρία πάνω στην οποία θα βασιστεί η δημιουργία όλων των συστημάτων της πτυχιακής εργασίας.

Τρίτων, ακολουθεί το κεφάλαιο της μεθοδολογίας όπου διεξάγεται η ενδελεχής ανάλυση των απαιτήσεων της πλήρης λύσης, αναγράφονται τα πλήρη χαρακτηριστικά της και γίνεται ο σχεδιασμός της.

Τέταρτων, στο 4ο κεφάλαιο γίνεται και αναλύεται η πλήρης υλοποίηση όλων των τμημάτων της διατριβής αυτής, ενώ στα κεφάλαια 5 και 6 γίνονται κάποιες βασικές μετρήσεις σχετικά με το παραχθέν αρχείο της βιβλιοθήκης.

Τέλος, στο 7ο και τελευταίο κεφάλαιο τη πτυχιακής εργασίας αναφέρεται το ποσοστό επιτυχίας των λειτουργιών της βιβλιοθήκης και αναλύονται μελλοντικές δυνατότητες επέκτασης και βελτιώσεων που μπορεί να λάβει η συγκεκριμένη λύση.

# Κεφάλαιο 2: Θεωρητικό πλαίσιο

## 2.1 Εισαγωγή κεφαλαίου

Στο κεφάλαιο αυτό θα αναλυθεί το θεωρητικό πλαίσιο της εργασίας εξετάζοντας προσεκτικά τύπους serializers που χρησιμοποιούνται στην βιομηχανία, θα γίνει μία σύντομη επισκόπηση των συστημάτων serialization προς ανάλυση στο κεφάλαιο 3, καθώς και μια σειρά άλλων σχετικών πηγών.

Επιπλέον, θα οροθετηθεί η έννοια των Foreign Function Interfaces, οι λειτουργίες τους άλλα και επικείμενα σοβαρά ζητήματα του σχεδιασμού τους. Επιπροσθέτως, θα αναλυθούν οι δύο βασικές κατηγορίες των wrapper και ποια η θεωρεία τους και ο ορισμός τους.

Τέλος, θα γίνει μία ανάλυση των μηχανισμών serialization και saving συστημάτων που περιέχονται στις μηχανές δημιουργίας βιντεοπαιχνιδιών Unity και Unreal Engine για άντληση επιπλέον τεχνικών.

## 2.2 Persistent Data στην Πληροφορική

Η «διατήρηση δεδομένων»(data preservation) στην Πληροφορική είναι η διαδικασία διατήρησης αρχείων και δεδομένων σε ένα στάδιο που τα καθιστά προσβάσιμα καθ’ όλη τη διάρκεια του χρόνου. Τα δεδομένα αυτά θα πρέπει να αποθηκεύονται σε μορφές αρχείων που θα τα καθιστούν πιο χρήσιμα στο μέλλον, θα πρέπει να φυλάσσονται σε πολλές τοποθεσίες και τέλος να διατηρούνται σε ένα προστατευμένο «περιβάλλον» προκειμένου να διατηρηθούν(ref).

Συγκεκριμένα, τα μόνιμα δεδομένα (persistent data) αναφέρονται σε δεδομένα που διατηρούνται επ' αόριστόν σε μη πτητικές συσκευές αποθήκευσης (non-volatile storage), όπως μαγνητικές ταινίες, σκληρούς δίσκους, Solid State Drives αλλά και οπτικούς δίσκους. Τα δεδομένα αυτά διατηρούνται ακόμη και μετά την απενεργοποίηση της συσκευής, δηλαδή χωρίς ρεύμα, από όπου προέρχεται και η ονομασία του τύπου μνήμης τους, ονόματη Non-Volatile Memory. Φυσικά, χρησιμοποιούνται πολλές τακτικές για να εξασφαλιστεί η προσβασιμότητα και η μακροπρόθεσμη διατήρησή τους. Χρησιμοποιούνται συστήματα αρχείων για την οργάνωση των δεδομένων (file systems), εφαρμόζονται τεχνικές δημιουργίας αντιγράφων ασφαλείας, όπως είναι τα διαφορικά (differential), τα αυξητικά (incremental) ή και τα πλήρη (complete) αντίγραφα ασφαλείας, τα οποία αποθηκεύονται εντός, εκτός, είτε στο cloud, και γίνεται αντιγραφή των δεδομένων σε διάφορες συσκευές ή τοποθεσίες για λόγους ανθεκτικότητας. (ref)

## 2.3 Data Serialization

Στην υποενότητα αυτή γίνεται μία σύντομη ανασκόπηση στους ορισμούς του Serialization και Deserialization όπως αυτοί χρησιμοποιούνται στην Πληροφορική (εν. 2.3.1) και εξηγούνται οι βασικοί τύποι serialization όπως είναι οι XML, JSON, YAML και binary (εν. 2.3.2). Τέλος στην ενότητα 2.4, γίνεται μία επεξήγηση της διαχείρισης των address references από τη CPU και συγκεκριμένα της εξάλειψης των object references στη μνήμη του υπολογιστή κατά τη διαδικασία του serialization.

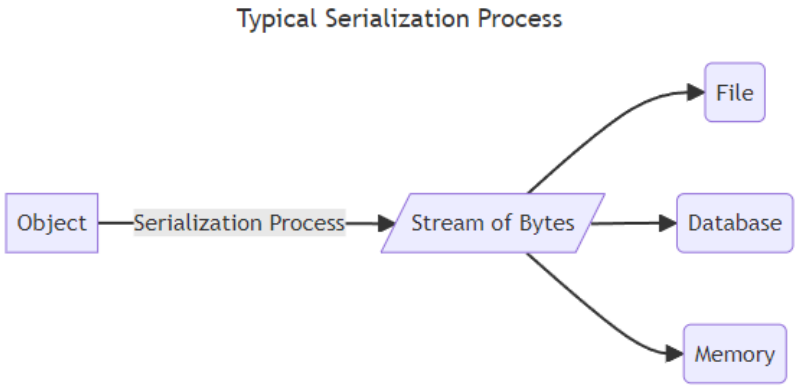
### 2.3.1 Ορισμός του Serialization και Deserialization

Στον κόσμο της Πληροφορικής, η διαδικασία της μετατροπής δεδομένων σε μία μορφή κατανοητή από έναν υπολογιστή ονομάζεται Serialization. Αναλυτικότερα, όπως διακρίνεται στην Εικόνα 1, η σειριοποίηση δεδομένων(data serialization) αναφέρεται στην πράξη του μετασχηματισμού περίπλοκων δομών δεδομένων ή καταστάσεων αντικειμένων (ref) από τη κατάσταση τους στη μνήμη σε έναν πίνακα από bytes ή σε μια μορφή που μπορεί εύκολα να καταγραφεί σε ένα αρχείο, να μεταφερθεί και να ξαναδημιουργηθεί αργότερα μέσω της αντίστροφης διαδικασίας, ονόματη Deserialization.

Συγκεκριμένα, το Deserialization είναι η διαδικασία μετάφρασης του αποθηκευμένου πίνακα ή αποθηκευμένης σειράς από bytes στην μορφή που ήταν πριν το serialization, κάτι που καθιστά τη διαδικασία του deserialization άμεσα συνδεδεμένη με τη διαδικασία του serialization. (ref)

Επιπλέον, είναι σημαντικό να αναφερθεί πως ο Serializer είναι κάτι διαφορετικό από το Serialization Format που εξάγει μετά τη διαδικασία του serialization. Το serialization format που εξάγει κάθε serializer βασίζεται σε συγκεκριμένη σύνταξη αναπαράστασης των serialized δεδομένων είτε αυτό είναι ένα JSON string, ένα YAML ή XML structure είτε είναι η binary αναπαράσταση αυτών. (ref)

Τέλος, κάθε serializer και με τη σειρά της η μορφή serialized δεδομένων που εξάγει έχει πλεονεκτήματα και εφαρμογές, όπου αυτά μπορούν να κυμαίνονται από το ποσοστό δυνατότητας ανθρώπινης ανάγνωσης μέχρι και το τελικό μέγεθος του παραχθέντος αρχείου για λόγους αποθήκευσης και μεταφοράς.(ref)




*Εικόνα 1: Serialization process*

### 2.3.2 Serialization formats

Η ακριβής σύνταξη και η δομή που χρησιμοποιείται για την αναπαράσταση των serialized δεδομένων καθορίζονται από τη μορφή του serialization. Οι μορφότυποι serialization εμφανίζουν ένα εύρος χαρακτηριστικών, όπως αναγνωσιμότητα, αποδοτικότητα και πολυπλοκότητα. Τα JSON, YAML, XML και Message Pack (binary) είναι μερικά παραδείγματα από αυτά τα μορφότυπα. Κάθε μορφή serialization περιγράφει κανόνες για την αναπαράσταση τύπων δεδομένων, την οργάνωση δομών δεδομένων και τον χειρισμό ειδικών περιπτώσεων όπως nested objects ή πίνακες, παρέχοντας έτσι μία σχεδίαση (schema) για το serialization δεδομένων σε διαφορετικά συστήματα και πλατφόρμες. (ref)

Τέλος, τα formats JSON, YAML, XML και binary διαθέτουν το καθένα μοναδικές ικανότητες προσαρμοσμένες σε συγκεκριμένες περιπτώσεις χρήσης, καλύπτοντας ποικίλες απαιτήσεις αναπαράστασης (data presentation), μετάδοσης (transmission) και αποθήκευσης δεδομένων (data preservation).

### 2.3.2a JSON Serialization

Ιστορικά, η μορφή JSON ή αλλιώς JavaScript Object Notation, είναι βασισμένη στο ανοικτό πρότυπο ECMA-262 3rd Edition 12/1999 της JavaScript (ref ECMA-262) από όπου και δημιουργήθηκε το «ECMA-404 The JSON Data Interchange Standard». Σκοπός της δημιουργίας του ήταν η μετατροπή των αναπαραστάσεων δεδομένων σε μορφή απλού κειμένου που διαβάζονται από τον άνθρωπο σε αντικείμενα ECMAScript. Οι συμβολισμοί που χρησιμοποιεί είναι παρόμοιοι με εκείνους των κοινών γλωσσών προγραμματισμού όπως η C, η C++ και η Java και είναι εντελώς ανεξάρτητη από τις άλλες γλώσσες προγραμματισμού, έτσι αποτελεί μια κατάλληλη επιλογή για τη μετάδοση δεδομένων μεταξύ συστημάτων λόγω της αναγνωσιμότητας και της απλότητάς γραφής του (ref ECMA-404). Η ομαλή ενσωμάτωση καθίσταται δυνατή χάρη στον ελαφρύ πηγαίο σχεδιασμό του και την εγγενή υποστήριξη για την πλειονότητα των γλωσσών προγραμματισμού, όπως προαναφέρθηκε. Το JSON λειτουργεί καλά σε καταστάσεις όπως τα διαδικτυακά API και τα αρχεία ρυθμίσεων, όπου είναι απαραίτητη η εύκολη ανάγνωση από τον άνθρωπο και η οργάνωση των δεδομένων.

Όπως φαίνεται στην Εικόνα 2, τα δεδομένα αποθηκεύονται σαν ζεύγη κλειδιών-τιμών (key-value pairs) και οι διαθέσιμοι τύποι δεδομένων προς αποθήκευση κυμαίνονται στα strings, στους integer/floating-point αριθμούς, στις Boolean τιμές, στις λίστες τακτοποιημένων τιμών (ordered lists), στις συλλογές μη τακτοποιημένων ζευγών κλειδιών-τιμών (unordered key-value collections) και τέλος, την αναπαράσταση της απουσίας τιμής μέσω του keyword null (ref). Αξίζει να σημειωθεί πως οι συλλογές με μη τακτοποιημένα ζεύγη κλειδιών-τιμών (unordered key-value collections) και αυτά εσωτερικά μπορούν με τη σειρά τους να αναπαραστήσουν δεδομένα με τους προαναφερθέντες τύπους. (ref)

**Καταχώρηση 1:** **sample.json**

```
{
  "string_example": "Hello, world!",
  "number_example": 42,
  "float_example": 3.14,
  "boolean_example": true,
  "array_example": [1, 2, 3, 4, 5],
  "object_example": {
    "name": "John",
```

```
 9.     "age": 30,
10.     "is_student": false
11.   },
12.   "null_example": null
13. }
14.
```

### 2.3.2β XML Serialization

Η eXtensible Markup Language (XML) είναι μια απλή, πολύ ευέλικτη μορφή κειμένου που προέρχεται από την Standard Generalized Markup Language (SGML) του ISO 8879 (ref ISO 8879).

Αρχικά, η XML αναπαριστά τα δομημένα δεδομένα με ετικέτες (tags) που περικλείονται σε αγκύλες. Αυτές οι ετικέτες οριοθετούν τη δομή των δεδομένων και μπορούν να περιλαμβάνουν χαρακτηριστικά και nested στοιχεία (elements). Κατά το serialization, τα αντικείμενα ή οι δομές δεδομένων μετατρέπονται σε μία μορφή XML με βάση το επιλεγμένο schema. Αυτό περιλαμβάνει τη διερεύνηση των ιδιοτήτων (attributes) ή των πεδίων (fields) του αντικειμένου και τη δημιουργία των αντίστοιχων στοιχείων και χαρακτηριστικών XML για την ακριβή αναπαράσταση των δεδομένων. Οι δομές XML Schema Definition (XSD) και Document Type Definition (DTD) προσφέρουν επίσημες προδιαγραφές για τον ορισμό της δομής και των περιορισμών των εγγράφων που δημιουργούνται (ref).

Επιπλέον, η XML επιτρέπει την δημιουργία custom δομών όπου μερικές από τις αλλαγές που μπορούν να επισημανθούν είναι το τελικό XML format που θα δημιουργηθεί, η διαχείριση ειδικών τύπων δεδομένων, ο καθορισμός συμβάσεων ονοματοδοσίας και η διαχείριση των namespaces (ref). Σαν serialization format, η XML βρίσκει ευρεία χρήση στις υπηρεσίες ιστού για την ανταλλαγή δεδομένων μεταξύ πελατών και διακομιστών.

Τέλος, είναι άξιο να σημειωθεί πως η XML μοιράζεται ομοιότητες με την HTML, καθώς και οι δύο είναι γλώσσες σήμανσης (markup languages) που χρησιμοποιούνται για τη δόμηση και την οργάνωση του περιεχομένου. Και οι δύο χρησιμοποιούν ετικέτες (tags) που περικλείονται σε αγκύλες για να ορίσουν τα στοιχεία μέσα σε ένα έγγραφο. Ενώ η HTML χρησιμοποιείται κυρίως για την εμφάνιση περιεχομένου στον ιστό, η XML χρησιμοποιείται για την αποθήκευση, τη μετάδοση και την ανταλλαγή δεδομένων σε διαφορετικά συστήματα και πλατφόρμες, επηρεασμένη από την προσέγγιση της HTML για τη σήμανση (markup) και τη δόμηση των δεδομένων (ref).

Ως προς τους τύπους δεδομένων που υποστηρίζονται από την XML, όπως αυτά αναπαρίστανται στην Εικόνα 3, οι πιο βασικοί είναι το text το οποίο δέχεται από απλό κείμενο μέχρι και HTML markup μέσα σε ενότητες CDATA, αριθμούς όπως integers, floating-point αριθμούς, δεκαδικούς αριθμούς αλλά και επιστημονική σημειογραφία (scientific notation).

Επιπλέον, υποστηρίζονται τιμές Boolean είτε σαν True-False είτε με 1 για True και 0 για False αλλά και η δυνατότητα αποθήκευσης δεδομένων σχετικά με την χρονολογία και την ώρα με βάση κάποιο format, όπως για παράδειγμα το "YYYY-MM-DD" για την χρονολογία (ISO 8601) (ref). Παρόλο που η XML είναι ένα format με βάση το κείμενο (text based) υποστηρίζει την αναπαράσταση δυαδικών δεδομένων (binary data) με τη χρήση τεχνικών όπως η κωδικοποίηση σε Base64 (ref) τα οποία και αποθηκεύονται σαν text μέσα στα XML στοιχεία.

Επιπροσθέτως, προσφέρει ιεραρχικές διατάξεις στοιχείων (elements) και χαρακτηριστικών (attributes), διευκολύνοντας τη δημιουργία περίπλοκων δομών δεδομένων. Ακόμη, παρέχει στους χρήστες τη δυνατότητα να προσαρμόζουν την αναπαράσταση των δεδομένων, ορίζοντας προσαρμοσμένους τύπους δεδομένων με τη χρήση elements και attributes, ικανοποιώντας έτσι τις απαιτήσεις συγκεκριμένων τομέων.

Εν κατακλείδι, η XML ενισχύει την ολοκληρωμένη διαχείριση δεδομένων, επιτρέποντας τη συμπερίληψη μεταδεδομένων (metadata ή information about data) στα έγγραφα. Αυτά τα metadata περιλαμβάνουν κρίσιμες λεπτομέρειες όπως το συγγραφικό δικαίωμα του συγγραφέα, την ημερομηνία δημιουργίας, τις πληροφορίες έκδοσης και άλλα συναφή metadata, ενισχύοντας έτσι την κατανόηση του πλαισίου και τη διαχείριση των δεδομένων. (ref)

**Καταχώρηση 2:** **sample.xml**

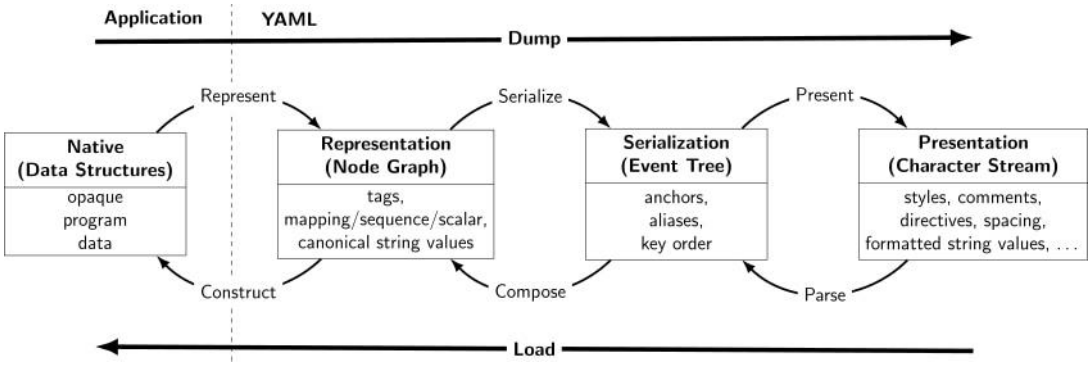
```xml
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <!-- Textual data -->
    <name>John Doe</name>

    <!-- Numeric data -->
    <age>30</age>

    <!-- Boolean value -->
    <is_student>true</is_student>

    <!-- Date and time -->
    <registration_date>2024-04-05T08:00:00</registration_date>

    <!-- Binary data (Base64 encoded) -->
    <profile_picture>base64_encoded_data_here</profile_picture>

    <!-- Structured data -->
    <address>
        <street>123 Main St</street>
        <city>New York</city>
        <state>NY</state>
        <zip_code>10001</zip_code>
    </address>

    <!-- Custom data type -->
    <product>
        <name>Laptop</name>
        <price>999.99</price>
        <manufacturer>XYZ Electronics</manufacturer>
    </product>

    <!-- Metadata -->
    <metadata>
        <author>John Smith</author>
        <creation_date>2024-04-05</creation_date>
        <version>1.0</version>
    </metadata>
</data>

```

### 2.3.2γ YAML Serialization

Η YAML, Ain't Markup Language (YAML), είναι ένα serialization format δεδομένων που χρησιμοποιείται για την αναπαράσταση δομημένων δεδομένων (structured data). Λόγο της εύκολης ανάγνωσης του από τον άνθρωπο χρησιμοποιείται συχνά σε αρχεία ρυθμίσεων, στην ανταλλαγή δεδομένων μεταξύ προγραμμάτων και σε εφαρμογές όπου η αναγνωσιμότητα από τον άνθρωπο αποτελεί προτεραιότητα.

Ιστορικά, η YAML εμπνευσμένη από την XML, δημιουργήθηκε το 2001 από μία ομάδα ερευνητών με κύριους στόχους την εύκολη ανάγνωση της από τον άνθρωπο, την εύκολη μεταφορά και μετάδοση δεδομένων μεταξύ διαφόρων γλωσσών προγραμματισμού και την ευκολία χρήσης της. (ref) Παρόλο που η YAML δεν στηρίζεται σε κάποιο προ υπάρχων standard, όπως η JSON και η XML (ref), μερικά βασικά χαρακτηριστικά του σχεδιασμού της περιέχουν την προσπάθεια ομοιογένειας μεταξύ των τύπων δεδομένων της με τις εγγενείς δομές δεδομένων (native data structures) που έχουν οι δυναμικές γλώσσες προγραμματισμού (dynamic languages), τη σχεδίαση ενός συνεπής μοντέλου το οποίο θα υποστηρίζει γενικά εργαλεία (generic tools) και την έμφαση στην εκφραστικότητα και την επεκτασιμότητα της.

Τέλος, ο μηχανισμός επεξεργασίας της με βάση την επίσημη έρευνα σχεδιασμού της στην ιστοσελίδα yaml.org, όπως αυτή αναπαρίσταται στην Εικόνα 4 (ref from yaml.org), χαρακτηρίζεται σαν μηχανισμός “one-pass processing” (ref). Αυτό σημαίνει ότι ο αναλυτής (parser) της YAML διαβάζει τα δεδομένα μόνο μία φορά από την αρχή έως το τέλος και τα επεξεργάζεται καθώς τα συναντά. (ref)




*Εικόνα 2: Επισκόπηση επεξεργασίας YAML (ref from yaml.org)*

Επιγραμματικά, οι πιο βασικοί τύποι δεδομένων που υποστηρίζει η YAML ως προς την αποθήκευση τους, όπως αυτά αναπαρίστανται στην Εικόνα 5, είναι οι κλίμακες (scalars) οι οποίες μπορούν να αποθηκεύσουν strings και multi-line strings, integers και floating-point αριθμούς, τιμές Boolean αλλά και την απουσία τιμής είτε με το keyword null ή με το σύμβολο ~ . Επιπλέον, υποστηρίζονται οι διατεταγμένες συλλογές αντικειμένων (ordered item collections) ή όπως τις ονομάζει η YAML, τα Sequences όπου μπορούν να περιέχουν κάθε αναφερόμενο τύπο δεδομένων, οι συλλογές (Mappings) από ζεύγη κλειδιών-τιμών (key-value pairs) όπου το κλειδί πρέπει να είναι μοναδικό μέσα στο σύνολο του Mapping αλλά η τιμή μπορεί να περιέχει ακόμα και nested sequences ή άλλα mappings.

Τέλος, η YAML περιέχει και τα λεγόμενα Anchors και References. Συγκεκριμένα, το χαρακτηριστικό των anchors και των references που διακρίνεται στη YAML δεν αναφέρεται στα memory address references των αντικειμένων που μόλις έγιναν serialized αλλά χρησιμοποιούνται για τη δημιουργία ψευδώνυμων (aliases) στο ίδιο το YAML αρχείο. Μέσω της χρήσης τους, αποφεύγεται η διπλοτυπία διατύπωσης των δεδομένων και αυξάνεται η ευκολία ανάγνωσης του.

**Καταχώρηση 3:** **sample.yaml**

```yaml
# Scalars
name: John Doe
age: 30
is_student: false
null_value: null

```

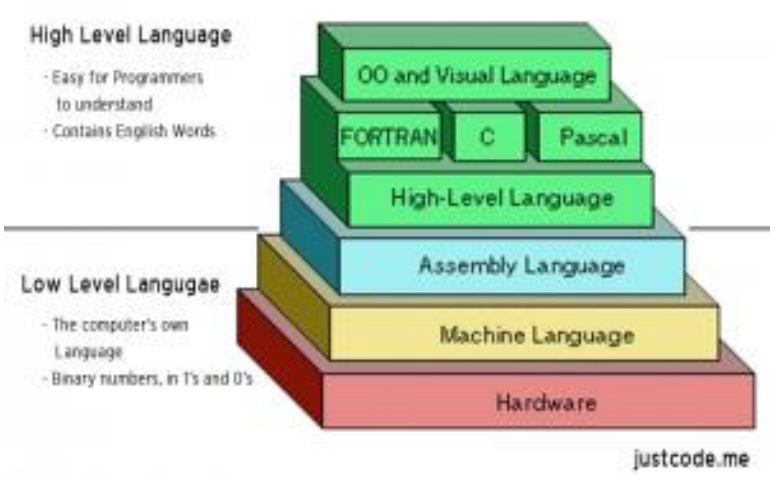
```yaml
# Sequences
hobbies:
  - Reading
  - Hiking
  - Cooking

# Mappings
address:
  city: New York
  street: 123 Main St
  zip_code: "10001"

# Anchors and References
person1: &person
  name: Alice
  age: 25

#Reference to anchor &person
person2: *person

```

### 2.3.2δ Binary Serialization

Η δυαδική σειριοποίηση (binary serialization) είναι μια διαδικασία μετατροπής δομών δεδομένων ή αντικειμένων σε δυαδική μορφή (binary), ώστε να μπορούν να αποθηκευτούν, να μεταδοθούν ή να ανακατασκευαστούν αποτελεσματικά αργότερα. Σε αντίθεση με τις μορφές JSON ή XML και YAML, οι οποίες είναι δυνατό να αναγνωστούν από έναν άνθρωπο, το binary serialization κωδικοποιεί τα δεδομένα σε μια συμπαγή, αναγνώσιμη μόνο από μηχανήματα μορφή, η οποία είναι ιδιαίτερα χρήσιμη σε σενάρια όπου η αποδοτικότητα, η ταχύτητα, το τελικό μέγεθος των serialized δεδομένων ή η μειωμένη χρήση bandwidth είναι σημαντικά ζητήματα.

Αρχικά, σε αντίθεση με τα προαναφερθέντα serialization formats και serializers, το binary serialization ως έννοια, υπάρχει εγγενώς λόγω της φύσης των υπολογιστών που χειρίζονται δυαδικά δεδομένα. Στις αρχές της πληροφορικής, το binary serialization ήταν συχνά χειροκίνητα κωδικοποιημένο, με τους προγραμματιστές να κωδικοποιούν και να αποκωδικοποιούν χειροκίνητα δομές δεδομένων σε δυαδική μορφή χρησιμοποιώντας τεχνικές χαμηλού επιπέδου (low-level), όπως bit-manipulation και πράξεις σε επίπεδο byte (ref).

Καθώς οι γλώσσες προγραμματισμού και οι πρακτικές ανάπτυξης λογισμικού ωρίμαζαν, αναπτύχθηκαν αφαιρετικές (abstractions) μέθοδοι και βιβλιοθήκες υψηλότερου επιπέδου (high-level) για το binary serialization για την απλούστευση της διαδικασίας και τη βελτίωση της παραγωγικότητας. Στην Εικόνα 6 διακρίνονται οι έννοιες low-level και high-level.




*Εικόνα 3: Low-Level και High-Level έννοιες σε γράφημα (ref)*

Ισχυρά παραδείγματα είναι οι γλώσσες C και Pascal, οι οποίες περιείχαν low-level εντολές για την ανάγνωση και γραφή δυαδικών δεδομένων σε αρχεία, ενώ μεταγενέστερες γλώσσες όπως η Java και η C# εισήγαγαν ενσωματωμένα frameworks για serialization όπως είναι το ObjectOutputStream και ObjectInputStream (ref) και ο BinaryFormatter (ref) αντίστοιχα, για την αυτοματοποίηση της διαδικασίας του serialization και τον χειρισμό πολύπλοκων δομών αντικειμένων.

Επιπροσθέτως, με βάση αυτές τις αρχές, οι binary serializers που έχουν δημιουργηθεί μπορούν να υποστηρίξουν ένα μεγάλο εύρος τύπων δεδομένων όπως (ref):

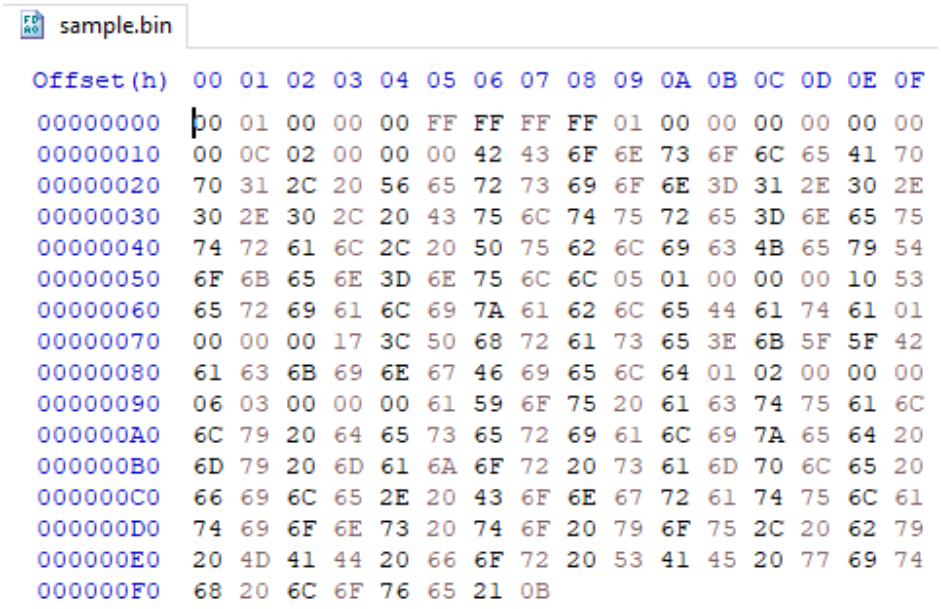
- **Primitive** τύπους: Οι binary serializers μπορούν να χειριστούν Primitive τύπους δεδομένων, όπως ακέραιους αριθμούς (π.χ. int, long, short), αριθμούς κινητής υποδιαστολής (π.χ. float, double), χαρακτήρες (π.χ. char) και τιμές Boolean.
- **Composite** τύπους: Οι binary serializers μπορούν να χειριστούν Composite τύπους δεδομένων που αποτελούνται από πολλούς primitive τύπους. Σε αυτούς περιλαμβάνονται πίνακες, λίστες, maps (ή dictionaries), sets, tuples και άλλοι collection τύποι.
- **Custom Objects**: Προσαρμοσμένα αντικείμενα ή κλάσεις, οι οποίες μπορεί να περιέχουν ένα συνδυασμό πρωτόγονων τύπων, σύνθετων τύπων και references σε άλλα αντικείμενα μπορούν επίσης να γίνουν serialized από έναν binary serializer. Οι βιβλιοθήκες serialization συχνά παρέχουν μηχανισμούς για την προσαρμογή της διαδικασίας serialization αυτών των αντικειμένων είτε μέσω κάποιου configuration αρχείου ή κάποιου custom object schema που θα δημιουργήσει ο προγραμματιστής.
- **Nullable** τύπους: Οι nullable τύποι ή οι τιμές στις οποίες μπορεί να ανατεθεί είτε ένα null reference είτε μια έγκυρη τιμή του υποκείμενου τύπου, διαχειρίζονται επίσης από τους binary serializers.
- **Strings**: Τα strings (πίνακες από char) είναι μία ιδιαίτερη περίπτωση όσον αφορά το binary serialization. Λόγο των διαφορετικών κωδικοποιήσεων όπως είναι οι UTF-8 και UTF-16 - μαζί με μία πληθώρα πολλών άλλον- συχνά οι binary serialization βιβλιοθήκες περιέχουν μηχανισμούς για τη διαχείριση του μεγέθους των string αλλά και υποκείμενα προβλήματα σχετικά με την επιλεγμένη κωδικοποίηση.

Το τελικό παραχθέν αρχείο, όπως διακρίνεται στην Εικόνα 7, δεν αποτελεί μία μορφή η οποία μπορεί να διαβαστεί από τον άνθρωπο και έτσι εάν κάποιος επιχειρήσει να το ανοίξει με κάποιον text editor όπως Notepad ή Notepad++ (ref from site) θα διακρίνει μία αλληλουχία γραμμάτων, συμβόλων, αριθμών ή και ακόμα μη κατανοητά από τον άνθρωπο σύμβολα. Αυτό το φαινόμενο συμβαίνει διότι το παραχθέν αρχείο περιέχει raw binary δεδομένα ή αλλιώς machine code (ref) τα οποία και είναι κατανοητά μόνο από έναν υπολογιστή. Με αυτό τον τρόπο παρέχεται και ένα μικρό επίπεδο ασφάλειας ως προς την ανάγνωση και τροποποίηση των δεδομένων, χωρίς όμως αυτό να αποτελεί κάποιο επίπεδο encryption (ref).


*Εικόνα 4: Δείγμα binary αρχείου, sample.bin*

Τέλος, στην Εικόνα 8, φαίνεται η αναπαράσταση του δείγματος στην Εικόνα 7 μέσω του προγράμματος HxD32 (ref) που με βάση την επίσημη ιστοσελίδα του περιγράφεται ως:

> «*HxD is a carefully designed and fast hex editor which, additionally to raw disk editing and modifying of main memory (RAM), handles files of any size*. »(ref)



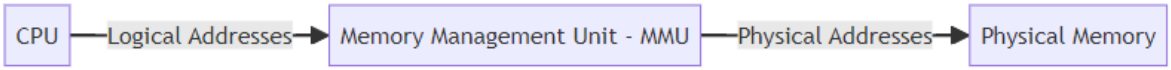

*Εικόνα 5: Inspection του αρχείου sample.bin με HxD32*

## 2.4 Memory management, addresses και serialization

Στην ενότητα αυτή γίνεται μία σύντομη ανάλυση των τρόπων με τους οποίους ένα υπολογιστικό σύστημα διαχειρίζεται τη μνήμη που του παρέχεται. Γίνεται μία αναφορά στις λογικές και φυσικές διευθύνσεις που παράγει ο επεξεργαστής, τις λειτουργίες διαχείρισης μνήμης και εικονικής μνήμης και τέλος εξηγείται τι συμβαίνει σε αυτές τις διευθύνσεις κατά τη διαδικασία του serialization.

### 2.4.1 Logical και Physical addresses

Στον τομέα των υπολογιστικών, οι λογικές διευθύνσεις (logical addresses) και οι φυσικές διευθύνσεις (physical addresses) έχουν έναν σημαντικό ρόλο στη διαδικασία διαχείρισης της μνήμης.
Οι λογικές διευθύνσεις, είναι εικονικές διευθύνσεις (virtual addresses) που δημιουργούνται από την CPU και αντιπροσωπεύουν θέσεις στο λογικό χώρο διευθύνσεων που είναι διαθέσιμες και προσβάσιμες σε ένα πρόγραμμα. Αυτές οι διευθύνσεις στην πραγματικότητα, αποτελούν μία αφηρημένη (abstracted) και ανεξάρτητη έννοια από την πραγματική διαμόρφωση της φυσικής μνήμης, παρέχοντας ένα σταθερό περιβάλλον διεπαφής για την αλληλεπίδραση των προγραμμάτων με τη μνήμη (ref). Από την άλλη πλευρά, οι φυσικές διευθύνσεις αναφέρονται στις φυσικές θέσεις στο hardware μνήμης του υπολογιστή, όπου αποθηκεύονται τα δεδομένα. Αντιστοιχούν άμεσα στα πραγματικά memory cells του hardware, καθορίζοντας τις πραγματικές θέσεις στις οποίες βρίσκονται τα δεδομένα. Η μετάφραση μεταξύ των λογικών διευθύνσεων και των φυσικών διευθύνσεων διαχειρίζεται από τη μονάδα διαχείρισης μνήμης ( Memory Management Unit - MMU) του λειτουργικού συστήματος, η οποία αντιστοιχίζει τις λογικές διευθύνσεις στις αντίστοιχες φυσικές διευθύνσεις, διευκολύνοντας την αποτελεσματική πρόσβαση και διαχείριση της μνήμης. Αυτή η διαδικασία αναπαρίσταται στην Εικόνα 9. Αυτό το επίπεδο αφαίρεσης μεταξύ λογικών και φυσικών διευθύνσεων επιτρέπει την ευέλικτη κατανομή, προστασία και αργότερα, virtualization της μνήμης στα σύγχρονα συστήματα υπολογιστών (ref).



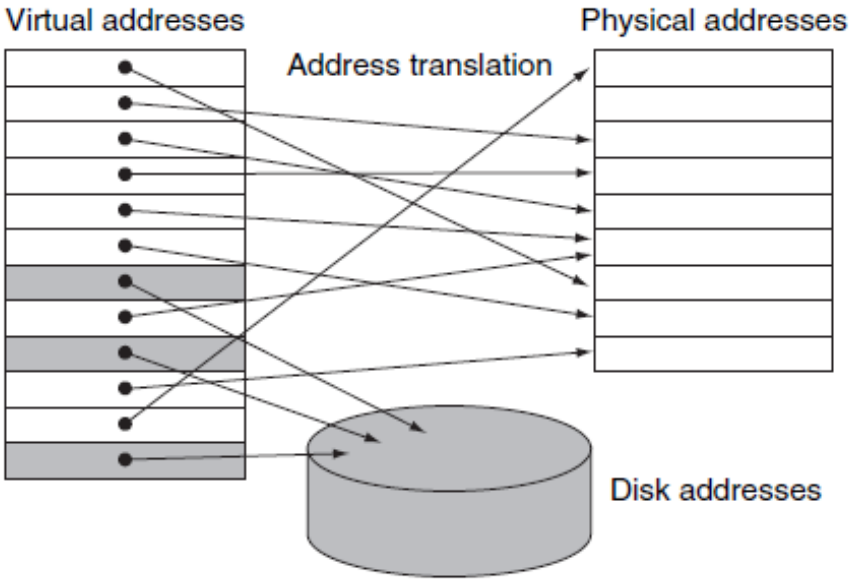

*Εικόνα 6: Address binding on the MMU (ref from book 2022)*

### 2.4.2 Memory Management και Virtual Memory σε ένα λειτουργικό σύστημα

Η διαχείριση της μνήμης σε ένα λειτουργικό σύστημα περιλαμβάνει την αποτελεσματική κατανομή και χρήση της μνήμης για την υποστήριξη των διεργασιών που εκτελούνται. Αυτό επιτυγχάνεται μέσω διαφόρων τεχνικών και schemes. Δύο βασικές τεχνικές που χρησιμοποιούνται στη διαχείριση της μνήμης είναι το paging και το segmentation. Το paging χωρίζει τη φυσική μνήμη σε μπλοκ σταθερού μεγέθους που ονομάζονται frames και τη λογική μνήμη σε μπλοκ σταθερού μεγέθους που ονομάζονται pages, επιτρέποντας την ευέλικτη κατανομή μνήμης και την αποτελεσματική χρήση της φυσικής μνήμης (ref).

Η εικονική μνήμη (virtual memory), όπως αυτή αναπαρίσταται στην Εικόνα 10, είναι μια θεμελιώδης έννοια στα λειτουργικά συστήματα που επιτρέπει την αποτελεσματική διαχείριση της μνήμης με την επέκταση της διαθέσιμης φυσικής μνήμης μέσω της χρήσης δευτερεύουσας αποθήκευσης, συνήθως ενός HDD ή ενός SSD. Στον πυρήνα της, η εικονική μνήμη επιτρέπει την εκτέλεση διεργασιών που ενδέχεται να μην χωρούν εξ ολοκλήρου στη διαθέσιμη φυσική μνήμη. Αντί να απαιτείται η ταυτόχρονη φόρτωση όλων των τμημάτων ενός προγράμματος στη μνήμη RAM, η εικονική μνήμη διαιρεί το χώρο διευθύνσεων μιας διεργασίας σε μικρότερες μονάδες που ονομάζονται pages ή segments. Αυτές οι μονάδες αντιστοιχίζονται στη συνέχεια δυναμικά μεταξύ της κύριας μνήμης και του δευτερεύοντος αποθηκευτικού

χώρου από το λειτουργικό σύστημα μέσω μίας διαδικασίας ονόματη memory mapping, επιτρέποντας στη CPU να έχει πρόσβαση στα δεδομένα ανάλογα με τις ανάγκες της. Αυτή η προσέγγιση επιτρέπει την ψευδαίσθηση ενός τεράστιου και συνεχούς χώρου μνήμης, ακόμη και όταν η φυσική μνήμη είναι περιορισμένη, με τη άμεση ανταλλαγή δεδομένων μεταξύ της RAM και του δίσκου, όπως απαιτείται. Η εικονική μνήμη παίζει καθοριστικό ρόλο σε multitasking περιβάλλοντα, επιτρέποντας την ταυτόχρονη εκτέλεση πολλαπλών διεργασιών, βελτιστοποιώντας παράλληλα τη χρήση της μνήμης και την απόδοση της.




*Εικόνα 7: Virtual memory representation (ref from book 2022)*

### 2.4.3 Τα memory addresses κατά το serialization

Όπως αναλύθηκε στις προηγούμενες ενότητες, οι διευθύνσεις μνήμης είναι συγκεκριμένες για το περιβάλλον εκτέλεσης στο οποίο εκτελείται ένα πρόγραμμα και δεν έχουν νόημα εκτός αυτού λόγω της παροδικής τους φύσης. Σε ένα εκτελούμενο πρόγραμμα, οι διευθύνσεις μνήμης εκχωρούνται δυναμικά από το MMU του λειτουργικού συστήματος καθώς οι δομές δεδομένων και οι μεταβλητές κατανέμονται στη μνήμη. Αυτές οι διευθύνσεις είναι σχετικές με το χώρο μνήμης του προγράμματος και μπορεί να αλλάζουν κάθε φορά που εκτελείται το πρόγραμμα ή ακόμη και κατά τη διάρκεια της εκτέλεσης του, καθώς η μνήμη κατανέμεται και αποδεσμεύεται δυναμικά. Κατά συνέπεια, οι διευθύνσεις μνήμης στερούνται της φορητότητας και δεν μπορούν να βασιστούν για μόνιμη αποθήκευση ή επικοινωνία μεταξύ διαφορετικών εκτελέσεων ενός προγράμματος ή μεταξύ διαφορετικών συστημάτων. Αντ' αυτού, οι διαδικασίες serialization, όπως προ αναλύθηκε στην ενότητα 2.3, επικεντρώνονται στην καταγραφή της κατάστασης των δεδομένων ενός αντικειμένου, όπως οι τιμές και η δομή του, η οποία μπορεί να ανακατασκευαστεί ανεξάρτητα από συγκεκριμένες διευθύνσεις μνήμης όταν γίνει deserialize σε άλλο περιβάλλον εκτέλεσης ή σύστημα.

## 2.5 Επισκόπηση Serializers προς ανάλυση

Στην ενότητα αυτή θα γίνει μία σχολαστική εξέταση των μεθόδων serialization δεδομένων όπου αποτελούν επιτακτική ανάγκη για τη βελτιστοποίηση της χρήσης των πόρων και τη βελτίωση της συνολικής απόδοσης ενός βιντεοπαιχνιδιού. Στο πλαίσιο αυτό, η επιλογή των κατάλληλων serializers αποκτά ύψιστη σημασία, ιδίως κατά την ανάπτυξη συστημάτων αποθήκευσης, τα οποία απαιτούν γρήγορους και αποτελεσματικούς μηχανισμούς αποθήκευσης και ανάκτησης δεδομένων. Μεταξύ της πληθώρας των διαθέσιμων επιλογών serialization, το MessagePack και το Protocol Buffers (Protobuf) αναδεικνύονται για τη συμπαγή δομή, την ταχύτητα και την ευελιξία τους.

### 2.5.1 Protocol Buffers

Το Protocol Buffers (Protobuf) είναι μια μέθοδος serialization δομημένων δεδομένων, που αναπτύχθηκε από την Google, κάτι που τον καθιστά ιδιαίτερα δημοφιλή αλλά και σταθερό. Έχει σχεδιαστεί για να είναι ένας γρήγορος, αποδοτικός και γλωσσικά ουδέτερος μηχανισμός serialization δομημένων δεδομένων, καθιστώντας τον ιδανικό για πρωτόκολλα επικοινωνίας, αποθήκευση δεδομένων και συστήματα RPC (Remote Procedure Call). Αυτό το επιτυγχάνει με την δημιουργία των λεγόμενων “messages”, όπως αυτή φαίνεται στην Εικόνα 11, όπου και παίζουν κομβικό ρόλο στην συνολική λειτουργία του serializer (ref).

**Καταχώρηση 4:** **sample.proto**

```
syntax = "proto3";

// Define a message called "Person"
message Person {
  string name = 1;
  int32 age = 2;
  string email = 3;

  // Nested message for address
  message Address {
    string street = 1;
    string city = 2;
    string postal_code = 3;
  }

  Address address = 4;

  // Field for a list of phone numbers
  repeated string phone_numbers = 5;
}

```

Τα βασικά χαρακτηριστικά που επιλέχθηκε το Protobuf προς ανάλυση είναι τα παρακάτω:

- **Αποδοτικό serialization**: Το Protobuf χρησιμοποιεί μια binary μορφή serialization, η οποία είναι πιο συμπαγής και αποδοτική σε σύγκριση με άλλες μορφές όπως η XML, YAML ή η JSON. Αυτό έχει ως αποτέλεσμα τα μικρότερα μεγέθη αρχείων μειώνοντας έτσι τις απαιτήσεις αποθήκευσης.

- **Schema Definition Language**: Το Protobuf χρησιμοποιεί μία language-agnostic γλώσσα δημιουργίας schemas για να ορίσει τη δομή των δεδομένων που γίνονται serialize. Αυτή η γλώσσα δημιουργίας schema επιτρέπει τον ορισμό πεδίων, τους τύπους αυτών και την σειρά τους μέσα στα "messages".

- **Data Streaming**: Το Protobuf υποστηρίζει data streaming, επιτρέποντας στις εφαρμογές να επεξεργάζονται αποτελεσματικά μεγάλα σύνολα δεδομένων χωρίς να φορτώνουν όλα τα δεδομένα στη μνήμη ταυτόχρονα.

- **Υποστήριξη πολλαπλών γλωσσών**: Υπάρχει υποστήριξη για πολλές γλώσσες προγραμματισμού, όπως C++, Java, Python, C#. Αυτό επιτρέπει τον ορισμό των δομών δεδομένων μία φορά στα προ αναφερθέντα αρχεία *.proto* και την χρήση τους σε διαφορετικές πλατφόρμες και γλώσσες.

- **Δημιουργία κώδικα**: Το Protobuf χρησιμοποιεί μια δυναμική τεχνική δημιουργίας κώδικα για τη δημιουργία κλάσεων μίας συγκεκριμένης γλώσσας για το serialization και το deserialization των "messages" που το αφορούν. Αυτός ο παραγόμενος κώδικας είναι ιδιαίτερα βελτιστοποιημένος για απόδοση και παρέχει ασφάλεια στους τύπους των δεδομένων.

- **Επεκτασιμότητα**: Με την δυνατότητα επέκτασης των ήδη υπάρχοντών "messages", μπορούν εύκολα να προστεθούν πεδία σε αυτά χωρίς την τροποποίηση του κυρίους schema του αρχικού "*message*". Αυτό είναι κάτι ιδιαίτερα χρήσιμο σε συστήματα που χρησιμοποιούν πολλαπλούς τύπους δεδομένων ή όταν γίνεται ενσωμάτωση τρίτων πακέτων σε μία εφαρμογή, κάτι που συμβαίνει διαρκώς στον τομέα των βιντεοπαιχνιδιών.

Συνοψίζοντας, το Protobuf προσφέρει έναν γρήγορο, αποτελεσματικό και γλωσσικά ουδέτερο τρόπο serialization δομημένων δεδομένων, με υποστήριξη πολλαπλών γλωσσών προγραμματισμού, αποτελεσματική σειριοποίηση, backwards compatibility και ενσωμάτωση με RPC frameworks (ref). Αυτό τον καθιστά μία πολύ καλή επιλογή για τη δημιουργία κατανεμημένων συστημάτων και πρωτοκόλλων επικοινωνίας, ιδιαίτερα σε εφαρμογές με κρίσιμες επιδόσεις, όπως υπηρεσίες ιστού μεγάλης κλίμακας ή βιντεοπαιχνίδια.

### 2.5.2 MsgPack C++

Το MessagePack είναι μία open-source binary μέθοδος serialization σχεδιασμένη με στόχο την αποδοτικότητα και το μικρό μέγεθος του τελικού αρχείου. Χρησιμοποιείται συχνά σε πρωτόκολλα επικοινωνίας και αποθήκευσης δεδομένων, όπου το μέγεθος και η ταχύτητα είναι κρίσιμα (ref thesis 2022). Παρόμοια με το Protobufs, το MessagePack χρησιμοποιεί structs ή κλάσεις γραμμένα στην εκάστοτε γλώσσα σαν data containers για το serialization των δεδομένων, όπως αυτό παρουσιάζεται στην Εικόνα 12.

**Καταχώρηση 5:** **sample.cpp**

```
1. #include <msgpack.hpp>
2. #include <iostream>
3. #include <vector>
4.
```

```
 5. struct Person {
 6.     std::string name;
 7.     int age;
 8.     std::vector<std::string> hobbies;
 9.
10.     // MessagePack serialization method
11.     MSGPACK_DEFINE(name, age, hobbies);
12. };
```

Τα βασικά χαρακτηριστικά που επιλέχθηκε το MessagePack προς ανάλυση είναι τα παρακάτω:

- **Συμπαγής binary μορφή**: Το MessagePack αναπαριστά τα δεδομένα σε δυαδική μορφή, που σύμφωνα με την επίσημη ιστοσελίδα του αλλά και μία έρευνα του University of Houston το 2022 (ref), τεκμηριώνεται πως είναι πιο συμπαγής σε σύγκριση με τις μορφές κειμένου όπως το JSON, XML και YAML ακόμα και σε σύγκριση με το Protobuf. Αυτή η συμπαγής μορφή μειώνει το μέγεθος των μεταδιδόμενων δεδομένων, καθιστώντας τα αποδοτικά για επικοινωνία και αποθήκευση στο δίκτυο.

- **Αποδοτικό serialization και deserialization**: Με βάση μία έρευνα του University of Houston το 2022 (ref), το MessagePack έχει έναν πολύ αποτελεσματικό αλγόριθμο serialization και deserialization. Αυτή η αποδοτικότητα είναι ιδιαίτερα επωφελής στη C++ όπου η απόδοση είναι κρίσιμη.

- **Language agnostic**: Αυτό σημαίνει ότι μπορεί να χρησιμοποιηθεί σε διάφορες γλώσσες προγραμματισμού χωρίς προβλήματα συμβατότητας λόγο της φύσης που έχει με το JSON format. Αυτό το καθιστά κατάλληλο για ετερογενή περιβάλλοντα όπου χρησιμοποιούνται πολλές γλώσσες για την ανάπτυξη μίας εφαρμογής.

- **Υποστήριξη τύπων δεδομένων**: Υπάρχει υποστήριξη για integers, floating-point numbers, συμβολοσειρές, πίνακες και maps.

- **Data Streaming**: Το MessagePack υποστηρίζει data streaming, επιτρέποντας στις εφαρμογές να επεξεργάζονται αποτελεσματικά μεγάλα σύνολα δεδομένων χωρίς να φορτώνουν όλα τα δεδομένα στη μνήμη ταυτόχρονα.

- **Open Source**: Είναι open source, κάτι που επιτρέπει στον εκάστοτε developer να σμιλέψει όλες τις διαδικασίες του όπως εκείνος επιθυμεί.

Συνοψίζοντας, το MessagePack προσφέρει μια συμπαγή, αποδοτική και γλωσσικά ανεξάρτητη μορφή serialization κατάλληλη για κρίσιμες σε απόδοση εφαρμογές. Η υποστήριξή του για διάφορους τύπους δεδομένων, το data streaming και η συμβατότητα μεταξύ πλατφορμών το καθιστούν μια ευέλικτη επιλογή για έργα που απαιτούν serialization και μετάδοση δεδομένων υψηλής απόδοσης, όπως για παράδειγμα τα βιντεοπαιχνίδια.

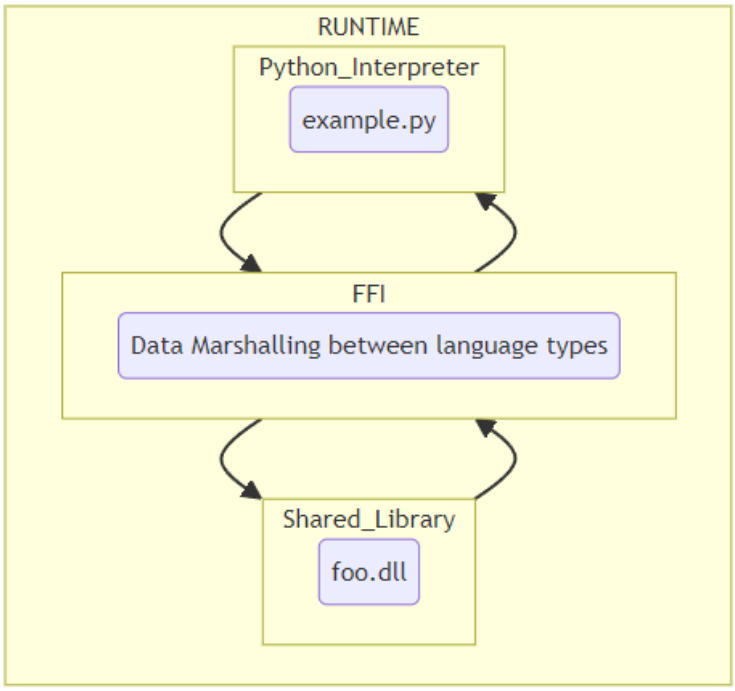
## 2.6 Foreign Function Interfaces

Στη ενότητα αυτή θα γίνει μία βασική ανάλυση των λειτουργιών των Foreign Function Interfaces (FFI) ξεκινώντας από τον ορισμό τους. Στη συνέχεια επεξηγούνται τρία βασικά σημεία που χρίζουν προσοχής κατά τη δημιουργία και τον σχεδιασμό ενός FFI με αυτά να είναι το λεγόμενο Data Marshalling, η διαχείριση των σφαλμάτων και τέλος η υπολογιστική βαρύτητα που επιφέρει το κάλεσμα μεθόδων μεταξύ διαφορετικών γλωσσών.

### 2.6.1 Ορισμός του Foreign Function Interface

Τα Foreign Function Interfaces (FFIs) αποτελούν έναν σημαντικό μηχανισμό που διευκολύνει τη δια λειτουργικότητα μεταξύ διαφορετικών γλωσσών προγραμματισμού, επιτρέποντας έτσι την απρόσκοπτη ενσωμάτωση διαφορετικών στοιχείων σε ένα γενικότερο περιβάλλον μίας εφαρμογής.
Στον τομέα ανάπτυξης λογισμικού, τα FFI χρησιμεύουν ως γέφυρες, επιτρέποντας στον κώδικα που είναι γραμμένος σε μια γλώσσα (host language) να έχει πρόσβαση σε συναρτήσεις και δομές δεδομένων που ορίζονται σε μια άλλη γλώσσα (guest language) ή και το ανάποδο. Αυτή η ικανότητα είναι καθοριστική σε σενάρια που απαιτούν τη χρήση πολλαπλών γλωσσών σε ένα ενιαίο project, όπως η αξιοποίηση των πλεονεκτημάτων απόδοσης των βιβλιοθηκών low-level γλωσσών όπως η C ή C++ από γλώσσες high-level όπως η C#, Python ή η JavaScript (ref). Η διαδικασία αυτή διακρίνεται στην Εικόνα 13.

Μέσω των FFI, οι προγραμματιστές μπορούν να ξεπεράσουν τα γλωσσικά εμπόδια, αξιοποιώντας τα πλεονεκτήματα των διαφόρων γλωσσών και εξασφαλίζοντας παράλληλα τη συμβατότητα και τη συνοχή σε ολόκληρη τη βάση του κώδικα. Ωστόσο, αυτή η αρχιτεκτονική λύση φέρει μαζί της και τα ανάλογα προβλήματα και εμπόδια, όπως είναι το data marshalling μεταξύ των δύο γλωσσών που θα πρέπει να διαχειριστεί ο εκάστοτε προγραμματιστής και το performance overhead που υπάρχει λόγω των εξωτερικών μεθόδων που καλούνται αλλά και λόγο του context switching που συμβαίνει εκείνη τη στιγμή στον επεξεργαστή. Τέλος, ένας τομέας των FFI που χρήζει σημαντικής προσοχής είναι το error handling μεταξύ των γλωσσών (ref).




*Εικόνα 8: FFI και Data Marshalling*

### 2.6.2 Data Marshalling στα Foreign Function Interfaces

Η «διακίνηση δεδομένων» ή αλλιώς Data Marshalling, περιλαμβάνει τη μετατροπή δεδομένων από μια αναπαράσταση σε μια άλλη, συνήθως με σκοπό τη μεταφορά τους μεταξύ διαφορετικών συστημάτων ή γλωσσών. Στο πλαίσιο των FFI, το data marshalling συνήθως συνδέεται άμεσα με το serialization και deserialization και χρησιμοποιείται για τη μετατροπή δεδομένων μεταξύ των αναπαραστάσεων που χρησιμοποιούνται από την καλούσα γλώσσα και την καλούμενη γλώσσα, όπως αυτό φαίνεται στην Εικόνα 13. Για παράδειγμα εάν μια συνάρτηση σε μια βιβλιοθήκη C αναμένει μία συμβολοσειρά string και καλείται από την Python, η οποία συνήθως χρησιμοποιεί συμβολοσειρές Unicode, το FFI θα πρέπει να διαμορφώσει τη συμβολοσειρά Unicode της Python σε μια μορφή που μπορεί να κατανοήσει η συνάρτηση C, όπως ένα null-terminated ASCII string.

Μια παρόμοια διαδικασία μπορεί να εννοηθεί και μεταξύ των Εικόνων 14 και 15, όπου ο προγραμματιστής θα έπρεπε να είχε δημιουργήσει και τα δυο data types για να μπορέσει να «μεταφέρει» την πληροφορία και τα δεδομένα από την πλευρά της Python στη βιβλιοθήκη της C++, με το FFI να διαχειρίζεται τη σωστή «λήψη» και «αποστολή» των δεδομένων.

**Καταχώρηση 6:** **ffiSample.py**

```
1. # Imports the necessary C types
2. import ctypes
3.
4. # Base class definition
5. class Data(ctypes.Structure):
6.     _fields_ = [
7.         ('intValue', ctypes.c_int),
8.         ('floatValue', ctypes.c_float),
```

```
 9.         ('boolValue', ctypes.c_bool)
10.     ]
11.
```

**Καταχώρηση 7:** **ffiSample.cpp**

```
 1. /* A simple C++ struct
 2. with data type mirroring
 3. of the Python Data class */
 4. struct Data
 5. {
 6.     int intValue;
 7.     float floatValue;
 8.     bool boolValue;
 9. };
10.
```

### 2.6.3 Διαχείριση των Exceptions μεταξύ των γλωσσών

Όταν καλείται μία συνάρτηση σε διαφορετικές γλώσσες, είναι σημαντικό να εξετάζεται ο τρόπος διάδοσης και χειρισμού των σφαλμάτων μεταξύ των διαφορετικών γλωσσών. Αυτό μπορεί να περιλαμβάνει τη μετατροπή των αναπαραστάσεων των σφαλμάτων μεταξύ των γλωσσών ή την παροχή μηχανισμών για τη μετάφραση των σφαλμάτων που ανακύπτουν σε μια γλώσσα σε exceptions ή κωδικούς σφαλμάτων σε μια άλλη γλώσσα (ref). Μερικοί ενδεικτικοί τρόποι σωστής διαχείρισης των σφαλμάτων μεταξύ διαφορετικών γλωσσών είναι οι ακόλουθοι:

- **Error Propagation**: Σε πολλές περιπτώσεις, τα σφάλματα που προκαλούνται από συναρτήσεις που καλούνται μέσω ενός FFI μεταδίδονται πίσω στον κώδικα που καλεί την συνάρτηση. Αυτό σημαίνει ότι εάν μια συνάρτηση που καλείται μέσω του FFI αντιμετωπίσει ένα σφάλμα, όπως ένα runtime exception ή έναν κωδικό σφάλματος, μπορεί να κάνει throw ένα exception ή να επιστρέψει μια ένδειξη σφάλματος στον καλούντα κώδικα.

- **Κωδικοί Σφαλμάτων**: Τα FFI μπορούν να χρησιμοποιήσουν κωδικούς σφαλμάτων ή ειδικές τιμές επιστροφής για να υποδεικνύουν σφάλματα. Για παράδειγμα, μια συνάρτηση που καλείται μέσω ενός FFI μπορεί να επιστρέψει μια αρνητική τιμή ή έναν δείκτη NULL για να σηματοδοτήσει μια κατάσταση σφάλματος.

### 2.6.4 Performance Overhead στα Foreign Function Interfaces

Η χρήση των FFI συνεπάγει μεταξύ άλλων και ζητήματα επιδόσεων, που αφορούν κυρίως τη μετατροπή δεδομένων, την επιβάρυνση κλήσης συναρτήσεων και τη διαχείριση μνήμης. Η μετατροπή δεδομένων μεταξύ διαφορετικών αναπαραστάσεων καθώς διασχίζει τα όρια των γλωσσών μπορεί να εισάγει μία υπολογιστική επιβάρυνση, ιδίως για μεγάλα σύνολα δεδομένων, καθώς συχνά περιλαμβάνει λειτουργίες marshalling και unmarshalling, όπως αυτές εξηγήθηκαν στην ενότητα 2.6.2. Παρομοίως, η πράξη της κλήσης συναρτήσεων μεταξύ διαφορετικών γλωσσών προσθέτει στο φορτίο επεξεργασίας λόγω βημάτων όπως η μεταφόρτωση παραμέτρων και η εναλλαγή περιβάλλοντος μεταξύ γλωσσών, το λεγόμενο context switch (ref). Επιπλέον, οι αποκλίσεις στους μηχανισμούς διαχείρισης μνήμης μεταξύ των γλωσσών

μπορεί να επηρεάσουν την απόδοση, γεγονός που απαιτεί προσεκτική εξέταση, ιδίως σε σενάρια που περιλαμβάνουν συχνές λειτουργίες μνήμης όπως ένα library με cache και δομές δεδομένων.

## 2.7 Library Wrappers

Για να πλαισιωθεί σωστά μία πλήρης λειτουργική βιβλιοθήκη με σκοπό τη χρήση της από πολλαπλές γλώσσες, είναι ζωτικής σημασίας κομμάτι της ο λεγόμενος library wrapper, γνωστός και ως DLL Wrapper. Για την σωστή κατανόηση αυτού όμως χρειάζεται η διαλεύκανση κάποιον βασικών εννοιών όπως αυτές του managed κώδικα έναντι του unmanaged κώδικα, οι οποίες αποτελούν δύο από τους βασικούς πυλώνες γνώσης σχετικά με την διαχείριση μνήμης και δεδομένων στην Πληροφορική. Σε αυτή την ενότητα θα αναλυθεί η διαφορά του Managed κώδικα έναντι του Unmanaged κώδικα, θα δοθεί ένας βασικός ορισμός σχετικά με τους DLL wrappers και τέλος θα εξηγηθούν τα βασικά περιβάλλοντα χρήσης τους μέσω διαφορετικών τρόπων διεπαφών με εξωτερικό κώδικα.

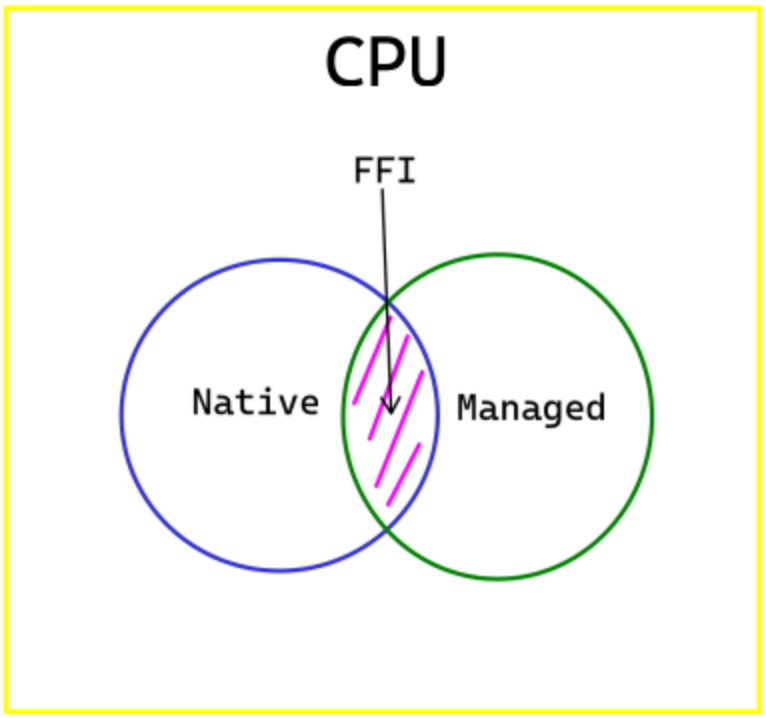
### 2.7.1 Managed και Unmanaged κώδικας

Όπως προαναφέρθηκε, ο managed και ο unmanaged κώδικας είναι θεμελιώδεις έννοιες στην ανάπτυξη λογισμικού, ιδίως στο πλαίσιο των γλωσσών και των περιβαλλόντων προγραμματισμού.

Συγκεκριμένα, ο managed κώδικας λειτουργεί σε ένα διαχειριζόμενο περιβάλλον, το οποίο συνήθως πλαισιώνεται από ένα περιβάλλον runtime, όπως το Common Language Runtime (CLR) στο .NET Framework ή η Java Virtual Machine (JVM) στη Java. Αυτό το περιβάλλον παρέχει αυτόματη διαχείριση μνήμης, συμπεριλαμβανομένων χαρακτηριστικών όπως το Garbage Collection (GC), η οποία χειρίζεται τις εργασίες κατανομής και απομάκρυνσης μνήμης. Ο managed κώδικας είναι γνωστός για το υψηλότερο επίπεδο αφαίρεσης και την ανεξαρτησία πλατφόρμας του, καθιστώντας την ανάπτυξη, το debugging και τη συντήρηση του ευκολότερη. Παραδείγματα γλωσσών που μεταγλωττίζονται σε managed κώδικα είναι η C#, η Visual Basic .NET, η Java και ορισμένες υλοποιήσεις της Python, όπως η Jython ή η IronPython όπου η τελευταία αποτελεί μία υλοποίηση της Python με bindings στη .NET (ref).

Αντίθετα, ο unmanaged, ή αλλιώς "native" κώδικας εκτελείται απευθείας στο hardware του συστήματος χωρίς την παρέμβαση ενός περιβάλλοντος runtime. Συνήθως γραμμένος σε γλώσσες όπως η C ή η C++, ο unmanaged κώδικας παρέχει στους προγραμματιστές άμεσο έλεγχο των εργασιών διαχείρισης μνήμης, συμπεριλαμβανομένης της κατανομής και της αποδέσμευσης της. Ενώ ο unmanaged κώδικας μπορεί να προσφέρει καλύτερες επιδόσεις και στενότερο έλεγχο των πόρων του συστήματος, απαιτεί προσεκτική διαχείριση της μνήμης και μπορεί να είναι ευάλωτος σε θέματα όπως memory leaks και ευπάθειες ασφαλείας, αν δεν αντιμετωπιστεί σωστά. Παραδείγματα μη διαχειριζόμενου κώδικα περιλαμβάνουν native κώδικα που έχει μεταγλωττιστεί από C ή C++, καθώς και κώδικα σε assembly (ref).

Όπως μπορεί να διακριθεί στην Εικόνα 16, υπάρχει δυνατότητα διεπαφής μεταξύ των δύο αυτών τύπων γλωσσών, μέσω του προ αναφερόμενου Foreign Function Interface (FFI) όπως αυτό αναλύθηκε στην ενότητα 2.6.




*Εικόνα 9: Γεφύρωση managed και unmanaged κώδικα μέσω ενός FFI*

Συνοψίζοντας, οι βασικές διαφορές των δύο αυτών εννοιών είναι η «ασφαλής», για τον managed κώδικα, διαχείριση της μνήμης σε αντίθεση με τον unmanaged κώδικα που δεν διαθέτει λειτουργίες όπως ο Garbage Collector για την αυτόματη αποδέσμευση αυτής και η αποδέσμευση της μνήμης πρέπει να γίνεται χειροκίνητα. Στη συνέχεια, ο unmanaged κώδικας υπερισχύει όσον αφορά την «ταχύτητα» του διότι δεν βασίζεται σε κάποιο runtime περιβάλλον, ενώ ο managed κώδικας παρέχει μία ανεξαρτησία σχετικά με την πλατφόρμα εκτέλεσης του και ευκολία ανάπτυξης.

Τέλος, ακριβώς λόγο αυτής της ανεξαρτησίας του σχετικά με την πλατφόρμα εκτέλεσης του, ο managed κώδικας είναι πιο φορητός σε διαφορετικές πλατφόρμες, ενώ ο unmanaged κώδικας μπορεί να χρειαστεί να ξανά γίνει compile εκ νέου ή να προσαρμοστεί για διαφορετικές πλατφόρμες.

### 2.7.2 Ορισμός των Library/DLL Wrappers στην Πληροφορική

Οι DLL wrappers, γνωστοί και ως dynamic-link library wrappers, είναι components λογισμικού ή βιβλιοθήκες που έχουν σχεδιαστεί για να απλοποιούν τη χρήση των dynamically-linked libraries (DLL) σε ένα συγκεκριμένο περιβάλλον προγραμματισμού. Παρέχουν έναν τρόπο για τις εφαρμογές να τμηματοποιήσουν τον κώδικα τους σε λογικές βιβλιοθήκες και να μοιραστούν πόρους (ref), όπως για παράδειγμα διακρίνεται στην Εικόνα 13.

Οι wrappers αυτοί κατά επέκταση, προσφέρουν ένα επίπεδο αφαίρεσης ή higher-level διεπαφής γύρω από τις βιβλιοθήκες DLL, καθιστώντας ευκολότερη τη χρήση τους εντός μιας συγκεκριμένης γλώσσας προγραμματισμού ή ενός συγκεκριμένου πλαισίου. Συχνά παρέχουν μια πιο φιλική προς το προγραμματιστή διεπαφή, αποκρύπτοντας τις πολυπλοκότητες της άμεσης αλληλεπίδρασης με τα DLL αυτά (ref).

Συγκεκριμένα, μέσω των wrappers μπορεί να επιτευχθεί η συμβατότητα με ποικίλες γλώσσες, για παράδειγμα μία διεπαφή ενός C++ DLL με την C# της .NET ή ακόμα και με ένα περιβάλλον Python και επιτυγχάνεται η απλοποίηση του API (Application Programming Interface) ενός σύνθετου DLL με τη χρήση higher-level γραφής και αρχιτεκτονικής. Επιπλέον, μέσα σε έναν wrapper μπορούν να ενσωματωθούν οι προαναφερθέντες μηχανισμοί exception handling (εν. 2.6.3) για τη διαχείριση των exceptions ή των κωδικών σφαλμάτων που μπορεί να επιστρέψει μία συνάρτηση από το DLL.

Τέλος, σε έναν τέτοιου τύπου wrapper μπορούν να υλοποιηθούν και μηχανισμοί σωστής δέσμευσης και αποδέσμευσης της μνήμης για την ομαλή λειτουργία της εκάστοτε εφαρμογής (ref).

### 2.7.3 Managed και Unmanaged code wrappers

Οι managed και unmanaged code wrappers ακολουθούν τη θεωρεία που παρουσιάστηκε στην ενότητα 2.7.1. Χρησιμοποιούνται και οι δύο για να γεφυρώσουν το χάσμα μεταξύ managed και unmanaged περιβαλλόντων κώδικα σε συνεργασία με τις λειτουργίες των FFIs, αλλά εξυπηρετούν διαφορετικούς σκοπούς και λειτουργούν με διαφορετικούς τρόπους. Οι managed wrappers χρησιμοποιούνται για να παρέχουν στον managed κώδικα (όπως η C#, η VB.NET, η Java ή άλλες γλώσσες) πρόσβαση σε unmanaged κώδικα, συνήθως DLL ή COM objects (ref about COM Objects). Χρησιμοποιούνται συχνά για την χρήση παλαιότερου κώδικα ή εξωτερικών βιβλιοθηκών που είναι γραμμένες σε γλώσσες όπως η C ή η C++ που δεν είναι άμεσα συμβατές με το .NET Framework. Αντίστοιχα, οι unmanaged wrappers, επίσης γνωστοί ως platform invoke (P/Invoke) wrappers, χρησιμοποιούνται για την κλήση managed κώδικα από unmanaged κώδικα, χωρίς όμως αυτό να περιορίζει τις Platform Invoke λειτουργίες μόνο σε περιβάλλοντα unmanaged κώδικα. Επιτρέπουν σε unmanaged κώδικα γραμμένο σε γλώσσες όπως η C ή η C++ να καλεί συναρτήσεις που ορίζονται σε managed σύνολα, συνήθως DLL που δημιουργούνται με γλώσσες .NET.

## 2.8 Συστήματα και λύσεις serialization στις μηχανές βιντεοπαιχνιδιών

Στη τελευταία αυτή ενότητα του θεωρητικού πλαισίου, θα γίνει μία τυπική αναφορά στους μηχανισμούς serialization ή ακόμα και ολόκληρες λύσεις που διαθέτουν ενσωματωμένες δύο από τις διασημότερες μηχανές δημιουργίας βιντεοπαιχνιδιών, με αυτές να είναι οι Unity και Unreal Engine.

Στο επόμενο κεφάλαιο της Μεθοδολογίας (Κεφ. 3), παρουσιάζονται τα χαρακτηριστικά και ο σχεδιασμός του μηχανισμού που θα αναπτυχθεί με βάση όλη την προαναφερόμενη θεωρία.

### 2.8.1 Δυνατότητες serialization στη Unity

Η Unity προσφέρει διάφορους μηχανισμούς για το serialization δεδομένων και τη δημιουργία μηχανισμών αποθήκευσης στα project της, καλύπτοντας διαφορετικές ανάγκες και πολυπλοκότητες.
Στην απλούστερη μορφή του, οι προγραμματιστές μπορούν να χρησιμοποιήσουν το PlayerPrefs (ref from docs), ένα σύστημα αποθήκευσης τιμών-κλειδιών κατάλληλο για την αποθήκευση μικρών ποσοτήτων δεδομένων, όπως οι προτιμήσεις των παικτών και οι βασικές ρυθμίσεις του παιχνιδιού που καταγράφονται στο registry των συστημάτων. Σημειώνοντας προσαρμοσμένες κλάσεις με το attribute [Serializable], οι προγραμματιστές μπορούν κάνουν serialize και να deserialize την πολύπλοκη κατάσταση του παιχνιδιού,

την πρόοδο του παίκτη και άλλα βασικά δεδομένα του παιχνιδιού. Αυτό φυσικά είναι εφικτό μόνο με τη δημιουργία ενός μηχανισμού αποθήκευσης διότι η Unity δεν παρέχει κάποιον τέτοιο μηχανισμό έτοιμο.

Τέλος, η Unity υποστηρίζει serialization σε μορφές JSON και XML μέσω των κλάσεων JsonUtility και XmlSerializer αντίστοιχα, παρέχοντας ευελιξία στις μορφές αποθήκευσης και ανταλλαγής δεδομένων (ref). Στη συνέχεια, για πιο αποδοτικό μέγεθος αρχείου και χειρισμό δεδομένων, η Unity υποστηρίζει binary serialization μέσω της κλάσης BinaryFormatter. Αυτή η μέθοδος ωστόσο δεν συστήνεται διότι με βάση το επίσημο documentation της .NET η κλάση BinaryFormatter έγινε Obsolete (ref) στην έκδοση .NET 8.

Αυτές οι επιλογές serialization δίνουν τη δυνατότητα στους προγραμματιστές να δημιουργήσουν συστήματα αποθήκευσης προσαρμοσμένα στις απαιτήσεις του παιχνιδιού τους.

### 2.8.2 Δυνατότητες serialization στην Unreal Engine

Η Unreal Engine προσφέρει μια μεγάλη ποικιλία εργαλείων για το serialization δεδομένων και τη δημιουργία συστημάτων αποθήκευσης. Πρώτον, οι προγραμματιστές μπορούν να αξιοποιήσουν την κλάση SaveGameObject για να ορίσουν προσαρμοσμένες δομές δεδομένων για την αποθήκευση της προόδου του παιχνιδιού. Αυτή η κλάση διευκολύνει το serialization των δεδομένων του παιχνιδιού και μπορεί να γίνει ακόμα και subclassed μέσω της κλάσης USaveGame έτσι ώστε να υλοποιηθούν οι απαραίτητες μέθοδοι serialization ανάλογα με τις απαιτήσεις του παιχνιδιού (ref).

Επιπλέον, η Unreal Engine παρέχει ένα σύνολο συναρτήσεων, όπως οι SaveGameToSlot() και LoadGameFromSlot(), για τη διαδικασία αποθήκευσης και φόρτωσης δεδομένων από και προς το δίσκο σε συνεργασία με τις κλάσης SaveGameObject. Η μηχανή υποστηρίζει επίσης data archiving και data compression, επιτρέποντας την αποτελεσματική αποθήκευση της κατάστασης των δεδομένων.

Τέλος, οι προγραμματιστές μπορούν να κάνουν χρήση του ενσωματωμένου συστήματος SaveGame, το οποίο αυτοματοποιεί μεγάλο μέρος της διαδικασίας αποθήκευσης και φόρτωσης, απλοποιώντας την υλοποίηση ενός συστήματος game saving στα project, έστω και με περιορισμούς.

# Κεφάλαιο 3: Μεθοδολογία

## 3.1 Εισαγωγή κεφαλαίου

Στο κεφάλαιο αυτό θα γίνει μία ενδελεχής ανάλυση των απαιτήσεων και των χαρακτηριστικών που θα πρέπει να έχει η τελική λύση του πρακτικού μέρους της πτυχιακής εργασίας και επιπλέον θα επιλεχθεί ο βασικός εσωτερικός serializer της βιβλιοθήκης αλλά και η γλώσσα γραφής της.

Στη συνέχεια, θα σχεδιαστεί ο τρόπος με τον οποίο θα διατηρηθούν τα references μεταξύ των objects κατά το serialization, θα παρουσιαστεί ο αναλυτικός σχεδιασμός της βιβλιοθήκης, του Two-Fold FFI, του C# wrapper και της προτεινόμενης αρχιτεκτονικής χρήσης που θα τα περικλείει στο test environment.

Τέλος, γίνεται αναφορά στον βαθμό πραγματοποίησης των στόχων της διατριβής αλλά γίνεται και η ανάδειξη των σχεδιαστικών στοιχείων του πρακτικού μέρους.

## 3.2 Ανάλυση απαιτήσεων της συνολικής λύσης

Το open-source solution που πρόκειται να παρουσιαστεί μετέπειτα έχει σαν πυρήνα του μερικές βασικές αρχές όπου θα βασιστεί ο σχεδιασμός του.

Αρχικά, βασικά θεμιτά χαρακτηριστικά της λύσης αυτής είναι, όπως προαναφέρθηκε, η open source φύση του και με βάση αυτό πρέπει να ληφθούν υπόψη μερικά σημαντικά σημεία, όπως είναι το ποσοστό το οποίο ο εκάστοτε προγραμματιστής θα μπορεί να διευρύνει τις λειτουργίες της βιβλιοθήκης για να την σμιλέψει στα δικά του project. Έτσι καταλήγουμε σε μία βιβλιοθήκη που θα πρέπει να είναι αρκετά modular, versatile αλλά και extensible χωρίς όμως να χάνει την αρχική της δομή και λειτουργία. Ο κώδικας της θα βασιστεί με μία αρχιτεκτονική Component-driven (ref) έτσι ώστε να μπορούν πολλά σημεία της να αντικατασταθούν ή να επεκταθούν.

Στη συνέχεια, ένα επιπλέον βασικό χαρακτηριστικό της βιβλιοθήκης είναι η συμβατότητα της με διαφορετικές μηχανές βιντεοπαιχνιδιών, για τις οποίες και προβλέπεται η χρήση της, χωρίς ωστόσο αυτό να την περιορίζει. Για τον λόγο αυτό, θα πρέπει η βιβλιοθήκη και τα επί μέρους κομμάτια της να είναι language-agnostic, να μην περιορίζονται δηλαδή από την εξωτερική γλώσσα που θα τα περικλείει.

Τέλος, λόγο του ίδιου χαρακτηριστικού θα πρέπει να χρησιμοποιηθούν οι κατάλληλες δομές, τεχνικές και τρόποι γραφείς, ώστε σαν σύνολο να μην έχει κάποιο μεγάλο αποτύπωμα μνήμης, κάτι που ως αποτέλεσμα θα επιφέρει και την βέλτιστη απόδοση. Φυσικά πάντα μέσα στα πλαίσια τα οποία επιτρέπουν οι δομές και οι τεχνικές που θα χρησιμοποιηθούν.

## 3.3 Ανάλυση επιλογής C++ ως βασική γλώσσα συγγραφής

Η επιλογή της χρήσης της C++ για ένα DLL εξαρτάται από διάφορους παράγοντες, όπως οι απαιτήσεις απόδοσης, η υπάρχουσα βάση κώδικα, η συμβατότητα με άλλες γλώσσες και η εξοικείωση του προγραμματιστή. Η χρήση της C++ για μια βιβλιοθήκη DLL που προορίζεται για χρήση σε μηχανές παιχνιδιών προσφέρει πολλά πλεονεκτήματα.

Αρχικά, όπως προαναφέρθηκε στο κεφάλαιο του θεωρητικού πλαισίου, η C++ φημίζεται για τις εξαιρετικές επιδόσεις της και τον low-level έλεγχο πόρων και μνήμης που παρέχει, καθιστώντας την ιδανική επιλογή για την ανάπτυξη παιχνιδιών όπου η απόδοση είναι ζωτικής σημασίας (ref). Επιπλέον, πολλές μηχανές παιχνιδιών παρέχουν ισχυρή υποστήριξη για plugins και βιβλιοθήκες C++, διευκολύνοντας την ενσωμάτωση της παρούσας βιβλιοθήκης στη διαδικασία παραγωγής τους.

Στη συνέχεια, η φορητότητα του κώδικα που είναι γραμμένος σε C++ επιτρέπει το compile του για πολλαπλές πλατφόρμες, απλοποιώντας την ανάπτυξη παιχνιδιών πολλαπλών πλατφορμών. Αυτή η ευελιξία είναι ιδιαίτερα πολύτιμη στο σημερινό τοπίο των παιχνιδιών, όπου τα παιχνίδια αναμένεται συχνά να τρέχουν σε διάφορες συσκευές και λειτουργικά συστήματα.

Τέλος, η εκτεταμένη διαθεσιμότητα βιβλιοθηκών και frameworks που σχετίζονται με παιχνίδια και είναι γραμμένα σε C++ απλοποιεί τις ανάπτυξης τους, παρέχοντας στους προγραμματιστές πληθώρα πόρων και εργαλείων και καθιστώντας την παρούσα βιβλιοθήκη συμβατή με πολλά από αυτά τα plugins (ref).

## 3.4 Επιλογή βασικού serializer

Τόσο το MessagePack όσο και οι Protocol Buffers (Protobuf) είναι μορφότυποι serialization που χρησιμοποιούνται για την αποτελεσματική κωδικοποίηση (encoding) και αποκωδικοποίηση (decoding) δομημένων δεδομένων. Ωστόσο, έχουν διαφορετικές φιλοσοφίες σχεδιασμού και εφαρμογές, που επηρεάζουν την επιλογή μεταξύ τους, όπως αυτές αναλύθηκαν στην ενότητα 2.5.

Πρώτον, το MessagePack χαρακτηρίζεται για την απλότητα και την ευκολία χρήσης του, παρέχοντας μια απλή μορφή που βοηθά στο debugging και την κατανόηση των δεδομένων, λόγο της μετάφρασης των binary δεδομένων του σε μορφή JSON. Η ανώτερη απόδοσή του, ιδίως όσον αφορά την ταχύτητα κωδικοποίησης και αποκωδικοποίησης, πλεονεκτεί για απαιτητικές εφαρμογές. Επιπλέον, η υποστήριξη του MessagePack σε διάφορες γλώσσες προγραμματισμού το καθιστά πολύ καλή επιλογή για ενσωμάτωση σε ετερογενή περιβάλλοντα, ενισχύοντας την ευελιξία του.

Ένα άλλο αξιοσημείωτο πλεονέκτημα του MessagePack είναι η runtime δυνατότητα του να κάνει serialize κάθε είδος τύπου, που του επιτρέπει να χειρίζεται ποικίλους τύπους δεδομένων χωρίς προκαθορισμένα schemes. Αυτή η ευελιξία είναι ιδιαίτερα πολύτιμη σε σενάρια με δυναμικές απαιτήσεις δεδομένων, προσφέροντας προσαρμοστικότητα χωρίς να θυσιάζεται η αποδοτικότητα.

Από την άλλη πλευρά, τα protobufs προσφέρουν διακριτά πλεονεκτήματα που μπορεί να είναι προτιμότερα σε ορισμένα πλαίσια. Με τα protobufs, η επιβολή κάποιου schema εξασφαλίζει ισχυρές εγγυήσεις δομής δεδομένων κατά το serialization και το deserialization, προωθώντας την ακεραιότητα των δεδομένων. Επιπλέον, η λειτουργία code generation παράγει κλάσεις που αφορούν τη γλώσσα με βάση το schema της, απλοποιώντας την ενσωμάτωση σε βάσεις κώδικα της C++ και παρέχοντας type-safety.

Επιπλέον, τα protobufs έχουν σχεδιαστεί με γνώμονα το backwards compatibility, επιτρέποντας το extension των ήδη υπάρχοντών schemas δεδομένων χωρίς να διαταράσσεται η συμβατότητα με τα υπάρχοντα serialized δεδομένα.

Συμπερασματικά, η επιλογή μεταξύ MessagePack και protobufs εξαρτάται από συγκεκριμένες απαιτήσεις και περιορισμούς. Το MessagePack ωστόσο υπερέχει ως προς την απλότητα, τις επιδόσεις και τη γλωσσική ανεξαρτησία, ενώ τα protobufs προτιμώνται για την επιβολή schema, τις δυνατότητες δημιουργίας κώδικα και το backwards compatibility. Ωστόσο, αξίζει να σημειωθεί ότι η δια λειτουργικότητα του MessagePack με το JSON ενισχύει περαιτέρω τη χρησιμότητά του, επιτρέποντας την ομαλή μετάφραση σε μορφή JSON και διευκολύνοντας την επικοινωνία με συστήματα που βασίζονται στο JSON για την ανταλλαγή δεδομένων.

Τέλος, είναι και open source που αυτό επιτρέπει σε κάθε προγραμματιστή να το σμιλέψει στα χαρακτηριστικά που αυτός επιθυμεί.

Με βάση αυτές τις πληροφορίες όπως αναφέρθηκαν επιγραμματικά σε αυτή την ενότητα αλλά και στην ενότητα 2.5 όπως επίσης και με γνώμονα τους βασικούς στόχους περί επιλογής serializer στη παρούσα βιβλιοθήκη, θα χρησιμοποιηθεί το C++ MsgPack (ref) για τη διαδικασία του serialization.

## 3.5 Χαρακτηριστικά λύσης

Σε αυτή την ενότητα θα αναλυθούν επιγραμματικά τα χαρακτηριστικά που θα πρέπει να έχουν η C++ βιβλιοθήκη DLL, το ενδιάμεσο FFI και ο C# wrapper.

### 3.5.1 Χαρακτηριστικά C++ DLL

Βασικά χαρακτηριστικά της C++ βιβλιοθήκης αποτελούν η διαχείριση των I/O λειτουργιών, το caching των δεδομένων προς serialization και η διαχείριση των SMRI (βλ. εν. 3.6). Επιπλέον, θα δύναται στο χρήστη πρόσβαση στα δεδομένα προς serialization, το update αυτών αλλά και η διαγραφή τους μέσω ενός εκλεπτυσμένου API. Τέλος, σε συνεργασία με το FFI σε περίπτωση κάποιου exception θα επιστρέφονται πίσω error codes, όπου αυτό είναι εφικτό.

### 3.5.2 Χαρακτηριστικά του Foreign Function Interface

Το Foreign Function Interface (FFI) θα αποτελείται από δύο μέρη, την πλευρά και τη διαχείριση των δομών, της μνήμης και άλλων, μέσα στη βιβλιοθήκη της C++ αλλά και τη διαχείριση των δεδομένων κατά τη μεταφορά τους για serialization στη πλευρά της C# στη Unity. Φυσικά, και στις δύο πλευρές θα υπάρχει μηχανισμός error handling για τους κωδικούς και τα exceptions του library όπου επεκτείνεται στον C# wrapper.

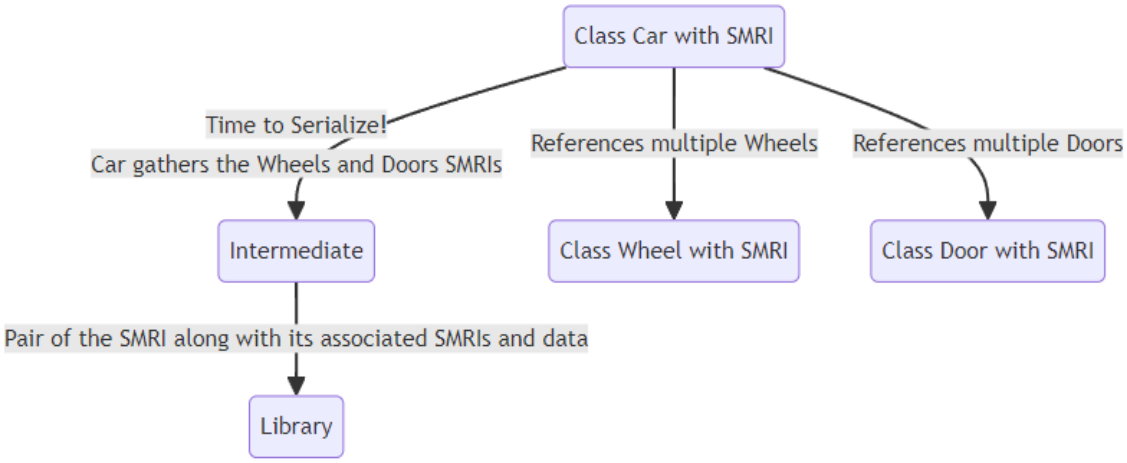
### 3.5.3 Χαρακτηριστικά του C# wrapper

Κλείνοντας την ενότητα των χαρακτηριστικών, ο C# wrapper στην πλευρά της Unity θα κάνει encapsulate τις wrapped μεθόδους μέσα από το C++ DLL, θα έχει ένα ευανάγνωστο API με στόχο την πιο εύκολη πρόσβαση από τον προγραμματιστή. Σε συνεργασία με το FFI θα παρέχει την κατάλληλη διαχείριση των error codes, των exceptions και θα αναλαμβάνει και τη μεταφορά των δεδομένων στη βιβλιοθήκη για αποθήκευση.

## 3.6 Διατήρηση των references μετά το serialization

Όπως αναλύθηκε στην ενότητα 2.4.3, δεν υπάρχει κάποιος απτός τρόπος να συντηρηθούν τα references μεταξύ των instances κατά το serialization. Για αυτό τον λόγο, ένας μηχανισμός συντήρησης αυτών θα χρειαστεί να δημιουργηθεί. Μία πάγια τακτική είναι η συσχέτιση ενός value με ένα ακέραιο αριθμό (int).

Ο λεγόμενος SMRI (Snapshot Memory Reference Index) θα είναι υπεύθυνος να μεταφέρεται μαζί με τις πληροφορίες πίσω στη βιβλιοθήκη και θα γίνεται και αυτός μαζί serialize, έτσι ώστε μετά το deserialization να μπορεί να ξαναγίνει η συσχέτιση των δεδομένων μεταξύ τους, όπως αυτό διακρίνεται στην Εικόνα 17.




*Εικόνα 10: Σχεδιάγραμμα λογικής μηχανισμού SMRI*

Ο παραπάνω μηχανισμός θα αναλυθεί εις βάθος στην ενότητα 3.8.

## 3.7 Περιβάλλον δοκιμών στη Unity

Σχετικά με τις δοκιμές, θα στηθεί ένα απλό περιβάλλον με primitive Game Objects (ref) με μερικά custom χαρακτηριστικά όπου θα γίνει χρήση του πλήρους μηχανισμού που θα σχεδιαστεί στη συνέχεια.

Τα δεδομένα θα αντικατοπτρίζουν ένα πραγματικό σενάριο παιχνιδιού ώστε να υπάρχει μία σωστή διαδικασία debugging του library αλλά και του serialization.

## 3.8 Σχεδιασμός λύσης

Παρακάτω ακολουθεί ο αναλυτικός θεωρητικός σχεδιασμός του τεχνικού μέρους της πτυχιακής εργασίας.

### 3.8.1 Σχεδιασμός βασικής βιβλιοθήκης σε C/C++

Η βιβλιοθήκη θα διαχειρίζεται το λεγόμενο current SMRI, παρέχοντας μεθόδους για την αύξηση, μείωση αλλά και επαναφορά αυτού πίσω στην προκαθορισμένη του αρχική τιμή του -1. Είναι απαραίτητη η σωστή διαχείριση αυτού, διότι μέσω του SMRI θα γίνεται και ολόκληρη η διαφοροποίηση των δεδομένων σε πακέτα.

Στη συνέχεια, η βιβλιοθήκη θα παρέχει μεθόδους για τη διαχείριση I/O και συγκεκριμένα θα έχει accessor και mutator μεθόδους για το save directory μέσω absolute path τύπου string αλλά και αντίστοιχα θα έχει mutator και accessor μεθόδους για το όνομα του παραχθέντος αρχείου από το οποίο θα κληθεί στην περίπτωση του deserialization, να διαβάσει τα serialized δεδομένα ώστε να ξανά επαναφέρει τα δεδομένα στην προηγούμενη τους μορφή.

Επιπλέον, η βιβλιοθήκη θα δέχεται τα δεδομένα σε μορφή πινάκων (arrays) τύπου char που θα αποθηκεύονται στη βιβλιοθήκη και θα περνάνε απευθείας μία διαδικασία containerization. Δηλαδή αντί να

αποθηκεύονται απευθείας μέσα σε μία δομή αποθήκευσης θα αντιγράφονται by-value μέσα σε ειδικά διαμορφωμένα struct instances και μετά αυτά τα instances θα αποθηκεύονται σε μία δομή δεδομένων. Αυτά τα instances, θα είναι προσβάσιμα μέσω του SMRI τους για διαγραφή από τη δομή, την επιστροφή των δεδομένων τους πίσω στην host γλώσσα αλλά και τη λήψη των Referenced SMRI τους. Για την αποτελεσματική μεταφορά των δεδομένων θα προτιμηθεί η μεταφορά ενός pointer στην αρχή του πίνακα (array) των δεδομένων προς αποθήκευση και όχι ένας συγκεκριμένος τύπος δεδομένων από τη host γλώσσα προς τη βιβλιοθήκη. Αυτό καθιστά γρήγορη και αποτελεσματική την διαχείριση των δεδομένων εσωτερικά της βιβλιοθήκης και επίσης δεν αναγκάζει τον εκάστοτε προγραμματιστή να δημιουργεί διπλότυπα structs και instances των δεδομένων που θέλει να αποθηκεύσει και στην host γλώσσα αλλά και μέσα στην βιβλιοθήκη π.χ. ένα C# struct τύπου “MyData” που θα έπρεπε να δημιουργηθεί και στην πλευρά της C# αλλά και στην πλευρά της C++ ώστε να γίνει το απαραίτητο data marshalling κατά τη μεταφορά.

Επιπροσθέτως, θα υπάρχουν οι απαραίτητες μέθοδοι για packing (serialization) και unpacking (deserialization) των δεδομένων από το προαναφερθείς save directory αλλά και μέθοδοι για την manual εκκαθάριση της μνήμης της βιβλιοθήκης και επαναφοράς των τιμών της.

Τέλος, ολόκληρη η διαχείριση των σφαλμάτων της βιβλιοθήκης θα γίνεται μέσω error codes που θα δίνονται μαζί με την βιβλιοθήκη.

### 3.8.2 Σχεδιασμός του Two-Fold Foreign Function Interface

Όπως αναφέρθηκε στο κεφ. 2.6, ένα FFI είναι υπεύθυνο για τη σωστή διεκπεραίωση του data marshalling και τη μετάδοση των σφαλμάτων ή των κωδικών αυτών, στην host γλώσσα, οποιαδήποτε κι αν είναι αυτή. Για να μπορέσει αυτό να λειτουργήσει αποτελεσματικά στην παρούσα βιβλιοθήκη το FFI θα αποτελείται από δύο μέρη, την πλευρά της βιβλιοθήκης από όπου θα προετοιμάζονται τα δεδομένα των μεθόδων ή οι κωδικοί σφάλματος για επιστροφή στην host γλώσσα μετά το πέρας της λειτουργίας τους και τη δεύτερη πλευρά όπου θα ενσωματωθεί μέσα στον C# wrapper ώστε αυτός με τη σειρά του να κάνει το απαραίτητο marshalling των τύπων και των δεδομένων για αποστολή ή λήψη από τη βιβλιοθήκη.

### 3.8.3 Σχεδιασμός C# Wrapper

Όπως αναλύθηκε στο κεφ. 2.7, ένας language wrapper είναι υπεύθυνος για το encapsulation των μεθόδων μίας γλώσσας για χρήση σε μία άλλη γλώσσα, που στο περιβάλλον της παρούσας πτυχιακής εργασίας η host γλώσσα είναι η C# και η guest γλώσσα είναι η C++ λόγο της φύσης του DLL. Με βάση τα χαρακτηριστικά του κεφ. 3.5.3 λοιπόν, ο static C# wrapper θα παρέχει μεθόδους όπου εμφανισιακά θα είναι πανομοιότυπες με τις μεθόδους της βιβλιοθήκης αλλά θα είναι ειδικά διαμορφωμένες για το managed περιβάλλον της C#. Συγκεκριμένα, μέσω της χρήσης των Platform Invocation Services του .NET θα γίνονται οι απαραίτητες κλήσης στις μεθόδους της βιβλιοθήκης αλλά και η διαχείριση των επιστρεφόμενων error codes για να μπορεί ο εκάστοτε προγραμματιστής να κάνει το απαραίτητο debugging σε περίπτωση κάποιου σφάλματος.

## 3.9 Σχεδιασμός test environment στη μηχανή Unity

Στο test environment θα γίνει μία προσομοίωση ενός πραγματικού σεναρίου παιχνιδιού το οποίο θα περιλαμβάνει δεδομένα τύπου double, float, string, int, Vector3 και Vector4 (Quaternion). Φυσικά, για να μπορέσει να ενσωματωθεί σωστά η βιβλιοθήκη μέσα σε ένα παιχνίδι αλλά και για να φανεί η χρήση των SMRIs, στο ακόλουθο κεφάλαιο θα αναλυθεί και σχεδιαστεί η αρχιτεκτονική χρήσης της βιβλιοθήκης σε ένα περιβάλλον αντικειμενοστραφής προγραμματισμού. Αυτό φυσικά δεν περιορίζει την χρήση της βιβλιοθήκης εκτός παιχνιδιών.

### 3.9.1 Δεδομένα και αντικείμενα

Το test environment θα προσομοιώνει ένα shooter περιβάλλον στο οποίο ο εκάστοτε παίχτης έχει στην κατοχή του ένα inventory, το οποίο inventory με τη σειρά του περιέχει έναν Ν αριθμό όπλων.

Τα δεδομένα που θα πρέπει να αποθηκευτούν είναι τα ακόλουθα:

1. SPlayer
    a. uint Smri
    b. int[] RefSmris
    c. float _Health
    d. float _Stamina
    e. float _Shield
    f. bool _IsAlive
    g. Vector3 _Position
    h. Quaternion _Rotation

2. SInventory
    a. uint Smri
    b. int[] RefSmris
    c. int _MaxItems
    d. Vector3 _Position
    e. Quaternion _Rotation

3. SWeapon
    a. uint Smri
    b. int[] RefSmris
    c. int _Ammo
    d. bool _Loaded
    e. Vector3 _Position
    f. Quaternion _Rotation

Τα παραπάνω δεδομένα είναι σχεδιασμένα να αναδείξουν τις ικανότητες της βιβλιοθήκης στον χειρισμό των δεδομένων εσωτερικά αυτής αλλά και την ορθή χρήση των SMRIs σε ένα development περιβάλλον.

### 3.9.2 Αρχιτεκτονική χρήσης SMRI

Τα Snapshot Memory Reference Indexes (SMRIs) αποτελεί το κύριο χαρακτηριστικό διαχείρισης των δεδομένων εσωτερικά και κατά συνέπεια εξωτερικά της βιβλιοθήκης. Δρουν, σαν unique identifiers για τα πακέτα δεδομένων και στην πραγματικότητα η βιβλιοθήκη θα είναι υπεύθυνη για την σωστή καταχώριση των πακέτων αυτών μέσα στις δομές αποθήκευσης της. Για να επιτευχθεί αυτό, σε ένα αντικειμενοστραφές περιβάλλον, θα σχεδιαστεί και θα προταθεί μία «αρχιτεκτονική χρήσης» ώστε να μπορέσει η βιβλιοθήκη να αξιοποιηθεί στο μέγιστο της.

Συγκεκριμένα, στο loading δεδομένων στα βίντεο παιχνίδια υπάρχουν δύο βασικά σενάρια loading. Το σενάριο στο οποίο ο κόσμος αποθηκεύεται ολόκληρος με τη χρήση κάποιων metadata και μετέπειτα φορτώνεται πίσω στο ίδιο στάδιο και μορφή που ήταν κατά τη διάρκεια του save και το σενάριο στο οποίο ο κόσμος φορτώνεται σε ένα default state – σαν δηλαδή να φορτώνεται η σκηνή και τα δεδομένα όπως θα φορτωνόταν σε ένα καινούργιο save – και μετέπειτα ένας μηχανισμός περνάει όλα τα αποθηκευμένα δεδομένα από το αρχείο μέσα στο παιχνίδι σε μία σειριακή αλληλουχία με συγκεκριμένη σειρά. Η προτεινόμενη αρχιτεκτονική θα βασιστεί στο δεύτερο σενάριο.
Οι κλάσης και τα αντικείμενα που ακολουθούν θα ανήκουν στο namespace Proposed Architecture:

1. Common
    a. Θα αποτελεί το μόνο Singleton του παιχνιδιού και θα είναι υπεύθυνη για το instance caching των World Loader και Save Manager.

2. Global Properties
    a. Θα περιέχει constants όπως το save folder path και το save folder name για χρήση από οποιοδήποτε σημείο του παιχνιδιού.

3. World Loader
    a. Θα είναι υπεύθυνη για το σωστό φόρτωμα των αντικειμένων του Player, Inventory και Weapon ανάλογα με το load state του παιχνιδιού, εάν αυτό θα είναι “Fresh Save” ή “From Load”. Στην περίπτωση που το παιχνίδι ξεκινάει σε “From Load” state, ο World Loader θα είναι υπεύθυνος να καλέσει και την ανάλογη μέθοδο από τον -ακόλουθο- Save Manager μέσω του Common singleton.

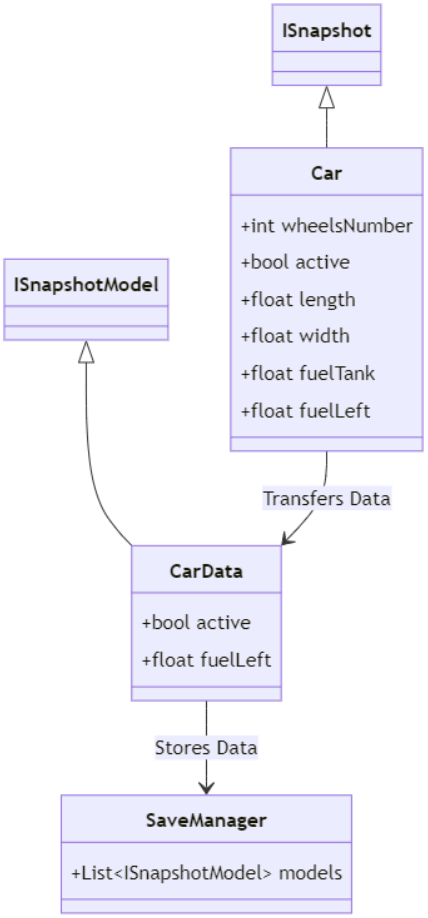
4. Save Manager
    a. Η κλάση Save Manager θα είναι υπεύθυνη να κάνει encapsulate τον C# wrapper ο οποίος περιέχει μέσα τις static μεθόδους που με την σειρά τους καλούν τις μεθόδους από την βιβλιοθήκη. Συγκεκριμένα, θα περιέχει μεθόδους για το registering των ISnapshots -αναλύονται αμέσως μετά- και των ISnapshot Models -αναλύονται αμέσως μετά- όπως και για το unregistering τους. Επιπλέον, θα περιέχει μεθόδους για το “Saving” και το “Loading” των δεδομένων μέσα από την βιβλιοθήκη. Τέλος, θα είναι υπεύθυνη για τον καθαρισμό της βιβλιοθήκης κατά την έξοδο του παιχνιδιού.

5. ISnapshot interface
    a. Το ISnapshot interface θα είναι το πρώτο πιο σημαντικό component της αρχιτεκτονικής αφού θα είναι υπεύθυνο να ξεχωρίζει τις κλάσεις που προορίζονται για saving από τον Snapshot μηχανισμό. Θα περιέχει όλες τις απαραίτητες μεθόδους που

θα πρέπει να υλοποιήσει μία κλάση ώστε να αποστείλει τα δεδομένα της στον κεντρικό Save Manager την στιγμή του saving αλλά και στην περίπτωση του loading να δεχθεί το αποθηκευμένο πακέτο δεδομένων ώστε να ξανά κάνει mutate τα serialized class fields του που είχαν αποθηκευτεί. Στη συνέχεια, θα περιέχει μεθόδους για την ανάκτηση των referenced SMRIs του ώστε μέσω του Save Manager να ανακτήσει τα references στα objects που είχε κατά τη διάρκεια του save. Τέλος, θα περιέχει και ένα field για το SMRI της ίδιας της κλάσης.

6. ISnapshot Model interface
    a. Το ISnapshot Model interface θα είναι το δεύτερο πιο σημαντικό component της αρχιτεκτονικής αφού θα είναι υπεύθυνο να ξεχωρίζει τα λεγόμενα "μοντέλα των ISnapshot" που θα περιέχουν τα δεδομένα προς αποθήκευση στη βιβλιοθήκη. Θα περιέχει δύο fields, ένα για το SMRI της κλάσης που αντιπροσωπεύει και ένα δεύτερο για τα Referenced SMRIs που θα έχει η κλάση αυτή. Ουσιαστικά, θα γίνεται inherit από απλά data carriage structs.

Η σχέση των Save Manager, ISnapshot και ISnapshotModel μπορεί να διακριθεί στην Εικόνα 17 και στην Εικόνα 18.




*Εικόνα 11: Αρχιτεκτονική σχέση μεταξύ Save Manager, ISnapshot και IsnapshotModel*

## 3.10 Σύγχρονη ή Ασύγχρονη προσέγγιση;

Όλες οι μέθοδοι της βιβλιοθήκης θα λειτουργούν στο main thread της εφαρμογής, εκτός αν η host γλώσσα κάνει χρήση των μεθόδων της σε ένα καινούργιο thread, όπου σε εκείνη την περίπτωση οι λειτουργίες της βιβλιοθήκης θα λειτουργούν ασύγχρονα σε νέο thread αλλά θα τρέχουν σύγχρονα με βάση την σειρά που καλούνται στο εκάστοτε thread. Αυτή η απόφαση πάρθηκε έτσι ώστε να μην υπάρχουν περιορισμοί στην χρήση της βιβλιοθήκης από κάποια άλλη γλώσσα και η απόφαση αυτή θα βρίσκεται στην κρίση του εκάστοτε developer που κάνει χρήση της βιβλιοθήκης.

## 3.11 Ανάδειξη σχεδιαστικών στοιχείων

Παρόλο που η βιβλιοθήκη και τα επί μέρους τμήματα της πτυχιακής αυτής έχουν σχεδιαστεί με βασικές πρακτικές του Object Oriented Design, το ζήτημα του serialization των object references παρέμεινε. Έτσι, τα SMRIs και το containerization των δεδομένων εσωτερικά και εξωτερικά της βιβλιοθήκης σχεδιάστηκαν για να μπορέσουν τα δεδομένα προς αποθήκευση να παραμείνουν απαράλλακτα κατά τη μεταφορά τους. Επιπλέον, όπως έχει προαναφερθεί στα προηγούμενα κεφάλαια, τα SMRIs δρουν σαν απλοί αριθμοί που συσχετίζουν ένα πακέτο δεδομένων με την κλάση από την οποία ήρθαν και έτσι αποθηκεύοντας το SMRI μίας κλάσης αλλά μαζί και τα SMRI των referenced objects που έχει αυτή η κλάση, μπορεί μετά ένας μηχανισμός εύκολα και με τη χρήση μίας κεντρικής δομής να αναθέσει ξανά τα ίδια references στις κλάσεις που ήταν κατά τη διάρκεια της αποθήκευσης.

Στη συνέχεια, η συγκεκριμένη βιβλιοθήκη προορίζεται να δημοσιευθεί σαν open source library, οπότε ο component-driven σχεδιασμός της, θα επιτρέπει στον εκάστοτε προγραμματιστή να αλλάξει εύκολα μέσω του source κώδικα της διαφορετικά μέρη της βιβλιοθήκης, όπως ο εσωτερικός serializer, πράγματα σχετικά με την ονοματοδοσία του τελικού αρχείου ή και ακόμα να την επεκτείνει εύκολα μέσω μίας Utilities κλάσης που θα περιέχει βοηθητικές μεθόδους για χρήση εσωτερικά της βιβλιοθήκης.

Τέλος, δύο επίσης σημαντικά χαρακτηριστικά, είναι πως οι μέθοδοι της θα δέχονται μόνο primitive type μεταβλητές και pointers κάτι που την καθιστά language agnostic και δεύτερων πως οι δομές που θα χρησιμοποιηθούν θα βασίζονται επάνω στα BigO metrics τους ώστε να μην υπάρχει μεγάλο performance overhead.

# Κεφάλαιο 4: Υλοποίηση

## 4.1 Εισαγωγή κεφαλαίου

Στο κεφάλαιο αυτό, γίνεται η πλήρης και αναλυτική επεξήγηση της υλοποίησης του πρακτικού μέρους της πτυχιακής εργασίας. Στο τέλος του κεφαλαίου γίνεται αναφορά στα προβλήματα που προέκυψαν κατά την συγγραφή του πρακτικού μέρους.

## 4.2 Υλοποίηση API και C++ DLL

Αρχικά, οι βασικές ρυθμίσεις της βιβλιοθήκης είναι το configuration type της στη ρύθμιση “Dynamic Library (.dll)”, Windows SDK Version σε “10.0”, Platform Toolset σε “Visual Studio 2022”, C++ Language Standard σε “ISO C++ 17 Standard (/std:c++17)” και τέλος η ρύθμιση C Language Standard σε “ISO C17 (2018) Standard (std:c17)”. Οι ρυθμίσεις αυτές διακρίνονται στην Εικόνα 19.

| | |
|---|---|
| Configuration Type | **Dynamic Library (.dll)** |
| Windows SDK Version | **10.0 (latest installed version)** |
| Platform Toolset | **Visual Studio 2022 (v143)** |
| C++ Language Standard | **ISO C++17 Standard (/std:c++17)** |
| C Language Standard | **ISO C17 (2018) Standard (/std:c17)** |

*Εικόνα 12: Βιβλιοθήκη - Visual Studio base configurations*

Στη συνέχεια, οι επί μέρους ρυθμίσεις όπως το Character Set και το CLR Support είναι ρυθμισμένα στα “Use Unicode Character Set” και “.NET Framework Runtime Support (/clr)”, αντίστοιχα. Ο λόγος που έχει ενεργοποιηθεί το Common Language Runtime Support είναι για να είναι συμβατό με το Unity project. Είναι μία ρύθμιση που κατά το compilation ένας χρήστης μπορεί να την απενεργοποιήσει. Οι ρυθμίσεις αυτές μπορούν να διακριθούν στην Εικόνα 20.

| | |
|---|---|
| Character Set | **Use Unicode Character Set** |
| Whole Program Optimization | **<different options>** |
| MSVC Toolset Version | Default |
| Enable MSVC Structured Output | Yes |
| **C++/CLI Properties** | |
| Common Language Runtime Support | **.NET Framework Runtime Support (/clr)** |

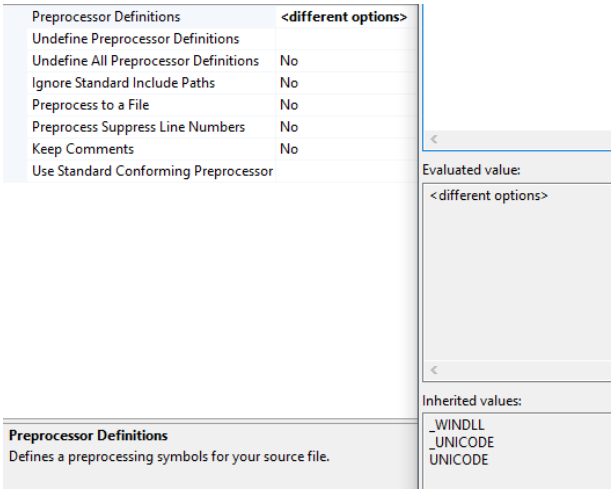
*Εικόνα 13: Βιβλιοθήκη - Visual Studio advanced configurations*

Επιπλέον, έχει ενεργοποιηθεί το precompiled header σε με ονομασία “pch.h”, όπως αυτό φαίνεται στην Εικόνα 21.

| | |
|---|---|
| Precompiled Header | **Use (/Yu)** |
| Precompiled Header File | **pch.h** |
| Precompiled Header Output File | $(IntDir)$(TargetName).pch |

*Εικόνα 14: Βιβλιοθήκη - Visual Studio pch*

Τέλος, η ρύθμιση Preprocessor Definitions έχει τα values “_WINDLL”, “_UNICODE” και “UNICODE”, όπως φαίνεται στην Εικόνα 22. Οι ρυθμίσεις του compiler και του linker παρέμειναν στα default values τους από το Visual Studio.




*Εικόνα 15: Βιβλιοθήκη - Visual Studio preprocessor definitions*

Η βιβλιοθήκη δημιουργήθηκε και χωρίστηκε σε 9 συγκεκριμένα τμήματα με αυτά να είναι τα:

1. Library Specific Structs
    a. Τα structs Data και DataContainer χρησιμοποιούνται για το serialization των δεδομένων από το MsgPack και το containerization των δεδομένων αντίστοιχα.

2. GlobalVariables
    a. Περιέχει σημαντικές μεταβλητές για την χρήση τους εσωτερικά της βιβλιοθήκης αλλά και την βασική δομή αποθήκευσης τύπου std::map<unsigned int, DataContainer> όπου είναι υπεύθυνη για το caching των πακέτων δεδομένων.

3. SavePath
    a. Περιέχει τις mutator και accessor μεθόδους για τη σωστή διαχείριση του directory absolute save path.

4. SMRI Handling
    a. Περιέχει όλες τις μεθόδους για τη διαχείριση των SMRI εξωτερικά της βιβλιοθήκης αλλά και τη δυνατότητα διαγραφής ενός πακέτου δεδομένων με την εισαγωγή του SMRI του σαν παράμετρο.

5. DataCaching_Packing
    a. Περιέχει όλες τις μεθόδους για τη διαχείριση των αποθηκευμένων δεδομένων αλλά και την μέθοδο packData που ξεκινάει τη διαδικασία serialization των δεδομένων της βιβλιοθήκης.

6. LoadFromFile_Unpacking
    a. Περιέχει όλες τις μεθόδους για access και mutate του αρχείου που θα πρέπει να φορτώσει η βιβλιοθήκη ώστε να το κάνει deserialize και την βασική μέθοδο deserialization ονόματη unpackData.

7. DLL_Cleanup
    a. Περιέχει τις μεθόδους εκκαθάρισης της μνήμης του DLL.

8. LibraryUtils file
    a. Περιέχει utility μεθόδους που χρησιμοποιούνται για string formatting και path validation εσωτερικά της βιβλιοθήκης.

9. SnapshotReturnCodes enumeration
    a. Το enumeration αυτό χρησιμοποιείται μόνο για τη διαχείριση και τη μεταφορά των error codes στην host language, διότι όλες οι λειτουργίες της βιβλιοθήκης συμβαίνουν μέσα σε try-catch blocks.

### 4.2.1 Library specific structs

Το struct DataContainer έχει ως μοναδικό σκοπό το caching των δεδομένων του εισαγόμενου SMRI και των Referenced SMRI του. Περιέχει επίσης και τη template μέθοδο του MsgPack serializer ώστε να μπορεί το header file να κάνει το απαραίτητο serialization. Είναι άξιο να σημειωθεί, πως τα values των εισαγόμενων δεδομένων αποθηκεύονται σαν unsigned chars και όχι σαν unsigned char* (pointer) και αυτό συμβαίνει διότι όπως είναι φυσικό δεν μπορεί μία θέση μνήμης να γίνει serialize αλλά πρέπει να γίνουν serialized τα values αυτών των θέσεων μνήμης.

**Καταχώρηση 8:** **SnapshotLib.DataContainer struct**

```
/*
A struct for incoming data handling.
Used solemly for data caching, packing and unpacking.
*/
struct DataContainer
{
	/*The saved data cache parent SMRI*/
	unsigned int _Smri;
	/*The cached data array size*/
	int _DataSize;
	/*The cached data converted in unsigned char for serialization*/
	std::vector<unsigned char> _DataValues;
	/*The referenced SMRI values this SMRI has.*/
	std::vector<int> _RefSmris;

	/*Packing msgpack::cppack method*/
	template<class T>
	void pack(T& pack) {
		pack(_Smri, _DataSize, _DataValues, _RefSmris);
	}
};

```

Το struct Data δημιουργήθηκε μόνο για τη διευκόλυνση της διαδικασίας του packing και του unpacking από το MsgPack.

**Καταχώρηση 9:** **SnapshotLib.Data struct**

```
/*
A struct used solemly for packing and unpacking purposes.
Helps in making DataContainer serialization and deserialization easier.
*/
struct Data
{
	/*The map containing the smri-data pairs meant for serialization and deserialization.*/
	std::map<unsigned int, DataContainer> _ModelsCache;

	/*Packing msgpack::cppack method*/
	template<class T>
	void pack(T& pack) {
		pack(_ModelsCache);
	}
};

```

### 4.2.2 Region Global Variables

Στο region αυτό διακρίνεται η βασική δομή αποθήκευσης τύπου std::map<unsigned int, DataContainer> _ModelsCache στο οποίο το key είναι το εισαγόμενο SMRI και το value είναι το δημιουργημένο instance τύπου DataContainer όπου περιέχει τα αντιγραμμένα δεδομένα προς αποθήκευση. Επιλέχθηκε η δομή std::map για την εύκολη χρήση της ως key-value storage η οποία στην πραγματικότητα είναι ένα Red-Black Tree και στα Big O metrics της έχει average αλλά και worst case scenarios, time complexity θ(log(n)) και O(log(n)) αντίστοιχα (ref). Το space complexity του φυσικά παραμένει O(n).

Τέλος, οι constant μεταβλητές SAVE_FORMAT και SAVE_EXTENSION μπορούν να αλλαχθούν χωρίς κάποια επίπτωση στις υπόλοιπες λειτουργίες της βιβλιοθήκης με τη χρήση της LibraryUtils βιβλιοθήκης εάν ο προγραμματιστής επιθυμεί κάποιο άλλο παραγόμενο όνομα.

**Καταχώρηση 10:** **SnapshotLib – Global Variables**

```
#pragma region GlobalVariables
/*The save format of the saved files.*/
const std::string SAVE_FORMAT = "{date}_{cnt}";
/*The save file extension*/
const std::string SAVE_EXTENSION = ".sav";
/*
The global SMRI value used in data storage and reference preservation.
Default value is -1
*/
int _GlobalSmriValue = -1;
/*A map storing SMRI-Data pairs. Used in serialization and deserialization.*/
std::map<unsigned int, DataContainer> _ModelsCache;
/*The externally-set absolute save path.*/
std::string _SavePath = "";
/*The externally-set file name to unpack from.*/
std::string _LoadFile = "";
#pragma endregion

```

### 4.2.3 Region Save Path

Αρχικά, η μέθοδος “setSavePath” δέχεται ένα const char* (array) που εκπροσωπεί ένα string με το absolute save path. Ο φάκελος αποθήκευσης δημιουργείται τη στιγμή που καλείται αυτή η μέθοδος στην περίπτωση που δεν προϋπήρχε. Στην περίπτωση αποτυχίας οποιασδήποτε λειτουργίας, το ανάλογο error code επιστρέφεται σε μορφή τύπου short με βάση το προκαθορισμένο κωδικό από το enumeration SnapshotReturnCodes (βλ. κεφ. 4.2.9). Στην περίπτωση επιτυχίας, επιστρέφεται ο σταθερός κωδικός «0».

**Καταχώρηση 11:** **SnapshotLib.setSavePath**

```
#pragma region SavePath
/*
Sets the library save path variable.
The final directory gets created if it does not already exist.

@param const char* _savePath:
      A string containing the absolute save path to a directory.
@return         OperationSuccessful: Successful set of the path
@return         DirectoryNotFound: If the directory is not found
@return         OperationFailed: In case of any other error
*/
short setSavePath(const char* _savePath) {
      try {
                handleSaveDirectory(_savePath);

                _SavePath = _savePath;
                return OperationSuccessful;
      }
      catch (std::runtime_error) {
                return DirectoryNotFound;
      }
      catch (...) {
                return OperationFailed;
      }
}
```

Τέλος, η μέθοδος “getSavePath” επιστρέφει την αποθηκευμένη μεταβλητή _SavePath σαν C string και σε περίπτωση αποτυχίας κάποιας λειτουργίας επιστρέφεται ένας null pointer.

**Καταχώρηση 12:** **SnapshotLib.getSavePath**

```
/*
Returns the stored save directory absolute path.
@return The stored save path as a C string.
@return Nullptr in any other case
*/
const char* getSavePath() {
      try {
                return _SavePath.c_str();
      }
      catch (...) {
                return nullptr;
      }
}
```

### 4.2.4 Region SMRI Handling

Αρχικά, οι μέθοδοι “getSmri” και “decreaseSmri” αποτελούν τις βασικές μεθόδους χειρισμού των κωδικών SMRI των πακέτων, με την μία να επιστρέφει τον αυξανόμενο κωδικό και με την δεύτερη να μειώνεται κατά 1 σε περίπτωση κάποιου λάθους. Τα SMRIs είναι μόνο θετικοί αριθμοί, εξού και η χρήση unsigned int σαν τύπο επιστροφής. Η μέθοδος “getCurrentSmri” επιστρέφει τον κωδικό SMRI χωρίς να τον αυξάνει.

**Καταχώρηση 13:** **SnapshotLib.getSmri - decreaseSmri**

```
/*
Increases and returns the current global SMRI.
In case of any error, call decreaseSmri().
@return The current SMRI value incremented by 1.
*/
unsigned int getSmri() {
	_GlobalSmriValue += 1;
	return _GlobalSmriValue;
}

/*
Decreases the Global SMRI by 1.
Caps to default value: -1
*/
void decreaseSmri() {
	_GlobalSmriValue -= 1;
	if (_GlobalSmriValue < -1) {
		_GlobalSmriValue = -1;
	}
}
```

**Καταχώρηση 14:** **SnapshotLib.getCurrentSmri**

```
/*
@return The current global SMRI without incrementing it.
*/
unsigned int getCurrentSmri() {
	return _GlobalSmriValue;
}

```

Τέλος, η μέθοδος “deleteSmriData” είναι υπεύθυνη για τη διαγραφή του πακέτου δεδομένων που σχετίζονται με την παράμετρο που περάστηκε στην μέθοδο. Στην περίπτωση αποτυχίας οποιασδήποτε λειτουργίας, το ανάλογο error code επιστρέφεται σε μορφή τύπου short με βάση το προκαθορισμένο κωδικό από το enumeration SnapshotReturnCodes (βλ. κεφ. 4.2.9). Στην περίπτωση επιτυχίας, επιστρέφεται ο σταθερός κωδικός «0».

**Καταχώρηση 15:** **SnapshotLib.deleteSmriData**

```
/*
Deletes the associated SMRI Data Container from the ModelCache.
@param unsigned int _smri
	The SMRI to delete
@return OperationSuccessful: Deletion was successful
@return OperationFailed: Any other error.
*/
short deleteSmriData(unsigned int _smri) {
	try {
		_ModelsCache.erase(_smri);
```

```
11.                 return OperationSuccessful;
12.         }
13.         catch (...) {
14.                 return OperationFailed;
15.         }
16. }
17.
```

### 4.2.5 Region Data Caching and Packing

Αρχικά, η μέθοδος “cacheData” αποτελεί μία από τις σημαντικότερες και βασικές μεθόδους της βιβλιοθήκης. Δέχεται και μετατρέπει τα δεδομένα by-value από byte arrays σε ένα vector από unsigned char, ώστε να μπορούν τα δεδομένα να γίνουν serialized μέσω του MsgPack, διότι δεν γίνεται να αποθηκευτεί ένας pointer στη θέση μνήμης. Το ίδιο συμβαίνει και για τα referenced SMRIs. Όλα τα δεδομένα αντιγράφονται μέσα σε ένα instance τύπου DataContainer και αποθηκεύονται στην κεντρική δομή _ModelsCache. Η λειτουργία αυτή διακρίνεται στην Εικόνα 31. Είναι σημαντικό να σημειωθεί, πως η συγκεκριμένη μέθοδος δέχεται και τα sizes των δύο arrays ώστε να μπορεί να γίνει «ασφαλής» η αντιγραφή των δεδομένων, διότι η C++ είναι μία “unsafe” γλώσσα και σε περίπτωση που περαστεί λάθος size, το for iteration θα συνεχίσει εκτός του array και θα αρχίσει να αποθηκεύει άλλα δεδομένα από τη stack που έχει γίνει allocated στην τρέχουσα εφαρμογή.

**Καταχώρηση 16:** **SnapshotLib.cacheData**

```
 1. /*
 2. That's the main data caching function of the library.
 3. Creates and caches the DataContainer with the passed parameters and saves it in the ModelCache.
 4. Passed _data are converted into unsigned char values.
 5. @param unsigned int _smri: The smri to associate with the data.
 6. @param int _dataSize: The size of the marshalled data.
 7. @param unsigned char* _data: Pointer to the marshalled data array (byte array)
 8. @param int _refsSize: The int ref smri array size
 9. @param int* _refSmris: Pointer to the marshalled int array containing the ref SMRIs of this
SMRI.
10. @return OperationSuccessful: If the caching and convertion were successful
11. @return OperationFailed: If any error occurs.
12. */
13. short cacheData(unsigned int _smri, int _dataSize, unsigned char* _data, int _refsSize, int*
_refSmris) {
14.         try {
15.                 DataContainer data = DataContainer();
16.                 data._Smri = _smri;
17.                 data._DataSize = _dataSize;
18.                 //Data copying
19.                 for (int i = 0; i < data._DataSize; ++i) {
20.                         data._DataValues.push_back(_data[i]);
21.                 }
22.
23.                 for (int i = 0; i < _refsSize; i++) {
24.                         data._RefSmris.push_back(_refSmris[i]);
25.                 }
26.
27.                 _ModelsCache[data._Smri] = data;
28.
29.                 return OperationSuccessful;
30.         }
31.         catch (...) {
32.                 return OperationFailed;
```

```
33.       }
34. }
35.
```

Στη συνέχεια, η μέθοδος “getData” επιστρέφει τα δεδομένα του ζητούμενου SMRI σε τύπο unsigned char* (array) αλλά και το size των δεδομένων ώστε η host γλώσσα να μπορεί να κάνει την απαραίτητη αντιγραφή και το απαραίτητο data marshalling. Στην περίπτωση αποτυχίας, επιστρέφεται ένας null pointer.

**Καταχώρηση 17:** **SnapshotLib.getData**

```
 1. /*
 2. Returns a pointer to the passed _smri associated data array.
 3. @param unsigned int _smri: The SMRI to retrieve data from
 4. @param int* _size: Is an output parameter and will be set with the array size of the returned
data.
 5. @return A pointer to the beginning of the SMRI associated data.
 6. @return A nullptr in any other case.
 7. */
 8. unsigned char* getData(unsigned int _smri, int* _size) {
 9.       try {
10.                 *_size = _ModelsCache.at(_smri)._DataSize;
11.                 return _ModelsCache.at(_smri)._DataValues.data();
12.       }
13.       catch (...) {
14.                 return nullptr;
15.       }
16. }
17.
```

Η μέθοδος “getRefSmris” επιστρέφει αντίστοιχα τα referenced SMRIs του ζητούμενου SMRI κωδικού αλλά και το size του array τους.

**Καταχώρηση 18:** **SnapshotLib.getRefSmris**

```
 1. /*
 2. Returns a pointer to the referenced SMRI values of the passed _parentSmri;
 3. @param unsigned int _parentSmri: The SMRI to retrieve the referenced SMRIs array from.
 4. @param  int* _size: Is an output parameter and will be set with the array size of the returned
data.
 5. @return int* A pointer to the beginning of the Reference SMRI int array.
 6. @return A nullptr in any other case.
 7. */
 8. int* getRefSmris(unsigned int _parentSmri, int* _size) {
 9.       try {
10.                 *_size = _ModelsCache.at(_parentSmri)._RefSmris.size();
11.                 return _ModelsCache.at(_parentSmri)._RefSmris.data();
12.       }
13.       catch (...) {
14.                 return nullptr;
15.       }
16. }
17.
```

Τέλος, η μέθοδος “packData” αποτελεί την μοναδική μέθοδο serialization των cached δεδομένων της βιβλιοθήκης. Για την δημιουργία του αρχείου με τα binary serialized δεδομένα, δημιουργείται δυναμικά ένα string με βάση τη global constant μεταβλητή SAVE_FORMAT όπου αντικαθίσταται το κομμάτι {date}

με την ημερομηνία της αποθήκευσης σε format “DD_MM_YYY” και το μέρος {cnt} με τον αύξον αριθμό των ήδη υπάρχων αρχείων μέσα στο save directory. Για λόγους ευκολίας του serialization, ολόκληρη η δομή _ModelsCache αποθήκευεται μέσα σε ένα instance τύπου Data.

**Καταχώρηση 19:** **SnapshotLib.packData**

```
 1. /*
 2. The main serialization method of the library.
 3. The whole _ModelCache gets serialized and stored in the set save path along with a dynamically
created name controlled from the SAVE_FORMAT variable.
 4. The serialization is handled from the msgpack hpp library.
 5. @return OperationSuccessful: If the serialization was successful
 6. @return OperationFailed: If any error occurs.
 7. */
 8. short packData() {
 9.       try {
10.                 Data container = Data();
11.                 container._ModelsCache = std::map<unsigned int, DataContainer>(_ModelsCache);
12.                 std::vector<uint8_t> serData = msgpack::nvp_pack(container); //Serialization
13.
14.                 int cnt = getFileCount(_SavePath);
15.                 std::string dt = getCurrentDate();
16.                 std::string finalSaveName = formatSaveString(SAVE_FORMAT, dt, cnt);
17.
18.                 std::string saveStr = combinePath(_SavePath, finalSaveName) + SAVE_EXTENSION;
19.                 std::ofstream outfile(saveStr, std::ios::out | std::ios::binary);
20.                 outfile.write(reinterpret_cast<const char*>(serData.data()), serData.size());
21.                 outfile.close();
22.
23.                 return OperationSuccessful;
24.       }
25.       catch (...) {
26.                 return OperationFailed;
27.       }
28. }
29.
```

### 4.2.6 Region Load from File and Unpacking

Αρχικά, όπως η μέθοδος “packData”, η αντίστοιχη μέθοδος “unpackData” είναι η μοναδική μέθοδος deserialization των αποθηκευμένων δεδομένων. Είναι υπεύθυνη για την ανάγνωση των binary serialized δεδομένων από το save directory και μέσα από το pre-set save file name. Όλα τα δεδομένα μετατρέπονται πίσω σε μία δομή τύπου std::map<unsigned int, DataContainer> και είναι προσβάσιμες με τις προαναφερθέντες μεθόδους. Είναι άξιο αναφοράς, πως ο κωδικός SMRI γίνεται ίσως με το _ModelsCache size – 1 ώστε να μπορούν να προστεθούν επιπλέον δεδομένα στη δομή χωρίς να διαγραφεί κάποιο deserialized δεδομένο. Ολόκληρη η λειτουργία διακρίνεται στην Εικόνα 35.

**Καταχώρηση 20:** **SnapshotLib.unpackData**

```
 1. /*
 2. The deserialization function of the library.
 3. The serialized data are read directly from the save path and file name, get deserialized and
 4. cached inside the _ModelsCache library map.
 5. The GlobalSMRI is set to be equal to the size of the deserialized data map.
 6. @return CouldNotOpenFile: In case the file could not be opened.
 7. @return ReadNotSuccessful: In case the file could not be read.
```

```
 8. @return OperationSuccessful: In case the whole process was successful.
 9. @return OperationFailed: In any other error.
10. */
11. short unpackData() {
12.       try {
13.
14.                 std::ifstream dataFile(combinePath(_SavePath, _LoadFile), std::ios::binary);
15.                 if (!dataFile.is_open()) {
16.                           //Could not open file
17.                           return CouldNotOpenFile;
18.                 }
19.
20.                 //Sizing
21.                 dataFile.seekg(0, std::ios::end);
22.                 std::streampos fileSize = dataFile.tellg();
23.                 dataFile.seekg(0, std::ios::beg);
24.
25.                 //File read
26.                 std::vector<uint8_t> bytes(fileSize);
27.                 dataFile.read(reinterpret_cast<char*>(bytes.data()), fileSize);
28.
29.                 if (!dataFile) {
30.                           //Could not read file correctly
31.                           return ReadNotSuccessful;
32.                 }
33.
34.                 Data container = msgpack::nvp_unpack<Data>(bytes); //Deserialization
35.                 for (const std::pair<unsigned int, DataContainer>& pair :
container._ModelsCache) {
36.                           _ModelsCache[pair.first] = pair.second;
37.                 }
38.
39.                 //Size must be set here in case we unpack a cache so the rest SMRIs register
correctly.
40.                 int sz = _ModelsCache.size() - 1;
41.                 _GlobalSmriValue = sz <= 0 ? resetSmri() : sz;
42.
43.                 return OperationSuccessful;
44.       }
45.       catch (...) {
46.                 return OperationFailed;
47.       }
48. }
```

Τέλος, οι μέθοδοι “setLoadFileName” και “getLoadFileName” είναι υπεύθυνες για το accessing και το mutating της global μεταβλητής _LoadFile η οποία χρησιμοιπιείται για το deserialization. Στην περίπτωση που το αρχείο δεν υπάρχει, η μέθοδος “setLoadFileName” επιστρέφει το snapshot error code FileNotFound.

**Καταχώρηση 21:** **SnapshotLib.setLoadFileName**

```
 1. /*
 2. Sets the file name to read data to unpack from which must reside inside the set _SavePath.
 3. The name gets validated for its existance inside the _SavePath every time it is set.
 4. @param const char* _loadFileName: A string containing the file name to load from.
 5. @throw std::runtime_error: Passed _LoadFile does not exist in the saves folder.
 6. @return OperationSuccessful: If the name was successfully set
 7. @return FileNotFound: If the file was not found inside the _SavePath directory.
 8. @return OperationFailed: If any other error occurs.
 9. */
10. short setLoadFileName(const char* _loadFileName) {
```

```
	try {
		std::string comp = combinePath(_SavePath, _loadFileName);
		if (!fileExists(comp)) {
			throw std::runtime_error("Passed _LoadFile does not exist in the saves folder.");
		}

		_LoadFile = _loadFileName;
		return OperationSuccessful;
	}
	catch (std::runtime_error) {
		return FileNotFound;
	}
	catch (...) {
		return OperationFailed;
	}
}

/*
Returns the library cached _LoadFile value containing the file name to unpack data from.
@return A C string with the value of the _LoadFile
@return A nullptr in any other case.
*/
const char* getLoadFileName() {
	try {
		return _LoadFile.c_str();
	}
	catch (...) {
		return nullptr;
	}
}

```

### 4.2.7 Region DLL Cleanup

Η βιβλιοθήκη παρέχει και δύο μεθόδους για τον καθαρισμό της μνήμης της διότι η C++ δεν διαθέτει κάποιον αυτοματισμό τύπου garbage collection. Η μέθοδος “resetSmri” είναι υπεύθυνη για την επαναφόρα του global SMRI στη default τιμή του -1 και η μέθοδος “resetCache” είναι υπεύθυνη για την διαγράφη των αποθηκευμένων strings στις μεταβλητές _SavePath και _LoadFile αλλά και για την εκκαθάριση της δομής αποθήκευσης της βιβλιοθήκης.

**Καταχώρηση 22:** **SnapshotLib.resetSmri - resetCache**

```
/*
Resets the global SMRI back to its default value: -1
@return OperationSuccessful if the operation was successful.
@return OperationFailed: In any other error.
*/
short resetSmri() {
	try {
		_GlobalSmriValue = -1;
		return OperationSuccessful;
	}
	catch (...) {
		return OperationFailed;
	}
}

/*
```

```
17. Clears the ModelsCache, _SavePath and _LoadFile memory.
18. @return OperationSuccessful If the operation was successful.
19. @return OperationFailed: In any other error.
20. */
21. short resetCache() {
22.       try {
23.                 _ModelsCache.clear();
24.                 _SavePath = "";
25.                 _LoadFile = "";
26.                 return OperationSuccessful;
27.       }
28.       catch (...) {
29.                 return OperationFailed;
30.       }
31. }
32.
```

### 4.2.8 Library Utilities αρχεία

Η βιβλιοθήκη περιέχει κι ένα αρχείο με βοηθητικές μεθόδους για εσωτερική χρήση που κυμαίνονται από την λήψη του αριθμού των αρχείων μέσα σε ένα directory έως και το formatting ενός string και το validation ύπαρξης των εισαγόμενων file paths και file names. Στην Εικόνα 38 διακρίνεται το header file και στις Εικόνες 39 και 40 οι υλοποιήσεις των μεθόδων.

**Καταχώρηση 23:** **LibraryUtils.h**

```
 1. #pragma once
 2.
 3. #include "pch.h"
 4.
 5. namespace fs = std::filesystem;
 6.
 7. /*Method declaration*/
 8. int getFileCount(std::string _path);
 9. /*Method declaration*/
10. std::string getCurrentDate();
11. /*Method declaration*/
12. std::string formatSaveString(const std::string& _format, const std::string& date, const int
cnt);
13. /*Method declaration*/
14. std::string combinePath(const std::string _base, const  std::string _exte);
15. /*Method declaration*/
16. void handleSaveDirectory(const std::string& path);
17. /*Method declaration*/
18. bool fileExists(const std::string& path);
19.
```

**Καταχώρηση 24:** **LibraryUtils.getFileCount - getCurrentDate**

```
 1. #include "pch.h"
 2.
 3. /*
 4. Returns the file count from inside the passed  path.
 5. @param const std::string _path: A string containing the absolute path to a directory.
 6. @throw fs::filesystem_error: Passed _path does not exist.
 7. @return The file count from inside the passed _path.
 8. */
 9. int getFileCount(const std::string _path) {
10.       int fileCount = 0;
11.
```

```
12.       if (!fs::exists(_path)) {
13.                 throw fs::filesystem_error("Passed _path does not exist.",
std::filesystem::directory_entry(), std::make_error_code(std::errc::no_such_file_or_directory));
14.       }
15.
16.       for (const auto& entry : fs::directory_iterator(_path)) {
17.                 //Count only files
18.                 if (entry.is_regular_file()) {
19.                           fileCount++;
20.                 }
21.       }
22.
23.       return fileCount;
24. }
25.
26. /*
27. @return The current date in a DD_MM_YYYY format as a string.
28. */
29. std::string getCurrentDate() {
30.       time_t now;
31.       time(&now);
32.
33.       struct tm current_time;
34.       localtime_s(¤t_time, &now);
35.
36.       std::stringstream ss;
37.       ss << std::setfill('0') << std::setw(2) << current_time.tm_mday << "_"
38.                 << std::setw(2) << current_time.tm_mon + 1 << "_"
39.                 << current_time.tm_year + 1900;
40.
41.       return ss.str();
42. }
43.
```

**Καταχώρηση 25:** **LibraryUtils.formatSaveString – combinePath – handleSaveDirectory - fileExists**

```
 1. /*
 2. @param const std::string& _format: The format to replace the values from.
 3. @param const std::string& date: The date value to replace the {date}.
 4. @param const int cnt: The count value to replace the {count}.
 5. @return Replaces the date and count from the passed format and returns it.
 6. e.g.: {date}_{count} - 02_03_2024_1
 7. */
 8. std::string formatSaveString(const std::string& _format, const std::string& date, const int cnt)
{
 9.       std::string result = _format;
10.       std::size_t pos = result.find(_format);
11.       if (pos != std::string::npos) {
12.                 result.replace(pos, _format.length(), date + "_" + std::to_string(cnt));
13.       }
14.       return result;
15. }
16.
17. /*
18. Combines the passed strings with the corresponding system-relative path combination symbol.
19. No checking takes place.
20. @param const std::string _base: The base absolute path.
21. @param const std::string _exte: The extension relative path.
22. @return The combined path string.
23. */
24. std::string combinePath(const std::string _base, const std::string _exte) {
25.       fs::path _comb = fs::path(_base) / fs::path(_exte);
26.       return _comb.string();
27. }
```

```
28.
29. /*
30. Creates the passed directory if it does not exist.
31. @param const std::string& path: The directory absolute path.
32. */
33. void handleSaveDirectory(const std::string& path) {
34.       fs::path directoryPath(path);
35.
36.       if (!fs::exists(directoryPath)) {
37.                 fs::create_directory(directoryPath);
38.       }
39. }
40.
41. /*
42. @param const std::string& path: Absolute path to the file.
43. @return True if the file exists, false otherwise.
44. */
45. bool fileExists(const std::string& path) {
46.       fs::path directoryPath(path);
47.
48.       return fs::exists(directoryPath);
49. }
50.
```

### 4.2.9 Snapshot error codes enumeration

Μαζί με την βιβλιοθήκη παρέχονται και οι κωδικοί επιστροφής σφαλμάτων όπως αυτοί διακρίνονται στην Εικόνα 41.

**Καταχώρηση 26:** **SnapshotLib.h - SnapshotReturnCodes**

```
1. /*
2. All the available library return codes for error handling and validation.
3. */
4. enum SnapshotReturnCodes
5. {
6.       OperationSuccessful = 0,
7.       OperationFailed = 1,
8.       CouldNotOpenFile = 2,
9.       ReadNotSuccessful = 3,
10.      DirectoryNotFound = 76,
11.      FileNotFound = 404,
12. };
13.
```

### 4.2.10 Precompiled header file

Όπως προαναφέρθηκε η βιβλιοθήκη κάνει χρήση των precompiled headers του Visual Studio όπως αυτό φαίνεται στην Εικόνα 42.

**Καταχώρηση 27:** **Pch.h**

```
1. #ifndef PCH_H
2. #define PCH_H
3. #define NOMINMAX
4.
```

```
 5. //Snapshot imports
 6. #include "framework.h"
 7. #include <unordered_map>
 8. #include <iostream>
 9. #include <fstream>
10. #include <stdexcept>
11. //Utils imports
12. #include <string>
13. #include <filesystem>
14. #include <sstream>
15. #include <ctime>
16. #include <iomanip>
17. #include <sstream>
18. //Base header imports
19. #include "SnapshotLib.h"
20. #include "LibraryUtils.h"
21.
22. //msgpack imports
23. #include "msgpack/msgpack.hpp"
24.
25. #endif //PCH_H
26.
```

## 4.3 Υλοποίηση Foreign Function Interface

Για να επιτευχθεί το exposure όλων των μεθόδων του κεφαλαίου 4.2 σε άλλες γλώσσες, χρειάστηκε να δημιουργηθεί ένα FFI το οποίο θα πράττει τα απαραίτητα configurations και θα περιέχει όλες τις μεθόδους που χρειάζεται να είναι ορατές στις host languages που θα λειτουργεί παράλληλα μαζί τους το DLL της βιβλιοθήκης.

Αρχικά στο αρχείο “SnapshotLib.h” που βρίσκονται όλες οι ταυτότητες (signatures) των μεθόδων περιέχεται το declaration ενός macro (ref to c++ macros). Συγκεκριμένα, το macro “SNAPSHOT_API” χρησιμοποιείται για τη διαχείριση της εισαγωγής και εξαγωγής μεθόδων, κλάσεων και μεταβλητών σε ένα DLL. Οι λέξεις-κλειδιά “__declspec(dllexport)” και “__declspec(dllimport)” είναι ειδικές λέξεις-κλειδιά της Microsoft που καθορίζουν αν μια μέθοδος, μεταβλητή ή κλάση εξάγεται (διατίθεται σε άλλα προγράμματα) ή εισάγεται (χρησιμοποιείται από άλλο DLL). Όταν ορίζεται το “SNAPSHOT_EXPORTS”, το “SNAPSHOT_API” επεκτείνεται σε __declspec(dllexport), επισημαίνοντας τη μέθοδο ή την κλάση για εξαγωγή. Αντίθετα, όταν το “SNAPSHOT_EXPORTS” δεν ορίζεται, συνήθως κατά τη χρήση του DLL, το “SNAPSHOT_API” επεκτείνεται σε “__declspec(dllimport)”, υποδεικνύοντας ότι η μέθοδος ή η κλάση εισάγεται από ένα DLL όπως διακρίνεται στην Εικόνα 43.

**Καταχώρηση 28:** **SnapshotLib.h – SNAPSHOT_EXPORTS**

```
1. #ifdef SNAPSHOT_EXPORTS
2. #define SNAPSHOT_API __declspec(dllexport)
3. #else
4. #define SNAPSHOT_API __declspec(dllimport)
5. #endif
6.
```

Στη συνέχεια στο ίδιο αρχείο περιέχεται και το enumeration declaration των κωδικών σφαλμάτων όπως διακρίνεται στην Εικόνα 44.

**Καταχώρηση 29:** **SnapshotLib.h - SnapshotReturnCodes**

```
/*
All the available library return codes for error handling and validation.
*/
enum SnapshotReturnCodes
{
	OperationSuccessful = 0,
	OperationFailed = 1,
	CouldNotOpenFile = 2,
	ReadNotSuccessful = 3,
	DirectoryNotFound = 76,
	FileNotFound = 404,
};

```

Τέλος, το αρχείο “SnapshotLib.h” περιέχει και όλα τα signatures των μεθόδων της βιβλιοθήκης με τη χρήση της προδιαγραφής σύνδεσης “extern C” όπως διακρίνεται στις Εικόνες 45 έως 49. Η προδιαγραφή σύνδεσης extern "C" λέει στον μεταγλωττιστή (compiler) να χρησιμοποιήσει σύνδεση (linking) C αντί για σύνδεση C++, αποφεύγοντας την αλλοίωση των ονομάτων και επιτρέποντας τη συμβατότητα με άλλες γλώσσες ή εργαλεία που περιμένουν σύνδεση τύπου C. Αυτό είναι απαραίτητο για την έκθεση μεθόδων σε ένα DLL σε προγράμματα C ή σε άλλες γλώσσες που διασυνδέονται με βιβλιοθήκες C, καθώς χαρακτηριστικά της C++ όπως το method overloading και το name mangling δεν υπάρχουν στη C.

**Καταχώρηση 30:** **SnapshotLib.h – Save Path region**

```
#pragma region Save Path
/*
setSavePath "C"-like library exposure
*/
extern "C" SNAPSHOT_API short setSavePath(const char* _savePath);
/*
getSavePath "C"-like library exposure
*/
extern "C" SNAPSHOT_API const char* getSavePath();
#pragma endregion

```

**Καταχώρηση 31:** **SnapshotLib.h – SMRI Handling region**

```
#pragma region SMRI Handling
/*
getSmri "C"-like library exposure
*/
extern "C" SNAPSHOT_API unsigned int getSmri();
/*
decreaseSmri "C"-like library exposure
*/
extern "C" SNAPSHOT_API void decreaseSmri();
/*
deleteSmriData "C"-like library exposure
*/
extern "C" SNAPSHOT_API short deleteSmriData(unsigned int _smri);
/*
getCurrentSmri "C"-like library exposure
*/
extern "C" SNAPSHOT_API unsigned int getCurrentSmri();
#pragma endregion
```

```
19.
```

**Καταχώρηση 32:** **SnapshotLib.h – Data Caching and Packing region**

```
 1. #pragma region Data Caching and Packing
 2. /*
 3. cacheData "C"-like library exposure
 4. */
 5. extern "C" SNAPSHOT_API short cacheData(unsigned int _smri, int _dataSize, unsigned char* _data,
int _refsSize, int* _refSmris);
 6. /*
 7. getData "C"-like library exposure
 8. */
 9. extern "C" SNAPSHOT_API unsigned char* getData(unsigned int _smri, int* _size);
10. /*
11. getRefSmris "C"-like library exposure
12. */
13. extern "C" SNAPSHOT_API int* getRefSmris(unsigned int _parentSmri, int* _size);
14. /*
15. packData "C"-like library exposure
16. */
17. extern "C" SNAPSHOT_API short packData();
18. #pragma endregion
19.
```

**Καταχώρηση 33:** **SnapshotLib.h – Load from File region**

```
 1. #pragma region Load from File
 2. /*
 3. setLoadFileName "C"-like library exposure
 4. */
 5. extern "C" SNAPSHOT_API short setLoadFileName(const char* _loadFileName);
 6. /*
 7. getLoadFileName "C"-like library exposure
 8. */
 9. extern "C" SNAPSHOT_API const char* getLoadFileName();
10. /*
11. unpackData "C"-like library exposure
12. */
13. extern "C" SNAPSHOT_API short unpackData();
14. #pragma endregion
15.
```

**Καταχώρηση 34:** **SnapshotLib.h – Cleanup region**

```
 1. #pragma region DLL Cleanup
 2. /*
 3. resetSmri "C"-like library exposure
 4. */
 5. extern "C" SNAPSHOT_API short resetSmri();
 6. /*
 7. resetCache "C"-like library exposure
 8. */
 9. extern "C" SNAPSHOT_API short resetCache();
10. #pragma endregion
11.
```

## 4.4 Υλοποίηση C# wrapper

Ο C# wrapper δημιουργήθηκε για να περικλείσει τις λειτουργίες τις βιβλιοθήκης και να τις κάνει διαθέσιμες σε ένα περιβάλλον .NET όπως είναι ένα Unity project. Για αυτό το λόγο χρειάστηκε να γραφτούν οι μέθοδοι της βιβλιοθήκης στη γλώσσα C# στο αρχείο "SnapshotWrapper.cs" το οποίο θα βρίσκεται στο πακέτο της βιβλιοθήκης. Το συγκεκριμένο αρχείο αρχικά περιέχει το ίδιο enumeration με τους κωδικούς επιστροφής όπως φαίνεται στην Εικόνα 50 και στην συνέχεια στις Εικόνες 51, 52 και 53 διακρίνονται όλες οι στατικές ταυτότητες των μεθόδων προς τη βιβλιοθήκη. Είναι άξιο σημείωσης, πως οπουδήποτε εμπλέκονται strings, έχει περαστεί και η ρύθμιση "CharSet.Auto" μέσα στο "DllInvoke" attribute που είναι μέρος των P/Invoke Services της .NET, ώστε να μπορεί να γίνει αυτόματα η μετατροπή των strings στο ανάλογο character set. Συγκεκριμένα, η C/C++ ακολουθεί τα ASCII και UNICODE sets ενώ η C# ακολουθεί το UNICODE character set UTF-16.

Τέλος, έχουν χρησιμοποιηθεί οι αρχικοί τύποι των primitive τύπων της C# για πιο ξεκάθαρη ανάγνωση και debugging.

**Καταχώρηση 35:** **SnapshotWrapper.SnapshotReturnCodes enum**

```
        /// <summary>
        /// All the available return code of the library.
        /// </summary>
        enum SnapshotReturnCodes {
            OperationSuccessful = 0,
            OperationFailed = 1,
            CouldNotOpenFile = 2,
            ReadNotSuccessful = 3,
            DirectoryNotFound = 76,
            FileNotFound = 404,
        }

```

**Καταχώρηση 36:** **SnapshotWrapper.cs – Save Path and SMRI Handling external calls**

```
        //Save path
        ///<summary>Dll method invoke</summary>
        [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
        private static extern Int16 setSavePath(string _path);
        ///<summary>Dll method invoke</summary>
        [DllImport("SnapshotLib.dll", CharSet = CharSet.Auto, CallingConvention =
CallingConvention.Cdecl)]
        private static extern IntPtr getSavePath();

        //SMRI Handling
        ///<summary>Dll method invoke</summary>
        [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
        private static extern UInt32 getSmri();
        ///<summary>Dll method invoke</summary>
        [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
        private static extern void decreaseSmri();
        ///<summary>Dll method invoke</summary>
        [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
        private static extern Int32 getCurrentSmri();
        ///<summary>Dll method invoke</summary>
        [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
        private static extern Int16 deleteSmriData(UInt32 _smri);

```

**Καταχώρηση 37:** **SnapshotWrapper.cs – Data caching and Load from File external calls**

```
 1.         //Data caching and packing
 2.         ///<summary>Dll method invoke</summary>
 3.         [DllImport("SnapshotLib.dll", CharSet = CharSet.Auto, CallingConvention =
CallingConvention.Cdecl)]
 4.         private static extern Int16 cacheData(UInt32 _smri, Int32 _dataSize, byte[] _data, int
_refSmrisSize, Int32[] _refSmris);
 5.         ///<summary>Dll method invoke</summary>
 6.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
 7.         private static extern IntPtr getData(UInt32 _smri, out Int32 _arraySize);
 8.         ///<summary>Dll method invoke</summary>
 9.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
10.         private static extern IntPtr getRefSmris(UInt32 _parentSmri, out Int32 _size);
11.         ///<summary>Dll method invoke</summary>
12.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
13.         private static extern Int16 packData();
14.
15.         //Load from file
16.         ///<summary>Dll method invoke</summary>
17.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
18.         private static extern Int16 setLoadFileName(string _saveFileName);
19.         ///<summary>Dll method invoke</summary>
20.         [DllImport("SnapshotLib.dll", CharSet = CharSet.Auto, CallingConvention =
CallingConvention.Cdecl)]
21.         private static extern IntPtr getLoadFileName();
22.         ///<summary>Dll method invoke</summary>
23.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
24.         private static extern Int16 unpackData();
25.
```

**Καταχώρηση 38:** **SnapshotWrapper.cs – Memory cleanup external calls**

```
1.         //Memory cleanup
2.         ///<summary>Dll method invoke</summary>
3.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
4.         private static extern Int16 resetSmri();
5.         ///<summary>Dll method invoke</summary>
6.         [DllImport("SnapshotLib.dll", CallingConvention = CallingConvention.Cdecl)]
7.         private static extern Int16 resetCache();
8.
```

Όπως και στη βιβλιοθήκη όλες οι μέθοδοι είναι χωρισμένες σε εμφανή και αντίστοιχα regions. Στα παρακάτω κεφάλαια θα αναλυθούν οι υλοποιήσεις των μεθόδων της βιβλιοθήκης μέσα στον C# wrapper. Όλες οι ακόλουθοι μέθοδοι είναι public και στατικές ώστε να μπορούν να χρησιμοποιηθούν ελεύθερα μέσα στον κώδικα.

### 4.4.1 Snapshot Wrapper Save Path region

Αρχικά, όπως διακρίνεται στην Εικόνα 54, οι μέθοδοι “SetSavePath” και “GetSavePath” είναι υπεύθυνες να χειριστούν τα αποτελέσματα των καλεσμάτων στις αντιστοιχες μεθόδους μέσω των invocation μεθόδων.

Τέλος, στην περίπτωση της “GetSavePath” το αποτέλεσμα που επιστρέφει η “getSavePath” είναι ένας pointer, ο οποίος με τη χρήση της κλάσης Marshal της C#, μετατρέπεται σε ένα C# string. Και στις δυο μεθόδους, γίνεται είτε επιστροφή κάποιου Boolean value ή κάπου runtime exception.

### 4.4.2 Snapshot Wrapper SMRI Handling region

Στο region του SMRI handling διακρίνονται οι μέθοδοι για την αύξηση, μείωση και απολαβή του τρέχοντος SMRI για χρήση σε αντικείμενα του παιχνιδιού αλλά και για τη διαγραφή του πακέτου δεδομένων του εισαγόμενου SMRI, όπως φαίνεται στην Εικόνα 55. Όλες οι μέθοδοι διαχειρίζονται τις επιστρεφόμενες τιμές των μεθόδων της βιβλιοθήκης και επιστρέφουν είτε κάποια Boolean τιμή είτε κάποιοι exception.

**Καταχώρηση 39:** **SnapshotWrapper.cs – Save Path region**

```
1.          #region Save Path
2.          /// <summary>
3.          /// Sets the save path inside the dll.
4.          /// </summary>
5.          /// <param name="_path">The absolute path to the save directory</param>
6.          /// <returns>True if the set was successful, false otherwise.</returns>
7.          public static bool SetSavePath(string _path) {
8.              try {
9.                  Int16 ec = setSavePath(_path);
10.                 if (ec == (Int16)SnapshotReturnCodes.DirectoryNotFound) {
11.                     throw new DirectoryNotFoundException($"Passed path {_path} does not
exist.");
12.                 }
13.                 return ec == (Int16)SnapshotReturnCodes.OperationSuccessful;
14.             } catch (Exception exception) {
15.                 Debug.LogError($"Could not set the DLL save path:\n{exception}");
16.                 return false;
17.             }
18.         }
19.
20.         /// <summary>
21.         /// Returns the absolute save path from inside the dll.
22.         /// </summary>
23.         /// <returns>Returns the absolute save path from inside the dll.</returns>
24.         /// <exception cref="Exception">Could not get the current save path from DLL</exception>
25.         public static string GetSavePath() {
26.             try {
27.                 IntPtr strPtr = getSavePath();
28.                 return Marshal.PtrToStringAnsi(strPtr);
29.             } catch (Exception exception) {
30.                 throw new Exception("Could not get the current save path from DLL:\n{0}",
exception);
31.             }
32.         }
33.         #endregion
34.
```

### 4.4.3 Snapshot Wrapper Data Caching and Packing region

Αντίστοιχα με τις προηγούμενες υλοποιήσεις, στο region αυτό βρίσκονται όλες οι μέθοδοι σχετικά με το caching και την λήψη των δεδομένων μέσα από τη βιβλιοθήκη. Αξιοσημείωτες, ωστόσο, είναι οι μέθοδοι “GetData” και “GetRefSmris” όπου λαμβάνουν τα ζητούμενα δεδομένα μέσα από τη βιβλιοθήκη και μέσω της κλάσης Marshal κάνουν την ανάλογη αντιγραφή των επιστρεφόμενων arrays από τη C++ στους τύπους της C#, όπου είναι “byte[]” και “int[]” αντίστοιχα. Οι διαδικασίες αυτές φαίνονται στις Εικόνες 56 και 57 στις σειρές 193-200 και 216-219.

**Καταχώρηση 40:** **SnapshotWrapper.cs – SMRI Handling region**

```
 1.         #region SMRI Handling
 2.         /// <summary>
 3.         /// Increases and returns the global DLL SMRI used for data storing and reference
preservation.
 4.         /// </summary>
 5.         /// <returns>A uint representing the SMRI in the DLL</returns>
 6.         /// <exception cref="Exception">Could not retrieve SMRI from DLL.</exception>
 7.         public static uint GetSmri() {
 8.             try {
 9.                 return getSmri();
10.             } catch (Exception exception) {
11.                 throw new Exception("Could not retrieve increased smri due to:\n{0}",
exception);
12.             }
13.         }
14.
15.         /// <summary>
16.         /// Decreases the global SMRI by 1. Use when <seealso cref="GetSmri()"/> fails.
17.         /// </summary>
18.         /// <exception cref="Exception">Could not decrease SMRI from DLL</exception>
19.         public static void DecreaseSmri() {
20.             try {
21.                 decreaseSmri();
22.             } catch (Exception exception) {
23.                 throw new Exception("Could not decrease smri due to:\n{0}", exception);
24.             }
25.         }
26.
27.         /// <summary>
28.         /// Returns the current non-incremented global SMRI from the DLL.
29.         /// </summary>
30.         /// <exception cref="Exception">Could not retrieve current SMRI from DLL</exception>
31.         public static int GetCurrentSmri() {
32.             try {
33.                 return getCurrentSmri();
34.             } catch (Exception exception) {
35.                 throw new Exception("Could not retrieve current smri due to:\n{0}", exception);
36.             }
37.         }
38.
39.         /// <summary>
40.         /// Deletes the data associated with the passed SMRI inside the dll.
41.         /// </summary>
42.         /// <param name="_smri">The SMRI to delete data from</param>
43.         /// <returns>True if the deletion was successful, false otherwise.</returns>
44.         /// <exception cref="Exception">Could not delete the smri data</exception>
45.         public static bool DeleteSmriData(uint _smri) {
46.             try {
47.                 return deleteSmriData(_smri) == (Int16)SnapshotReturnCodes.OperationSuccessful;
48.             } catch (Exception exception) {
49.                 throw new Exception("Could not delete the smri data due to:\n{0}", exception);
50.             }
51.         }
52.         #endregion
53.
```

**Καταχώρηση 41:** **SnapshotWrapper.cs – Data Caching region**

```
 1.         #region Data Caching - Packing
 2.         /// <summary>
 3.         /// Caches the passed data and references SMRIs inside the dll cache.
```

```
4.          /// Values are copies so it's safe to also delete them if you want.
5.          /// </summary>
6.          /// <param name="_smri">The SMRI to associate the data array to</param>
7.          /// <param name="_data">The data to copy over to the dll</param>
8.          /// <param name="_refSmris">The references SMRI this SMRI needs.</param>
9.          /// <returns>True if the caching was successful, false otherwise.</returns>
10.         public static bool CacheData(uint _smri, byte[] _data, int[] _refSmris) {
11.             try {
12.                 return cacheData(_smri, _data.Length, _data, _refSmris.Length, _refSmris) ==
(Int16)SnapshotReturnCodes.OperationSuccessful;
13.             } catch (Exception exception) {
14.                 Debug.LogError($"Could not cache the passed (SMRI: {_smri}) to the
DLL:\n{exception}");
15.                 return false;
16.             }
17.         }
18.
19.         /// <summary>
20.         /// Returns the associated byte array of the passed smri from the dll.
21.         /// </summary>
22.         /// <param name="_smri">The SMRI to retrieve data for</param>
23.         /// <returns>A byte array containing the deserialized data or null.</returns>
24.         public static byte[] GetData(uint _smri) {
25.             try {
26.                 Int32 size = -1;
27.                 IntPtr containerData = getData(_smri, out size);
28.                 if (containerData == IntPtr.Zero) {
29.                     throw new Exception($"Passed SMRI: {_smri} returned a nullptr for
container.");
30.                 }
31.
32.                 byte[] data = new byte[size];
33.                 Marshal.Copy(containerData, data, 0, size);
34.
35.                 return data;
36.             } catch (Exception exception) {
37.                 Debug.LogError($"Could not get data for the passed SMRI: {_smri} from the
DLL:\n{exception}");
38.                 return null;
39.             }
40.         }
41.
42.         /// <summary>
43.         /// Returns an int array containing the referenced SMRI of the passed SMRI from the dll.
44.         /// </summary>
45.         /// <param name="_parentSmri">The SMRI to retrieve referenced SMRIs for.</param>
46.         /// <returns>A byte array containing the data or null.</returns>
47.         public static int[] GetRefSmris(uint _parentSmri) {
48.             try {
49.                 int size = -1;
50.                 IntPtr ptr = getRefSmris(_parentSmri, out size);
51.                 int[] refSmris = new int[size];
52.                 Marshal.Copy(ptr, refSmris, 0, size);
53.
54.                 return refSmris;
55.             } catch (Exception exception) {
56.                 Debug.LogError($"Could not get ref SMRIs for the passed SMRI: {_parentSmri} from
the DLL:\n{exception}");
57.                 return null;
58.             }
59.         }
60.
61.         /// <summary>
62.         /// Starts the packing sequence of the cached data inside the dll.
63.         /// </summary>
```

```
64.         /// <returns>True if packing was successful, false otherwise.</returns>
65.         public static bool PackData() {
66.             try {
67.                 return packData() == (Int16)SnapshotReturnCodes.OperationSuccessful;
68.             } catch (Exception exception) {
69.                 Debug.LogError($"Could not pack data in the DLL:\n{exception}");
70.                 return false;
71.             }
72.         }
73.         #endregion
74.
```

### 4.4.4 Snapshot Wrapper Load from File region

Στο συγκεκριμένο region, βρίσκονται όλες οι μέθοδοι που είναι υπεύθυνες για τη διαχείριση του file name από το οποίο θα φορτωθούν – όταν το επιλέξει ο προγραμματιστής – τα δεδομένα μέσα από τον φάκελο που έχει εισάγει από τη μέθοδο “SetSavePath”. Επίσης, εδώ βρίσκεται και η μέθοδος “UnpackData” όπου όταν καλεστεί, η βιβλιοθήκη ξεκινάει την διαδικασία του deserialization, που όπως αναφέρθηκε στο κεφάλαιο 4.2.6, επαναφέρει τα αποθηκευμένα δεδομένα μέσα στην δομή αποθήκευσης της βιβλιοθήκης και τα κάνει προσβάσιμα από τις μεθόδους “GetData” και “GetRefSmris”. Οι μέθοδοι αυτοί διακρίνονται στις Εικόνες 58 και 59.

**Καταχώρηση 42:** **SnapshotWrapper.cs – Load from File region**

```
 1.         #region Load from file
 2.         /// <summary>
 3.         /// Sets the save file name inside the dll.
 4.         /// </summary>
 5.         /// <param name="_loadFromFileName">The save file name to unpack from.</param>
 6.         /// <returns>True if set was successful, false otherwise.</returns>
 7.         public static bool SetLoadFileName(string _loadFromFileName) {
 8.             try {
 9.                 Int16 ec = setLoadFileName(_loadFromFileName);
10.                 if (ec == (Int16)SnapshotReturnCodes.FileNotFound) {
11.                     throw new DirectoryNotFoundException($"Passed filename {_loadFromFileName}
does not exist inside the save path: {GetSavePath()}.");
12.                 }
13.                 return ec == (Int16)SnapshotReturnCodes.OperationSuccessful;
14.             } catch (Exception exception) {
15.                 Debug.LogError($"Could not set the DLL load from filename:\n{exception}");
16.                 return false;
17.             }
18.         }
19.
20.         /// <summary>
21.         /// Returns the cached save file name from inside the dll.
22.         /// </summary>
23.         /// <returns>The saved file name stored in the dll. Can be an empty string.</returns>
24.         /// <exception cref="Exception">Could not get the current load from filename from
DLL</exception>
25.         public static string GetLoadFileName() {
26.             try {
27.                 IntPtr strPtr = getLoadFileName();
28.                 return Marshal.PtrToStringAnsi(strPtr);
29.             } catch (Exception exception) {
30.                 throw new Exception("Could not get the current load from filename from
DLL:\n{0}", exception);
31.             }
```

```
32.         }
33.
34.         /// <summary>
35.         /// Starts the unpacking sequence inside the dll to deserialize the serialized data and
store them in the dll cache.
36.         /// GlobalSMRI is set to be equal to the unpacked data size.
37.         /// Cached datas are overwritten.
38.         /// </summary>
39.         /// <returns>True if unpacking was successful, false otherwise.</returns>
40.         public static bool UnpackData() {
41.             try {
42.                 return unpackData() == (Int16)SnapshotReturnCodes.OperationSuccessful;
43.             } catch (Exception exception) {
44.                 Debug.LogError($"Could not pack data in the DLL:\n{exception}");
45.                 return false;
46.             }
47.         }
48.         #endregion
49.
```

### 4.4.5 Snapshot Wrapper Memory Cleanup region

Τέλος, στο κλείσιμο του wrapper, έχουν υλοποιηθεί οι μέθοδοι διαχείρισης της μνήμης που απλά επιστρέφουν μία τιμή Boolean σε περίπτωση αποτυχίας ή επιτυχίας της εκκαθάρισης της μνήμης της βιβλιοθήκης.

**Καταχώρηση 43:** **SnapshotWrapper.cs – Memory cleanup region**

```
 1.         #region Memory Cleanup
 2.         /// <summary>
 3.         /// Resets the DLL global SMRI back to its default value: -1.
 4.         /// </summary>
 5.         /// <returns>True if the reset was succesful, false otherwise with an error
log.</returns>
 6.         public static bool ResetSmri() {
 7.             try {
 8.                 return resetSmri() == (Int16)SnapshotReturnCodes.OperationSuccessful;
 9.             } catch (Exception exception) {
10.                 Debug.LogError($"Could not reset the DLL SMRI:\n{exception}");
11.                 return false;
12.             }
13.         }
14.
15.         /// <summary>
16.         /// Deallocates the dll data cache and clears it.
17.         /// Resets the set saved directory value to empty.
18.         /// Resets the set file name value to empty.
19.         /// </summary>
20.         /// <returns>True if the reset was successful, false otherwise.</returns>
21.         public static bool ResetCache() {
22.             try {
23.                 return resetCache() == (Int16)SnapshotReturnCodes.OperationSuccessful;
24.             } catch (Exception exception) {
25.                 Debug.LogError($"Could not reset the cache of the DLL:\n{exception}");
26.                 return false;
27.             }
```

```
28.         }
29.         #endregion
30.
```

## 4.5 Test environment στη Unity

Με βάση τον σχεδιασμό που προτάθηκε στο κεφάλαιο 3.9 αλλά και τα επί μέρους κεφάλαια του, το test environment στήθηκε στη μηχανή Unity ώστε να μπορέσουν να διεξαχθούν οι απαραίτητες δοκιμές αλλά και να γίνει χρήση της προτεινόμενης αρχιτεκτονικής. Ο στόχος των τεστ αυτών είναι αρχικά να γίνει επιτυχής το serialization (saving) και αργότερα το deserialization (loading) των δεδομένων και δεύτερων να γίνουν μερικές μετρήσεις στο μέγεθος του Garbage Collection κατά τις διαδικασίες αυτές.

### 4.5.1 Δεδομένα και αντικείμενα

Όπως σχεδιάστηκε στο κεφάλαιο 3.9.1, δημιουργήθηκε η κλάση “Player” όπου περιέχει τα class fields _Health, _Stamina, _Shield, _IsActive και _Inventory. Το Inventory γίνεται reference δυναμικά στο runtime μέσα από τη σκηνή. Αυτή η κλάση προστέθηκε επάνω σε ένα GameObject ονόματη Player και έγινε ένα prefab στο φάκελο prefabs.

Στη συνέχεια, δημιουργήθηκε η κλάση Inventory, όπου περιέχει τις μεταβλητές _MaxItems και μία λίστα όπου δέχεται references σε Weapon instances κατά το runtime της εφαρμογής. Εξίσου και αυτή η κλάση προστέθηκε επάνω σε ένα GameObject ονόματη Inventory και έγινε ένα prefab στο φάκελο prefabs.

Τέλος, δημιουργήθηκε η κλάση Weapon όπου τα ζητούμενα δεδομένα βάση σχεδιασμού ήταν οι τιμές των μεταβλητών _Ammo, _Loaded και το reference στο Inventory το οποίο ανήκουν. Ακολούθησαν την ίδια διαδικασία για τη δημιουργία Weapon prefabs.

### 4.5.2 Χρήση προτεινόμενης αρχιτεκτονικής

Αρχικά με βάση τον σχεδιασμό του κεφαλαίου 3.9.2, δημιουργήθηκε η κλάση singleton ονόματη Common και προστέθηκε επάνω σε ένα GameObject της σκηνής. Η κλάση “Common” περιέχει απλός ένα static field με reference στον εαυτό της κα είναι υπεύθυνη για τη δημιουργία του instance του “SaveManager” και την εύρεση του “WorldLoader” που θα είναι παρών στη σκήνη.

**Καταχώρηση 44:** **ProposedArchitecture.Common class**

```
 1. namespace ProposedArchitecture {
 2.
 3.     /// <summary>
 4.     /// Manager hub
 5.     /// </summary>
 6.     [DefaultExecutionOrder(-500)]
 7.     public class Common : MonoBehaviour {
 8.         ///<summary>Returns the Common singleton</summary>
 9.         public static Common Instance;
10.
11.         ///<summary>Reference to the WorldLoader</summary>
12.         WorldLoader _WorldLoader;
```

```
13.         ///<summary>Reference to the SaveManager</summary>
14.         SaveManager _SaveManager;
15.
16.         ///<summary>Returns the WorldLoader reference</summary>
17.         public WorldLoader WorldLoader => _WorldLoader;
18.         ///<summary>Returns the SaveManager reference</summary>
19.         public SaveManager SaveManager => _SaveManager;
20.
21.         void Awake() {
22.             Instance = this;
23.
24.             _WorldLoader = GetComponent<WorldLoader>();
25.             _SaveManager = new SaveManager(Instance);
26.         }
27.
28.         //Cleanup when the game closes.
29.         void OnDestroy() {
30.             Debug.Log($"Save Manager cleanup result: {SaveManager.Cleanup()}");
31.         }
32.     }
33. }
34.
```

Στη συνέχεια, δημιουργήθηκε η κλάση “GlobalProperties” στην οποία αναγράφονται τα save directory και save directory name paths.

**Καταχώρηση 45:** **ProposedArchitecture.GlobalProperties class**

```
 1. namespace ProposedArchitecture {
 2.
 3.     /// <summary>
 4.     /// Stores global game values and constants
 5.     /// </summary>
 6.     public static class GlobalProperties {
 7.         ///<summary>The save directory absolute path</summary>
 8.         public static readonly string SavePath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
 9.         ///<summary>The save folder name.</summary>
10.         public const string SaveFolderName = "SnapshotSaves";
11.     }
12. }
13.
```

Επιπλέον, δημιουργήθηκαν τα δύο βασικά interfaces της αρχιτεκτονικής ονόματη “ISnapshot” και “ISnapshotModel”. Το “ISnapshot” είναι το interface το οποίο θα κάνουν implement οι κλάσης προς serialization και το interface “ISnapshotModel” θα χρησιμοποιηθεί για των μοντέλων δεδομένων αυτών των κλάσεων.

**Καταχώρηση 46:** **ProposedArchitecture.ISnapshot interface**

```
 1. namespace ProposedArchitecture {
 2.
 3.     /// <summary>
 4.     /// Marks a class as ISnapshot-able
 5.     /// </summary>
 6.     public interface ISnapshot {
 7.         ///<summary>The class should have an SMRI field</summary>
 8.         public uint Smri { get; }
 9.
```

```
10.         public void RegisterToSaveManager();
11.         public void CacheModel();
12.         public ISnapshotModel ConstructModel();
13.         public void UnregisterToSaveManager();
14.         public void LoadModel(ISnapshotModel _model);
15.         public void RetrieveReferences(int[] _refSmris);
16.         public Type GetSnapshotModelType();
17.     }
18. }
19.
```

**Καταχώρηση 47:** **ProposedArchitecture.ISnapshotModel interface**

```
 1. namespace ProposedArchitecture {
 2.
 3.     public interface ISnapshotModel{
 4.         ///<summary>The class/struct should have an SMRI field</summary>
 5.         public uint Smri {get; set;}
 6.         ///<summary>The class/struct should have an int array to store its references
SMRIs</summary>
 7.         public int[] RefSmris {get; set;}
 8.     }
 9. }
```

Ακολουθώντας την αρχιτεκτονική, θα χρειαστεί η δημιουργία και της κλάσης "SaveManager" όπου είναι υπεύθυνη για τη σωστή διαχείριση των Snapshot κλάσεων και των μοντέλων τους. Η αρχικοποίηση της κλάσης συμβαίνει από τη κλάση "Common" όπου και δίνεται ένα reference αυτής. Τα βασικά fields της κλάσης SaveManager είναι ένα field τύπου List<ISnapshot> _Snapshots και ένα field τύπου List<ISnapshotModel> _Models που αρχικοποιούνται στον constructor της κλάσης όπως φαίνεται στην Εικόνα 65. Επιπλέον, μέσα στον constructor της κλάσης καλείται η μέθοδος "SetSavePath" από τον "SnapshotWrapper" ώστε να αρχικοποιηθεί και το directory path των saved αρχείων.

**Καταχώρηση 48:** **ProposedArchitecture.SaveManager class**

```
 1. using System;
 2. using System.Collections.Generic;
 3. using System.IO;
 4. using Snapshot;
 5.
 6. namespace ProposedArchitecture {
 7.
 8.     /// <summary>
 9.     /// Responsible for handling the SnapshotWrapper, ISnapshot and ISnapshotModel instances.
10.     /// Hands out SMRIs.
11.     /// </summary>
12.     public class SaveManager {
13.         ///<summary>Reference to the Common instance</summary>
14.         Common _Common;
15.         ///<summary>Caches the ISnapshot references of the game</summary>
16.         List<ISnapshot> _Snapshots;
17.         ///<summary>Caches the ISnapshotModel references of the ISnapshots</summary>
18.         List<ISnapshotModel> _Models;
19.
20.         ///<summary>Returns a read only collection of the Snapshot reference cache.</summary>
21.         public IReadOnlyList<ISnapshot> Snapshots => _Snapshots;
22.
23.         ///<summary>Register to this event to get notified before the packing sequence
starts.</summary>
24.         event Action OnSnapshotStart;
25.         ///<summary>Raises the OnSnapshotStart event</summary>
```

```
26.         void RaiseOnSnapshotStart() {
27.             OnSnapshotStart?.Invoke();
28.         }
29.
30.         /// <summary>
31.         /// Creates a SaveManager instance
32.         /// </summary>
33.         /// <param name="_common">Reference to the Common instance</param>
34.         public SaveManager(Common _common) {
35.             this._Common = _common;
36.             this._Snapshots = new List<ISnapshot>();
37.             this._Models = new List<ISnapshotModel>();
38.
39.             //Set the initial save directory
40.             SnapshotWrapper.SetSavePath(Path.Combine(GlobalProperties.SavePath,
GlobalProperties.SaveFolderName));
41.         }
42.
```

Στη συνέχεια, δημιουργήθηκε η μέθοδος “RegisterModel” όπου δέχεται ένα ISnapshot instance σαν reference και μέσω του interface ISnapshot, προσθέτουν την μέθοδο “CacheModel” στο event “OnSnapshotStart”. Η μέθοδος αυτή επιστρέφει το SMRI το οποίο η εκάστοτε κλάση που κάνει implement το interface ISnapshot, θα πρέπει να αποθηκεύσει κατά την εγγραφή της στον Save Manager.

Η μέθοδος “Save” είναι υπεύθυνη για την καταγραφή όλων των ISnapshotModels μέσα στη δομή αποθήκευσης της μέσω της χρήσης του OnSnapshotStart event, όπου όταν γίνει raise, όλες οι ISnapshot κλάσεις εγγεγραμμένες στον Save Manager θα κληθούν να αποθηκεύσουν ένα ISnapshotModel instance μέσα στον “SaveManager” μέσω της μεθόδου “CacheModel”.

**Καταχώρηση 49:** **ProposedArchitecture.SaveManager class (Continuation 1)**

```
 1.         /// <summary>
 2.         /// Registers the passed ISnapshot for data caching upon packing.
 3.         /// </summary>
 4.         /// <param name="_snapshot">The snapshot instance</param>
 5.         void RegisterToSnapshot(ISnapshot _snapshot) {
 6.             OnSnapshotStart += _snapshot.CacheModel;
 7.         }
 8.
 9.         /// <summary>
10.         /// Unregisters the passed ISnapshot from data caching upon packing.
11.         /// The passed snapshot is also removed from the Snapshot list.
12.         /// </summary>
13.         /// <param name="_snapshot"></param>
14.         public void UnregisterFromSnapshot(ISnapshot _snapshot) {
15.             OnSnapshotStart -= _snapshot.CacheModel;
16.             _Snapshots.Remove(_snapshot);
17.         }
18.
19.         /// <summary>
20.         /// Registers the passed ISnapshot instance to the serialization event handler and adds
it to the ISnapshot reference list.
21.         /// </summary>
22.         /// <param name="_snapshot">The snapshot instance</param>
23.         /// <returns>The SMRI of the registered model.</returns>
24.         public uint RegisterModel(ISnapshot _snapshot) {
25.             _Snapshots.Add(_snapshot);
26.             RegisterToSnapshot(_snapshot);
27.
```

```
28.             return SnapshotWrapper.GetSmri();
29.         }
30.
```

**Καταχώρηση 50:** **ProposedArchitecture.SaveManager class (Continuation 2)**

```
 1.         ///<summary>
 2.         /// Kicks off the packing sequence. All the cached ISnapshot.CacheData methods get
called and
 3.         /// their data are serialized and passed to the internal DLL library cache.
 4.         /// The cached SaveManager models list gets cleared afterwards.
 5.         ///</summary>
 6.         public void Save() {
 7.             RaiseOnSnapshotStart();
 8.
 9.             byte[] bytes;
10.             int[] refSmris;
11.
12.             for (int i = 0; i < _Models.Count; i++) {
13.                 bytes = MessagePack.MessagePackSerializer.Serialize<object>(_Models[i]);
14.                 refSmris = _Models[i].RefSmris;
15.
16.                 SnapshotWrapper.CacheData(_Models[i].Smri, bytes, refSmris);
17.             }
18.
19.             _Models = new List<ISnapshotModel>();
20.
21.             SnapshotWrapper.PackData();
22.         }
23.
24.         /// <summary>
25.         /// Stores the passed model to the SaveManager data container list.
26.         /// </summary>
27.         /// <param name="_model">The snapshot data container instance</param>
28.         public void CacheModel(ISnapshotModel _model) {
29.             _Models.Add(_model);
30.         }
31.
```

Στο τέλος του Save Manager, υπάρχει η διαδικασία του unpacking των δεδομένων, όπου αφού οι κλάσεις ISnapshot έχουν ξανά φορτωθεί στον κόσμο σαν να είναι ένα καινούργιο save, ένα iteration τους φορτώνει τα αποθηκευμένα δεδομένα τους. Τα object references που προϋπήρχαν κατά τη διάρκεια του saving και πριν κλείσει η εφαρμογή με τη κλήση της μεθόδου “RetrieveReferences” θα επιστρέψουν στην προηγούμενη τους μορφή. Ολόκληρη η διαδικασία αυτή διακρίνεται στην Εικόνα 68.

**Καταχώρηση 51:** **ProposedArchitecture.SaveManager class (Continuation 3)**

```
 1.         /// <summary>
 2.         /// Kicks off the Unpacking mechanism contained in the passed fileName.
 3.         /// The deserialized data are stored inside the SaveManager ISnapshotModel cache for
accessing.
 4.         /// Each cached ISnapshot.LoadModel and ISnapshot.RetrieveReferences method gets called
after each data retrieval from the dll cache.
 5.         /// </summary>
 6.         /// <param name="_fileName"></param>
 7.         public void LoadSaveFile(string _fileName) {
 8.             if (SnapshotWrapper.SetLoadFileName(_fileName)) {
 9.                 SnapshotWrapper.UnpackData();
10.
11.                 byte[] bytes;
```

```
12.                 ISnapshotModel model;
13.                 foreach (ISnapshot snapshot in _Snapshots) {
14.                     bytes = SnapshotWrapper.GetData(snapshot.Smri);
15.                     model =
(ISnapshotModel)MessagePack.MessagePackSerializer.Deserialize(snapshot.GetSnapshotModelType(),
bytes);
16.                     snapshot.LoadModel(model);
17.                 }
18.
19.                 foreach (ISnapshot snapshot in _Snapshots) {
20.                     snapshot.RetrieveReferences(SnapshotWrapper.GetRefSmris(snapshot.Smri));
21.                 }
22.             }
23.         }
24.
25.         /// <summary>
26.         /// Resets the dll library caches and SMRI.
27.         /// </summary>
28.         public bool Cleanup() {
29.             return SnapshotWrapper.ResetCache() && SnapshotWrapper.ResetSmri();
30.         }
31.     }
32. }
33.
```

### 4.5.3 Χρήση των ISnapshot

Για να ολοκληρωθεί η διαδικασία του saving οι κλάσεις Player, Inventory και Weapon πρέπει σμιλευθούν για να κάνουν χρήση του ISnapshot interface και χωρίζονται σε δύο μέρη. Το μέρος του saving και το μέρος του loading. Το μέρος του saving διακρίνεται στην Εικόνα 69 και το μέρος του loading διακρίνεται στη Εικόνα 70.

**Καταχώρηση 52:** **ProposedArchitecture – ISnapshot usage on a class**

```
 1.         void Start() {
 2.             //From load startup
 3.             if (Common.Instance.WorldLoader.FromLoad) { return; }
 4.
 5.             //Normal startup
 6.             _Inventory = FindObjectOfType<Inventory>();
 7.             Weapon[] temp = FindObjectsOfType<Weapon>();
 8.             for (int i = 0; i < temp.Length; i++) {
 9.                 temp[i].SetInventory(_Inventory);
10.                 _Inventory.AddWeapon(temp[i]);
11.             }
12.         }
13.
14.         ///<summary>Register the player to the save manager and set its SMRI</summary>
15.         public void RegisterToSaveManager() {
16.             _Smri = Common.Instance.SaveManager.RegisterModel(this);
17.         }
18.
19.         ///<summary>Dynamically called when its time to save</summary>
20.         public void CacheModel() {
21.             Common.Instance.SaveManager.CacheModel(ConstructModel());
22.         }
23.
24.         /// <summary>
25.         /// Returns an ISnapshotModel with the player needed data.
```

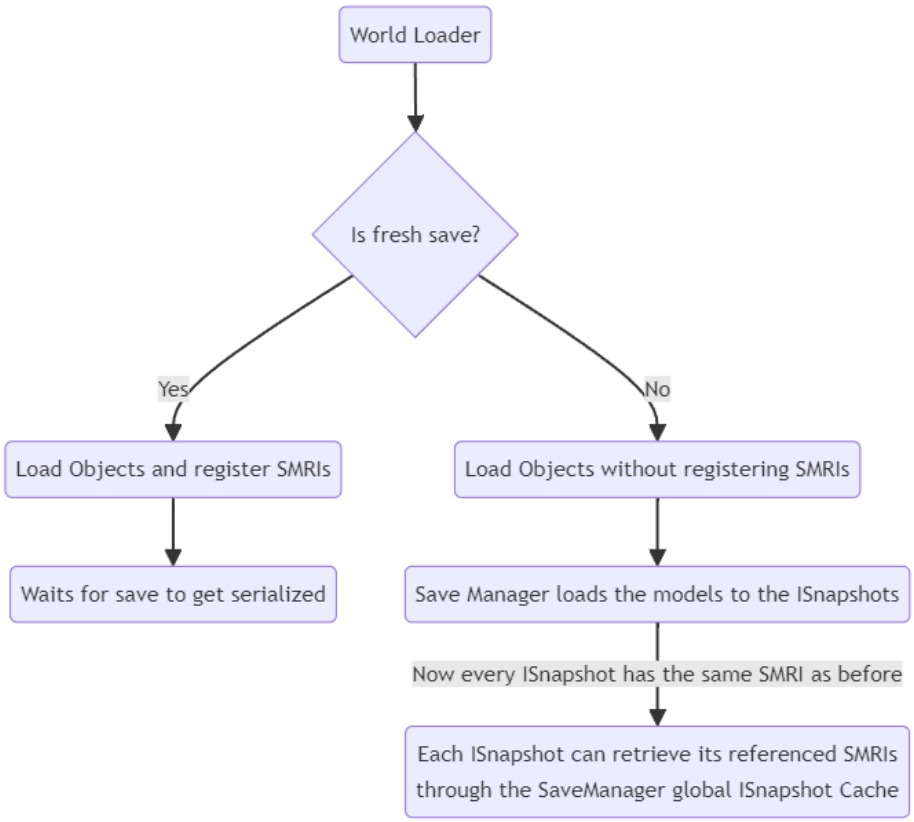
```
26.         /// </summary>
27.         public ISnapshotModel ConstructModel() {
28.             SPlayer temp = new SPlayer() {
29.                 Smri = this.Smri,
30.                 //Cache the SMRIs here
31.                 RefSmris = new int[]{
32.                     (int)_Inventory.Smri,
33.                 },
34.                 _Health = this._Health,
35.                 _Stamina = this._Stamina,
36.                 _Shield = this._Shield,
37.                 _IsAlive = this._IsAlive,
38.                 _Position = transform.position,
39.                 _Rotation = transform.rotation,
40.             };
41.
42.             return temp;
43.         }
44.
```

**Καταχώρηση 53:** **ProposedArchitecture – ISnapshot usage on a class (Continuation)**

```
 1.         ///<summary>The ISnapshot should get unregistered if it gets destroyed</summary>
 2.         void OnDestroy() {
 3.             UnregisterToSaveManager();
 4.         }
 5.
 6.         ///<summary>Unregisters the reference from the save manager</summary>
 7.         public void UnregisterToSaveManager() {
 8.             Common.Instance.SaveManager.UnregisterFromSnapshot(this);
 9.         }
10.
11.         /// <summary>
12.         /// Sets the player fields from the incoming deserialized model
13.         /// </summary>
14.         /// <param name="_model">SPlayer model containing the deserialized data</param>
15.         public void LoadModel(ISnapshotModel _model) {
16.             SPlayer model = (SPlayer)_model;
17.
18.             _Health = model._Health;
19.             _Stamina = model._Stamina;
20.             _Shield = model._Shield;
21.             _IsAlive = model._IsAlive;
22.             transform.position = model._Position;
23.             transform.rotation = model._Rotation;
24.         }
25.
26.         /// <summary>
27.         /// Sets any reference the player may have, like its inventory.
28.         /// </summary>
29.         public void RetrieveReferences(int[] _refSmris) {
30.             for (int i = 0; i < _refSmris.Length; i++) {
31.                 _Inventory = (Inventory)Common.Instance.SaveManager.Snapshots[_refSmris[i]];
32.             }
33.         }
34.
35.         /// <summary>
36.         /// Returns the type of the ISnapshotModel this ISnapshot's data get represented.
37.         /// </summary>
38.         public Type GetSnapshotModelType() {
39.             return typeof(SPlayer);
40.         }
41.     }
```

```
42.
```

Περιληπτικά, ολόκληρη η διαδικασία μπορεί να διακριθεί στο διάγραμμα της Εικόνας 71.




*Εικόνα 16: Architecture, Save loading diagram*

## 4.6 Προβλήματα κατά την υλοποίηση

Κατά την υλοποίηση της βιβλιοθήκης υπήρξαν μερικά μικρά προβλήματα incompatibility μεταξύ των εκδόσεων της C++ και του MsgPack που όμως επιλύθηκαν. Επιπλέον, για να είναι συμβατό το δημιουργημένο .dll της βιβλιοθήκης με το Unity project χρειάστηκε η ενεργοποίηση του CLR Support από τις ρυθμίσεις του Visual Studio. Στην πλευρά του project υπήρξαν αρκετά crashes της Unity κατά τη διάρκεια του development της βιβλιοθήκης μέχρι να κατασταλάξουν οι τύποι μεταφοράς των δεδομένων και να γίνει η κατάλληλη μετατροπή τους. Τέλος, ένα πρόβλημα που παρά λίγο να είναι παράγοντας αλλαγής του βασικού serializer ήταν πως εσωτερικά το MsgPack header file έκανε χρήση των std::min και std::max μεθόδων. Αυτό επιλύθηκε με τη χρήση του “NOMINMAX” macro μέσα στο αρχείο “pch.h” όπως διακρίνεται στην Εικόνα 42.

# Κεφάλαιο 5: Δοκιμές

## 5.1 Εισαγωγή κεφαλαίου

Στο παρών κεφάλαιο, αναλύεται η διαδικασία της συλλογής των δεδομένων προς ανάλυση στο κεφάλαιο 6 σε συνεργασία με τις πληροφορίες συστήματος.

## 5.2 Δεδομένα προς συλλογή

Ο σκοπός των παρακάτω δοκιμών είναι να συλλεχθούν δεδομένα σχετικά με το μέγεθος των παραγόμενων τελικών serialized αρχείων και το μέγεθος του Garbage Collection κατά το saving. Οι δοκιμές θα γίνουν, για το μέγεθος των αρχείων, αρχικά σε ένα ISnapshotModel (minimum), μετά σε δέκα (medium), μετά σε εκατό (normal scenario), μετά σε χίλια (realistic scenario) και τέλος δέκα χιλιάδες (unrealistic scenario). Οι μετρήσεις για το Garbage Collection Size, θα γίνουν στις βαθμίδες 1 έως 10. Για τις μετρήσεις θα χρησιμοποιηθεί ο profiler της Unity. Οι δοκιμές θα γίνουν επάνω το μοντέλο της κλάσης “Weapon” ονόματη “SWeapon”.

**Καταχώρηση 54:** **ProposedArchitecture – ISnapshotModel usage**

```
 1. namespace ProposedArchitecture {
 2.
 3.     [MessagePackObject]
 4.     public struct SWeapon : ISnapshotModel {
 5.         [Key(0)]
 6.         public uint Smri { get; set; }
 7.         [Key(1)]
 8.         public int[] RefSmris { get; set; }
 9.         [Key(2)]
10.         public int _Ammo;
11.         [Key(3)]
12.         public bool _Loaded;
13.         [Key(4)]
14.         public Vector3 _Position;
15.         [Key(5)]
16.         public Quaternion _Rotation;
17.     }
18. }
19.
```

## 5.3 Χαρακτηριστικά συστήματος δοκιμών

- Motherboard: Gigabyte AB350-Gaming 3-CF
- CPU: AMD Ryzen 7 1700 Eight-Core Processor 3892.8 MHz
- RAM: 32GB DDR4

- Storage Device: Adata XPG SX8200 Pro SSD 1TB M.2 NVMe PCI Express 3.0

## 5.4 Software δοκιμών

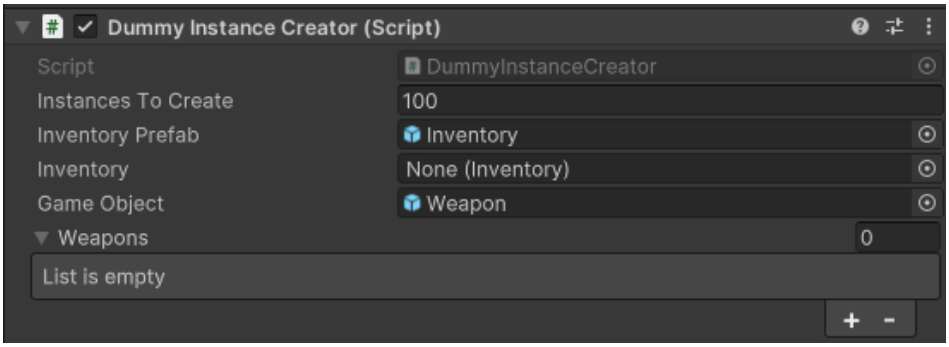
- Edition: Windows 10 Pro x64
- Version: 22H2
- Installed on: 16-Apr-23
- OS build: 19045.4412
- Experience: Windows Feature Experience Pack 1000.19056.1000.0
- Unity Version: 2022.3.26f1

## 5.5 Πειραματική διαδικασία

Δημιουργήθηκε η σκηνή στη Unity ονόματη “Benchmarking” στην οποία το script “DummyInstanceCreator” κατά την ενεργοποίηση του δημιουργούσε N αριθμό από Weapon instances.

**Καταχώρηση 55:** **DummyInstanceCreator class**

```
1. public class DummyInstanceCreator : MonoBehaviour {
2.     public int _InstancesToCreate;
3.     public GameObject _InventoryPrefab;
4.     public Inventory _Inventory;
5.     public GameObject _GameObject;
6.     public List<Weapon> _Weapons;
7.
8.     void Start() {
9.         _InventoryPrefab = Instantiate(_InventoryPrefab);
10.        _Inventory = _InventoryPrefab.GetComponent<Inventory>();
11.        GameObject temp = null;
12.        for (int i = 0; i < _InstancesToCreate; i++) {
13.            temp = Instantiate(_GameObject);
14.            temp.GetComponent<Weapon>().RegisterToSaveManager();
15.            _Weapons.Add(temp.GetComponent<Weapon>());
16.            _Weapons[i].SetInventory(_Inventory);
17.        }
18.    }
19. }
20.
```




*Εικόνα 17: Benchmarking script UI*

Η κάμερα της σκηνής ήταν απενεργοποιημένη ώστε να βγούνε σωστά τα δεδομένα. Η διαδικασία του saving έγινε κανονικά μέσω του SaveManager όπως σε ένα ρεαλιστικό περιβάλλον παιχνιδιού.

## 5.4 Προβλήματα κατά την πειραματική διαδικασία

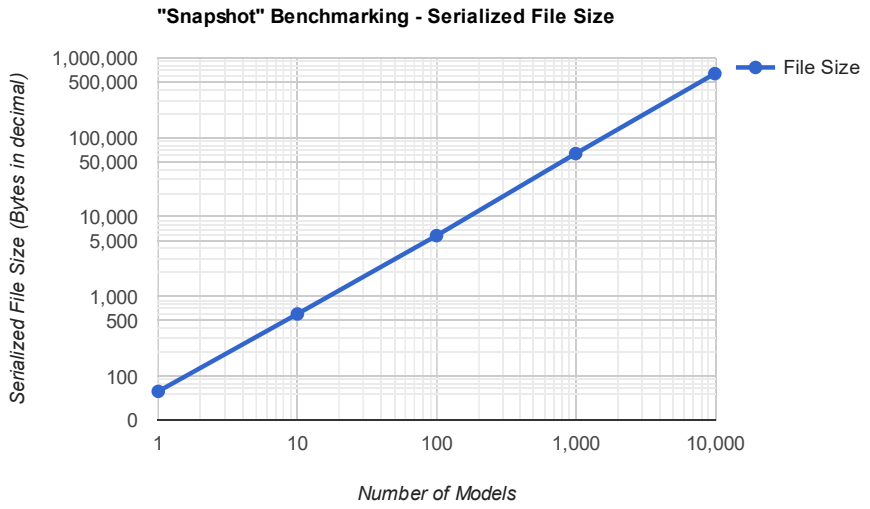
Κατά την πειραματική διαδικασία δεν υπήρξε κάποιο πρόβλημα ούτε κάποιο λειτουργικό σφάλμα.

# Κεφάλαιο 6: Αποτελέσματα

## 6.1 Εισαγωγή κεφαλαίου

Στο παρών κεφάλαιο αναλύονται τα αποτελέσματα από το στάδιο δοκιμών ώστε να υπάρχει μία αρχική βάση για τα μεγέθη των αρχείων αλλά και τα μεγέθη του garbage collection ανά πειραματική βαθμίδα.

## 6.2 Ανάλυση μεγέθους αρχείου

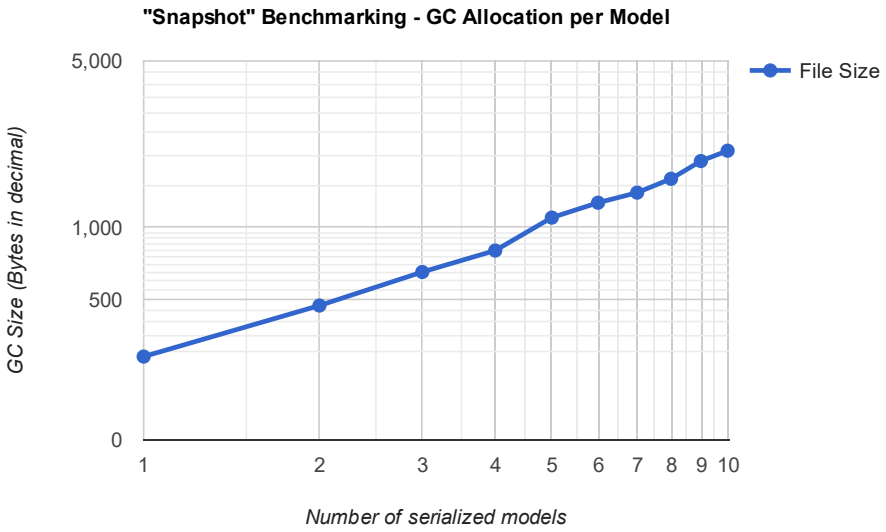



*Εικόνα 18: Μέγεθος αρχείου στα στάδια δοκιμών*

Με βάση τις πληροφορίες που συλλέχθηκαν κατά την πειραματική διαδικασία μπορεί να παρατηρηθεί πως τα δεδομένα αποθηκεύονται σωστά, χωρίς κάποιο memory leak και πως υπάρχει γραμμική αύξηση του μεγέθους των αρχείων ανά βαθμίδα. Συγκεκριμένα στο ένα μοντέλο το αρχείο έχει μέγεθος 64bytes, στα δέκα έχει μέγεθος 604bytes, στα εκατό έχει μέγεθος 5.860bytes, στα χίλια 63.300bytes και τέλος στα δέκα χιλιάδες έχει μέγεθος 643.000bytes.

## 6.3 Ανάλυση Garbage Collection size κατά την αποθήκευση

Βάσει του παρακάτω γραφήματος στην Εικόνα 76, θεωρήθηκε σκόπιμο να προταθεί μια συνάρτηση που θα μπορεί να προβλέψει το μέγεθος Garbage Collection (GC Size) για κάθε αριθμό μοντέλων (Number of Models). Η διαδικασία που ακολουθήθηκε είναι η εξής:




*Εικόνα 19: Garbage collection size στα στάδια δοκιμών*

| Number of models (int) | GC Size (Bytes in decimal) |
|---|---|
| 1 | 287 |
| 2 | 470 |
| 3 | ~650 |
| 4 | ~800 |
| 5 | ~1100 |
| 6 | ~1270 |
| 7 | ~1400 |
| 8 | ~1600 |
| 9 | ~1900 |
| 10 | ~2100 |

*Εικόνα 20: Δειγματοληψία γραφήματος*

Αρχικά, συλλέχθηκε ένας αριθμός πειραματικών δεδομένων, λαμβάνοντας έναν συγκεκριμένο τύπο μοντέλου (SWeapon) και καταγράφοντας, για διαφορετικό αριθμό αντιγράφων, τον αντίστοιχο αριθμό Bytes GC. Μετά την ανάλυση των αποτελεσμάτων, παρατηρήθηκε ότι το μέγεθος του GC αυξάνεται γραμμικά με τον αριθμό των μοντέλων. Επομένως, γίνεται να υποθέσουμε ότι η συνάρτησή μας θα έχει την εξής μορφή [1] [2]:

$$\boldsymbol{y = ax + b}$$

*σχέση (1)*

Χρησιμοποιώντας ορισμένα από τα πειραματικά δεδομένα, μπορούμε να διαμορφώσουμε το ακόλουθο σύστημα εξισώσεων:

$$287 = a * 1 + b$$

$$470 = a * 2 + b$$

Καταλήγουμε, συνεπώς, στην εξής σχέση:

$$a = \frac{470-287}{2-1} =>$$
$$a = \frac{183}{1} =>$$
$$a = 183$$

Διεξάγοντας έλεγχο με ένα άλλο ζεύγος σημείων:

$$470 = a * 2 + b$$
$$653 = a * 3 + b$$

$$a = \frac{653-470}{3-2} =>$$
$$a = \frac{183}{1} =>$$
$$\boldsymbol{a = 183}$$

Αντικαθιστώντας στη *σχέση (1)*, έχουμε:

$$287 = 183 * 1 + b =>$$
$$\boldsymbol{b = 104}$$

Επομένως, η συνάρτηση που περιγράφει τη σχέση αριθμό μοντέλων συναρτήσει του GC Size είναι η:

$$\boldsymbol{y = 183x + 104}$$

[1] Τα αρχικά μεγέθη είναι συγκεκριμένα για το ISnapshotModel που χρησιμοποιήθηκε για τις μετρήσεις.
[2] Υπάρχει φυσικά, ένα μικρό performance overhead του καλέσματος των μεθόδων και της χρήσης του FFI, εξού και τα προσεγγιστικά μεγέθη (~).

# Κεφάλαιο 7: Συμπεράσματα

## 7.1 Ποσοστό πραγματοποίησης των στόχων

@TODO

## 7.2 Περιγραφή μελλοντικών βελτιώσεων και επεκτάσεων

@TODO

## 7.3 Επίλογος

@TODO

# Παραρτήματα

### Παράρτημα Α: Καταχωρήσεις

@TODO

# Βιβλιογραφικές αναφορές


Adler, D. (2012) 'Foreign library interface', *R Journal*, 4(1). Διαθέσιμο στο: https://doi.org/10.32614/rj-2012-004.

Aihkisalo, T. and Paaso, T. (2011) 'A Performance Comparison of Web Service Object Marshalling and Unmarshalling Solutions', in *2011 IEEE World Congress on Services*. IEEE, pp. 122–129. Διαθέσιμο στο: https://doi.org/10.1109/SERVICES.2011.61.

AL-Jumaili, A.H.A. *et al.* (2023) 'Big Data Analytics Using Cloud Computing Based Frameworks for Power Management Systems: Status, Constraints, and Future Recommendations', *Sensors*, 23(6), p. 2952. Διαθέσιμο στο: https://doi.org/10.3390/s23062952.

Andrade, H. *et al.* (2015) 'Systematic evaluation of three data marshalling approaches for distributed software systems', in *Proceedings of the Workshop on Domain-Specific Modeling*. New York, NY, USA: ACM, pp. 71–76. Διαθέσιμο στο: https://doi.org/10.1145/2846696.2846705.

Attardi, G., Flagella, T. and Iglio, P. (1998) 'A customisable memory management framework for C++', *Software: Practice and Experience*, 28(11), pp. 1143–1184. Διαθέσιμο στο: https://doi.org/10.1002/(SICI)1097-024X(199809)28:11<1143::AID-SPE194>3.0.CO;2-7.

Beazley, D. (2019) 'Simplified Wrapper and Interface Generator', *swig.org* [Preprint]. Διαθέσιμο στο: https://www.swig.org/exec.html (Προσπελάστηκε: 3 April 2024).

Beazley, D.M. (2003) 'Automated scientific software scripting with SWIG', *Future Generation Computer Systems*, 19(5 SPEC). Διαθέσιμο στο: https://doi.org/10.1016/S0167-739X(02)00171-1.

Ben-Kiki, O., Evans, C. and Ingerson, B. (2009) 'YAML Ain't Markup Language (YAML™) Version 1.2', *Language* [Preprint]. Διαθέσιμο στο: https://yaml.org/spec/1.2.2/ (Προσπελάστηκε: 5 April 2024).

Bhatti, N. *et al.* (2005) 'Object Serialization and Deserialization Using XML', *Work* [Preprint]. Διαθέσιμο στο: https://www.researchgate.net/publication/46276571_Object_Serialization_and_Deserialization_Using_XML (Προσπελάστηκε: 3 April 2024).

Bormann, C. and Hoffman, P.E. (2015) *Concise Binary Object Representation (CBOR), Internet Engineering Task Force, IETF*. Διαθέσιμο στο: https://www.rfc-editor.org/rfc/rfc8949.html (Προσπελάστηκε: 3 April 2024).

Brahmia, Z., Hamrouni, H. and Bouaziz, R. (2020) 'XML data manipulation in conventional and temporal XML databases: A survey', *Computer Science Review*. Διαθέσιμο στο: https://doi.org/10.1016/j.cosrev.2020.100231.

Carrera, D., Rosales, J. and A., G. (2018) 'Optimizing Binary Serialization with an Independent Data Definition Format', *International Journal of Computer Applications*, 180(28), pp. 15–18. Διαθέσιμο στο: https://doi.org/10.5120/ijca2018916670.

Casey, A.M. (2022) *Performance of Serialization Libraries in a High Performance Computing Environment*. Διαθέσιμο στο: https://uh-ir.tdl.org/items/3adfe624-69db-45f2-bc2a-4319c28c5f7e (Προσπελάστηκε: 8 April 2024).

Challa, S. and Laksberg, A. (2002) 'Managed Wrappers for Native Types', in *Essential Guide to Managed Extensions for C++*. Berkeley, CA: Apress, pp. 279–295. Διαθέσιμο στο: https://doi.org/10.1007/978-1-4302-0834-1_20.

Chawla, R. (2013) 'Object Serialization Formats and Techniques a Review', *Global Journal of Computer Science and Technology Software & Data Engineering*, 13(6), pp. 5–8. Διαθέσιμο στο: https://globaljournals.org/GJCST_Volume13/7-Object-Serialization-Formats.pdf (Προσπελάστηκε: 8 June 2014).

Chen, X. *et al.* (2018) 'UMFS: An efficient user-space file system for non-volatile memory', *Journal of Systems Architecture*, 89. Διαθέσιμο στο: https://doi.org/10.1016/j.sysarc.2018.04.004.

Ching, T. and Eddelbuettel, D. (2019) 'RcppMsgPack: MessagePack Headers and Interface Functions for R', *The R Journal*, 10(2), p. 516. Διαθέσιμο στο: https://doi.org/10.32614/RJ-2018-068.

Chisnall, D. *et al.* (2017) 'CHERI JNI: Sinking the Java security model into the C', *ACM SIGPLAN Notices*, 52(4). Διαθέσιμο στο: https://doi.org/10.1145/3037697.3037725.

Chistikov, D. *et al.* (2022) 'The Big-O Problem', *Logical Methods in Computer Science*, Volume 18, Issue 1(1). Διαθέσιμο στο: https://doi.org/10.46298/lmcs-18(1:40)2022.

Crestani, M. (2008) 'Foreign-Function Interfaces for Garbage-Collected Programming Languages', *www-pu.informatik.uni-tuebingen.de* [Preprint]. Διαθέσιμο στο: https://www.semanticscholar.org/paper/Foreign-Function-Interfaces-for-Garbage-Collected-Crestani/278bd70d0d53a4f512e8519d30e99e0acfebd3ec (Προσπελάστηκε: 3 April 2024).

Dai, T. *et al.* (2014) 'Understanding complex binary loading behaviors', in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*. Διαθέσιμο στο: https://doi.org/10.1109/ICECCS.2014.15.

'Data structures and algorithms using C#' (2008) *Choice Reviews Online*, 45(10), pp. 45-5625-45–5625. Διαθέσιμο στο: https://doi.org/10.5860/CHOICE.45-5625.

David, F.M., Carlyle, J.C. and Campbell, R.H. (2007) 'Context switch overheads for Linux on ARM platforms', in *Proceedings of the 2007 workshop on Experimental computer science*. New York, NY, USA: ACM, p. 3. Διαθέσιμο στο: https://doi.org/10.1145/1281700.1281703.
Dessokey, M. *et al.* (2022) 'Importance of Memory Management Layer in Big Data Architecture', *International Journal of Advanced Computer Science and Applications*, 13(5). Διαθέσιμο στο: https://doi.org/10.14569/IJACSA.2022.0130554.

Eddelbuettel, D. and François, R. (2011) 'Rcpp: Seamless R and C++ integration', *Journal of Statistical Software*, 40(8). Διαθέσιμο στο: https://doi.org/10.18637/jss.v040.i08.

Ekblad, A. (2015) 'Foreign exchange at low, low rates: A lightweight FFI for web-targeting Haskell dialects', in *ACM International Conference Proceeding Series*. Διαθέσιμο στο: https://doi.org/10.1145/2897336.2897338.

Eriksson, M. and Hallberg, V. (2011) 'Comparison between JSON and YAML for data serialization', *The School of Computer Science and Engineering Royal Institute of Technology* [Preprint]. Διαθέσιμο στο: https://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/rapport/eriksson_malin_OCH_hallberg_victor_K11047.pdf (Προσπελάστηκε: 3 April 2024).

Ezra Tsur, E. (2018) 'Delivering the fundamentals of software architecture, design and abstraction by developing a ray tracer for 3-dimensional graphical scenes', *Computer Applications in Engineering Education*, 26(6). Διαθέσιμο στο: https://doi.org/10.1002/cae.21963.

Gallardo, C., Pogrebnoy, A. and Varela-Aldás, J. (2021) 'Development and Use of Dynamic Link Libraries Generated Under Various Calling Conventions', in, pp. 220–232. Διαθέσιμο στο: https://doi.org/10.1007/978-3-030-68285-9_22.

Giaimo, F. *et al.* (2015) 'Improving bandwidth efficiency with self-Adaptation for data marshalling on the example of a self-driving miniature car', in *ACM International Conference Proceeding Series*. Διαθέσιμο στο: https://doi.org/10.1145/2797433.2797454.

Gladstone, A. (2022) 'C# Clients: Consuming the Managed Wrapper', in *C++ Software Interoperability for Windows Programmers*. Berkeley, CA: Apress, pp. 67–88. Διαθέσιμο στο: https://doi.org/10.1007/978-1-4842-7966-3_4.

Grimmer, M. *et al.* (2018) 'Cross-Language Interoperability in a Multi-Language Runtime', *ACM Transactions on Programming Languages and Systems*, 40(2), pp. 1–43. Διαθέσιμο στο: https://doi.org/10.1145/3201898.

El Hajj, I. *et al.* (2016) 'SpaceJMP', *ACM SIGARCH Computer Architecture News*, 44(2), pp. 353–368. Διαθέσιμο στο: https://doi.org/10.1145/2980024.2872366.

Hall, J.G. and Rapanotti, L. (2017) 'A design theory for software engineering', *Information and Software Technology*, 87, pp. 46–61. Διαθέσιμο στο: https://doi.org/10.1016/j.infsof.2017.01.010.

Hardman, C. (2020) *Game Programming with Unity and C#, Game Programming with Unity and C: A Complete Beginner's Guide*. Berkeley, CA: Apress. Διαθέσιμο στο: https://doi.org/10.1007/978-1-4842-5656-5.

Harris, S.L. and Harris, D.M. (2016) *Digital Design and Computer Architecture*, *Digital Design and Computer Architecture: ARM Edition*. Elsevier. Διαθέσιμο στο: https://doi.org/10.1016/C2013-0-14352-8.

He, Y. *et al.* (2022) 'JNI Global References Are Still Vulnerable: Attacks and Defenses', *IEEE Transactions on Dependable and Secure Computing*, 19(1). Διαθέσιμο στο: https://doi.org/10.1109/TDSC.2020.2995542.

Hericko, M. *et al.* (2003) 'Object serialization analysis and comparison in Java and .NET', *ACM SIGPLAN Notices*, 38(8). Διαθέσιμο στο: https://doi.org/10.1145/944579.944589.

International, E. (2009) 'ECMA-262 ECMAScript Language Specification', *JavaScript Specification*, 16(June). Διαθέσιμο στο: https://ecma-international.org/publications-and-standards/standards/ecma-262/ (Προσπελάστηκε: 4 April 2024).

Jansen, A. and Bosch, J. (2005) 'Software architecture as a set of architectural design decisions', in *Proceedings - 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005*. Διαθέσιμο στο: https://doi.org/10.1109/WICSA.2005.61.

Jeong, J. and Son, Y. (2018) 'A dynamically linked library based indirect call function analysis for detecting banned API usage in binary code', *International Journal of Grid and Distributed Computing*, 11(3). Διαθέσιμο στο: https://doi.org/10.14257/ijgdc.2018.11.3.07.

Kernighan, B.W. and Ritchie, D.M. (2015) 'The C Programming Language: The C Programming Language', *TI The effect of two different electronic health record user interfaces on intensive care provider task load* [Preprint]. Διαθέσιμο στο: http://cslabcms.nju.edu.cn/problem_solving/images/c/cc/The_C_Programming_Language_%282nd_Edition_Ritchie_Kernighan%29.pdf (Προσπελάστηκε: 3 April 2024).

Kloss, G.K. (2008) ‘Automatic C Library Wrapping Ctypes from the Trenches’, *The Python Papers*, 3(3). Διαθέσιμο στο: https://mro.massey.ac.nz/handle/10179/4503 (Προσπελάστηκε: 3 April 2024).

Krishna Madasu, V., Venkata Swamy Naidu Venna, T. and Eltaeib, T. (2015) ‘SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP’, *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, 2(2). Διαθέσιμο στο: https://www.researchgate.net/publication/273451390_SOLID_Principles_in_Software_Architecture_and_Introduction_to_RESM_Concept_in_OOP (Προσπελάστηκε: 3 April 2024).

Kulshreshtha, K. *et al.* (2018) ‘Efficient computation of derivatives for solving optimization problems in R and Python using SWIG-generated interfaces to ADOL-C†’, *Optimization Methods and Software*, 33(4–6). Διαθέσιμο στο: https://doi.org/10.1080/10556788.2018.1425861.
Laaksonen, A. (2017) ‘A new book on competitive programming’, in *Olympiads in Informatics*. Διαθέσιμο στο: https://doi.org/10.15388/ioi.2017.14.

Lang, D.T. (2001) ‘Using {XML} for Statistics: The {XML} Package’, *R News*, 1(1). Διαθέσιμο στο: https://journal.r-project.org/articles/RN-2001-009/RN-2001-009.pdf (Προσπελάστηκε: 3 April 2024).

Laurie A. Schintler and Connie L. McNeely (2022) ‘Big O Notation’, in *Encyclopedia of Big Data*. Cham: Springer International Publishing, pp. 109–109. Διαθέσιμο στο: https://doi.org/10.1007/978-3-319-32010-6_300022.

Li, C., Ding, C. and Shen, K. (2007) ‘Quantifying the cost of context switch’, *Proceedings of the 2007 Workshop on Experimental Computer Science* [Preprint]. Διαθέσιμο στο: https://doi.org/10.1145/1281700.1281702.

Li, S. and Tan, G. (2014) ‘Exception analysis in the Java Native Interface’, *Science of Computer Programming*, 89(PART C), pp. 273–297. Διαθέσιμο στο: https://doi.org/10.1016/j.scico.2014.01.018.

Li, Z. *et al.* (2022) ‘Detecting Cross-language Memory Management Issues in Rust’, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 680–700. Διαθέσιμο στο: https://doi.org/10.1007/978-3-031-17143-7_33.

Lin, S.Y. *et al.* (2018) ‘Modeling Interactions in Community Resilience’, in *Structures Congress 2018: Blast, Impact Loading, and Response; and Research and Education - Selected Papers from the Structures Congress 2018*. Διαθέσιμο στο: https://doi.org/10.1061/9780784481349.001.

Liu, S., Tan, G. and Jaeger, T. (2017) ‘PtrSplit: Supporting general pointers in automatic program partitioning’, in *Proceedings of the ACM Conference on Computer and Communications Security*. Διαθέσιμο στο: https://doi.org/10.1145/3133956.3134066.

Madhavapeddy, A. and Minsky, Y. (2022) ‘Foreign Function Interface’, in *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press, pp. 405–423. Διαθέσιμο στο: https://doi.org/10.1017/9781009129220.027.

Maeda, K. (2012) ‘Performance evaluation of object serialization libraries in XML, JSON and binary formats’, in *2012 Second International Conference on Digital Information and Communication Technology and it’s Applications (DICTAP)*. IEEE, pp. 177–182. Διαθέσιμο στο: https://doi.org/10.1109/DICTAP.2012.6215346.

Mejia Alvarez, P., Leon Ayala, M. and Ortega Cisneros, S. (2022) ‘Main Memory Management on Relational Database Systems’, in *SpringerBriefs in Computer Science*. Διαθέσιμο στο: https://doi.org/10.1007/978-3-031-13295-7_2.

Microsoft Corporation (2012) ‘XAML Overview (WPF)’, *Msdn* [Preprint]. Διαθέσιμο στο: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-8.0 (Προσπελάστηκε: 3 April 2024).

Ogala, J., Ogala, B. and Onyarin, J. (2020) ‘Comparative Analysis of C, C++, C# and JAVA Programming Languages’, *Global Scientific Journals*, 8(5). Διαθέσιμο στο: https://www.academia.edu/43343591/COMPARATIVE_ANALYSIS_OF_C_C_C_AND_JAVA_PROGRAMMING_LANGUAGES (Προσπελάστηκε: 3 April 2024).

Oktafiani, I. and Hendradjaya, B. (2018) ‘Software Metrics Proposal for Conformity Checking of Class Diagram to SOLID Design Principles’, in *Proceedings of 2018 5th International Conference on Data and Software Engineering, ICoDSE 2018*. Διαθέσιμο στο: https://doi.org/10.1109/ICODSE.2018.8705857.

Otero, C.E. (2016) *Software Engineering Design: Theory and Practice*, *Software Engineering Design: Theory and Practice*. Διαθέσιμο στο: https://www.routledge.com/Software-Engineering-Design-Theory-and-Practice/Otero/p/book/9781439851685 (Προσπελάστηκε: 3 April 2024).

Ou, P. and Demsky, B. (2017) ‘Checking Concurrent Data Structures under the C/C++11 Memory Model’, *ACM SIGPLAN Notices*, 52(8). Διαθέσιμο στο: https://doi.org/10.1145/3018743.3018749.

Palensky, P. *et al.* (2017) ‘Cosimulation of Intelligent Power Systems: Fundamentals, Software Architecture, Numerics, and Coupling’, *IEEE Industrial Electronics Magazine*. Διαθέσιμο στο: https://doi.org/10.1109/MIE.2016.2639825.

Pardede, E., Rahayu, J.W. and Taniar, D. (2008) ‘XML data update management in XML-enabled database’, *Journal of Computer and System Sciences*, 74(2). Διαθέσιμο στο: https://doi.org/10.1016/j.jcss.2007.04.008.
Park, J. *et al.* (2023) ‘Static Analysis of JNI Programs via Binary Decompilation’, *IEEE Transactions on Software Engineering*, 49(5), pp. 3089–3105. Διαθέσιμο στο: https://doi.org/10.1109/TSE.2023.3241639.

Parker, A. (2018) *Algorithms and Data Structures in C++*, *Algorithms and Data Structures in C++*. Routledge. Διαθέσιμο στο: https://doi.org/10.1201/9781315137148.

De Paula, G.C. and Ierusalimschy, R. (2022) 'A Foreign Function Interface for Pallene', in *ACM International Conference Proceeding Series*. Διαθέσιμο στο: https://doi.org/10.1145/3561320.3561321.

Perez-Schofield, B.G. *et al.* (2019) 'Learning memory management with C-Sim: A C-based visual tool', *Computer Applications in Engineering Education*, 27(5). Διαθέσιμο στο: https://doi.org/10.1002/cae.22147.

Plauger, P.J. (2002) 'The C/C++ programming language', *C/C++ Users Journal*, 20(10). Διαθέσιμο στο: https://www.thriftbooks.com/w/standard-c-library-the_pj-plauger/260744/#edition=2342887&idiq=4011290 (Προσπελάστηκε: 3 April 2024).

Plauska, I., Liutkevičius, A. and Janavičiūtė, A. (2022) 'Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller', *Electronics*, 12(1), p. 143. Διαθέσιμο στο: https://doi.org/10.3390/electronics12010143.

Popic, S. *et al.* (2016) 'Performance evaluation of using Protocol Buffers in the Internet of Things communication', in *2016 International Conference on Smart Systems and Technologies (SST)*. IEEE, pp. 261–265. Διαθέσιμο στο: https://doi.org/10.1109/SST.2016.7765670.

Powers, B. *et al.* (2019) 'Mesh: Compacting memory management for C/C++ applications', in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Διαθέσιμο στο: https://doi.org/10.1145/3314221.3314582.

Qureshi, S. (2004) 'C#, COM objects, & interop service', *Dr. Dobb's Journal*. Διαθέσιμο στο: https://www.researchgate.net/publication/291668425_C_COM_objects_interop_service (Προσπελάστηκε: 3 April 2024).

Ramadhan Omar, N. *et al.* (2021) 'Enhancing OS Memory Management Performance: A Review', *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, 3(12). Διαθέσιμο στο: https://www.researchgate.net/publication/351969783_Enhancing_OS_Memory_Management_Performance_A_Review (Προσπελάστηκε: 3 April 2024).

Reddy, M. (2011) 'Chapter 11 - Scripting', in *API Design for C++*. Διαθέσιμο στο: https://www.oreilly.com/library/view/api-design-for/9780123850034/ (Προσπελάστηκε: 3 April 2024).

Robert C. Martin (2009) 'Clean Code: A Handbook of Agile Software Craftsmanship', *Kybernetes*, 38(6). Διαθέσιμο στο: https://doi.org/10.1108/03684920910973252.

Sakr, S. (2009) 'XML compression techniques: A survey and comparison', *Journal of Computer and System Sciences*, 75(5). Διαθέσιμο στο: https://doi.org/10.1016/j.jcss.2009.01.004.

Senoo, E.E.K. *et al.* (2022) 'Implementing SOLID principles for IoT Arduino sensor code', in *2022 10th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC)*. IEEE, pp. 21–26. Διαθέσιμο στο: https://doi.org/10.1109/JAC-ECC56395.2022.10043950.

Shah, H.B., Görg, C. and Harrold, M.J. (2010) 'Understanding exception handling: Viewpoints of novices and experts', *IEEE Transactions on Software Engineering*, 36(2). Διαθέσιμο στο: https://doi.org/10.1109/TSE.2010.7.

da Silva, P.D.S., Campos, R.O. and Rocha, C. (2021) 'OSS Scripting System for Game Development in Rust', in *IFIP Advances in Information and Communication Technology*, pp. 51–58. Διαθέσιμο στο: https://doi.org/10.1007/978-3-030-75251-4_5.

Simec, A. and Maglicic, M. (2014) 'Comparison of JSON and XML Data Formats', *Central European Conference on Information and Intelligent Systems*, (September 2014), pp. 272–275. Διαθέσιμο στο: https://www.researchgate.net/publication/329707959.

Sirgany, E. El (2019) 'Testing Benefits of SOLID Principles', *PNSQC Proceedings* [Preprint]. Διαθέσιμο στο: http://uploads.pnsqc.org/2019/papers/El-Singany-Testing-Benefits-of-SOLID-Principles-.pdf (Προσπελάστηκε: 3 April 2024).

Sotiriadis, S. *et al.* (2018) 'Evaluating the Java Native Interface (JNI): Data types and strings', *International Journal of Distributed Systems and Technologies*, 9(2). Διαθέσιμο στο: https://doi.org/10.4018/IJDST.2018040103.

Soukup, J. and Macháček, P. (2014) *Serialization and Persistent Objects*, *Serialization and Persistent Objects*. Διαθέσιμο στο: https://doi.org/10.1007/978-3-642-39323-5.

de Sousa, D.B.C. *et al.* (2020) 'Studying the evolution of exception handling anti-patterns in a long-lived large-scale project', *Journal of the Brazilian Computer Society*, 26(1). Διαθέσιμο στο: https://doi.org/10.1186/s13173-019-0095-5.

TC39 (2017) *ECMA-404 The JSON Data Interchange Standard.*, *ECMA*. Διαθέσιμο στο: https://ecma-international.org/publications-and-standards/standards/ecma-404/ (Προσπελάστηκε: 4 April 2024).

Uzayr, S. bin (2022) 'Memory Management in C++', in *Mastering C++ Programming Language*. Boca Raton: CRC Press, pp. 257–292. Διαθέσιμο στο: https://doi.org/10.1201/9781003214762-5.

Vanura, J. and Kriz, P. (2018) 'Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats', in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 166–175. Διαθέσιμο στο: https://doi.org/10.1007/978-3-319-94376-3_11.

Wu, Y., Yap, R.H.C. and Ramnath, R. (2010) 'Comprehending module dependencies and sharing', in *Proceedings - International Conference on Software Engineering*. Διαθέσιμο στο: https://doi.org/10.1145/1810295.1810309.

Yallop, J., Sheets, D. and Madhavapeddy, A. (2018) ‘A modular foreign function interface’, *Science of Computer Programming*, 164. Διαθέσιμο στο: https://doi.org/10.1016/j.scico.2017.04.002.

Younan, Y. *et al.* (2010) ‘Improving Memory Management Security for C and C++’, *International Journal of Secure Software Engineering*, 1(2). Διαθέσιμο στο: https://doi.org/10.4018/jsse.2010040104.

Zalewski, J. (2001) ‘Real-time software architectures and design patterns: Fundamental concepts and their consequences’, *Annual Reviews in Control*, 25. Διαθέσιμο στο: https://doi.org/10.1016/S1367-5788(01)80001-8.

Zinovyev, A. and Mirkes, E. (2013) ‘Data complexity measured by principal graphs’, *Computers & Mathematics with Applications*, 65(10), pp. 1471–1482. Διαθέσιμο στο: https://doi.org/10.1016/j.camwa.2012.12.009.

Zunke, S. and Souza, V.D.’ (2014) ‘JSON vs XML: A Comparative Performance Analysis of Data Exchange Formats’, *IJCSN International Journal of Computer Science and Network*, 3(4), pp. 257–261. Διαθέσιμο στο: https://ijcsn.org/IJCSN-2014/3-4/JSON-vs-XML-A-Comparative-Performance-Analysis-of-Data-Exchange-Formats.pdf (Προσπελάστηκε: 3 April 2024).