UNIVERSITY OF PATRAS - SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

Division: Electronics and Computers
Lab: Interactive Technologies Laboratory

Diploma Thesis

of the Department of Electrical and Computer Engineering student of the school of engineering of the University of Patras

Evangelos Lamprou

registration number: 1066519

Title

Design, Implementation, and Evaluation of a Framework for Applying Answer Set Programming in Games

Supervisor

Associate Professor Christos Fidas, University of Patras

Patras, June 2023

CERTIFICATION

It is certified that the Diploma Thesis titled

Design, Implementation, and Evaluation of a Framework for Applying Answer Set Programming in Games

of the Department of Electrical and Computer Engineering student

Evangelos Lamprou

(Registration Number: 1066519)

was presented publicly at the Department of Electrical and Computer Engineering at

5/7/2023

The Supervisor

Christos Fidas
Associate Professor

The Director of the Division

Grigorios Kalivas
Professor

Details of Diploma Thesis

Title: **Design, Implementation, and Evaluation of a Framework for Applying Answer Set Programming in Games**

Student: **Evangelos Lamprou**

Examining commitee
Associate Professor, Christos Fidas,
University of Patras

Professor, Pavlos Peppas,
University of Patras

Teaching and Research Staff, Christos Sintoris,
University of Patras

Labs
Interactive Technologies Laboratory

Period of thesis completion:
March 2022 - June 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ - ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας: Τομέας Ηλεκτρονικής και Υπολογιστών
Εργαστήριο: Interactive Technologies Laboratory

Διπλωματική Εργασία

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών της Πολυτεχνικής Σχολής του Πανεπιστημίου Πατρών

Ευάγγελου Λάμπρου του Νικολάου

αριθμός μητρώου: 1066519

Θέμα

Σχεδιασμός, Υλοποίηση και Αξιολόγηση Πλαισίου για την Εφαρμογή του Προγραμματισμού Συνόλου Απαντήσεων σε Παιχνίδια

Design, Implementation, and Evaluation of a Framework for Applying Answer Set Programming in Games

Επιβλέπων

Αναπληρωτής Καθηγητής Χρήστος Φείδας, Πανεπιστήμιο Πατρών

Πάτρα, Ιούνιος 2023

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η διπλωματική εργασία με θέμα

Σχεδιασμός, Υλοποίηση και Αξιολόγηση Πλαισίου για την Εφαρμογή του Προγραμματισμού Συνόλου Απαντήσεων σε Παιχνίδια

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

Ευάγγελου Λάμπρου του Νικολάου

(Α.Μ.: 1066519)

παρουσιάτηκε δημόσια και εξετάστηκε στο τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις

5/7/2023

Ο Επιβλέπων

Χρήστος Φείδας
Αναπληρωτής Καθηγητής

Ο Διευθυντής του Τομέα

Καλύβας Γρηγόριος
Καθηγητής

Στοιχεία διπλωματικής εργασίας

Θέμα: **Σχεδιασμός, Υλοποίηση και Αξιολόγηση Πλαισίου για την Εφαρμογή του Προγραμματισμού Συνόλου Απαντήσεων σε Παιχνίδια**

Φοιτητής: **Ευάγγελος Λάμπρου του Νικολάου**

Ομάδα επίβλεψης
Αναπληρωτής Καθηγητής Χρήστος Φείδας,
Πανεπιστήμιο Πατρών

Καθηγητής Παύλος Πέππας,
Πανεπιστήμιο Πατρών

Ε.ΔΙ.Π Χρήστος Σιντόρης,
Πανεπιστήμιο Πατρών

Εργαστήρια
Interactive Technologies Laboratory

Περίοδος εκπόνησης της εργασίας:
Μάρτιος 2022 - Ιούνιος 2023

Περίληψη

Η βιομηχανία των βιντεοπαιχνιδιών εξελίσσεται συνεχώς, με νέους τρόπους δημιουργίας παιχνιδιών να αναπτύσσονται. Ωστόσο, ακόμα και με τη διαθεσιμότητα ισχυρών μηχανών δημιουργίας παιχνιδιών (game engines), οι προγραμματιστές εξακολουθούν να χρειάζονται χρόνο και προσπάθεια για να υλοποιήσουν κοινά χαρακτηριστικά παιχνιδιών, όπως βασική τεχνητή νοημοσύνη, αλγορίθμους εύρεσης διαδρομής και απλές παραλλαγές σκηνών. Αυτό μπορεί να δυσκολέψει τους δημιουργούς παιχνιδιών στο να επικεντρωθούν στον πυρήνα του παιχνιδιού και το περιεχόμενό του, ειδικά αν δεν είναι έμπειροι προγραμματιστές ή δεν ενδιαφέρονται για την ανάπτυξη πολύπλοκων αλγορίθμων. Επιπλέον, η υλοποίηση τέτοιων χαρακτηριστικών συνήθως συνδέεται στενά με τη λογική και την αρχιτεκτονική του παιχνιδιού, καθιστώντας δυσχερή την επαναχρησιμοποίηση του κώδικα.

Η εργασία αυτή προτείνει μια προσέγγιση στην ανάπτυξη παιχνιδιών όπου τμήματα της λογικής του παιχνιδιού γράφονται σε μια δηλωτική γλώσσα προγραμματισμού. Δεν απορρίπτουμε τη χρήση διαδικαστικού προγραμματισμού, αλλά προτείνουμε εργαλεία που μπορούν να αποτελέσουν μέρος και να ενισχύσουν τη διαδικασία ανάπτυξης παιχνιδιών. Συγκεκριμένα, αυτή η έρευνα επικεντρώνεται στην εφαρμογή του Answer Set Programming (ASP), μιας δηλωτικής προγραμματιστικής παραδοχής, στην ανάπτυξη βιντεοπαιχνιδιών. Η μελέτη δείχνει πώς μπορεί να χρησιμοποιηθεί το ASP ως μέρος της διαδικασίας ανάπτυξης παιχνιδιών, παρέχοντας νέες δυνατότητες σχεδιασμού και υλοποίησης.

Επισημαίνουμε την προσαρμοστικότητα του ASP σε μια ευρεία γκάμα προβλημάτων και αναφερόμαστε στην ικανότητα του επιλυτή ASP να παράγει λύσεις σε λογικό χρονικό διάστημα. Επιπλέον, παρουσιάζονται βοηθητικά εργαλεία που μπορούν να ενισχύσουν την εμπειρία προγραμματισμού με ASP.

Παρουσιάζεται ένα πλαίσιο (framework) που ενσωματώνει έναν επιλυτή ASP σε μηχανή παιχνιδιών. Μέσα σε αυτήν, ο σχεδιαστής μπορεί να καθορίσει τους κανόνες που διέπουν ένα συγκεκριμένο μηχανισμό παιχνιδιού, όπως η τοποθέτηση αντικειμένων ή η συμπεριφορά των χαρακτήρων. Ο επιλυτής παράγει στη συνέχεια τις αντίστοιχες λύσεις οι οποίες ερμηνεύονται από τη μηχανή παιχνιδιού. Το πλαίσιο έχει σχεδιαστεί για να είναι αρθρωτό, απλό και επεκτάσιμο.

Τέλος, παρουσιάζονται ορισμένες εφαρμογές του ASP σε πλαίσιο παιχνιδιών και αξιολογείται η εφικτότητα και αποτελεσματικότητά της προτεινόμενης μεθόδου εκτελώντας μία εμπειρική μελέτη με χρήστες.

Extended Abstract in Greek

Εισαγωγή

Η βιομηχανία παιχνιδιών διευρύνεται και εξελίσσεται διαρκώς, με νέους τρόπους ανάπτυξης παιχνιδιών να αναπτύσσονται. Οι μηχανές παιχνιδιών προσφέρουν πληθώρα εργαλείων και χαρακτηριστικών που μπορούν να βοηθήσουν τους σχεδιαστές παιχνιδιών να δώσουν ζωή στις ιδέες τους [1]. Ωστόσο, στις περισσότερες μηχανές ανάπτυξης παιχνιδιών, οι προγραμματιστές παιχνιδιών αντιμετωπίζουν εντολές προστακτικού προγραμματισμού. Ο παραδοσιακός προστακτικός προγραμματισμός είναι μια προγραμματιστική παραδοχή που καθορίζει εντολές βήμα προς βήμα για τον υπολογιστή να εκτελέσει χρησιμοποιώντας μεταβλητές, βρόχους και συνθήκες.

Ο δηλωτικός προγραμματισμός επικεντρώνεται στο τι πρέπει να επιτύχει ένα πρόγραμμα, χρησιμοποιώντας υψηλού επιπέδου αφαιρέσεις για να ορίσει τον τομέα του προβλήματος, επιτρέποντας στον υπολογιστή να φτάσει αυτόματα στη λύση.

Ο Προγραμματισμός Συνόλου Απαντήσεων (Answer Set Programming - ASP) [31] είναι μια δηλωτική παραδοχή προγραμματισμού που έχει επιδείξει ικανότητα στην επίλυση πολύπλοκων προβλημάτων σε διάφορους τομείς, όπως η ανάθεση υπαλλήλων σε ομάδες [87], νομικά συμπεράσματα [5], η επίλυση σφαλμάτων διαμόρφωσης πακέτων λογισμικού [38] και η αυτόματη σύνθεση μουσικής [9].

Δηλωτικές τεχνικές έχουν ήδη εφαρμοστεί σε πλαίσιο παιχνιδιών [2], με παραδείγματα που κυμαίνονται από εμπορικό λογισμικό όπως το παιχνίδι *F.E.A.R* [82] και το *Halo 3* [55] μέχρι πιο ερευνητικές προσεγγίσεις, όπως η *Γλώσσα Περιγραφής Παιχνιδιών (GDL)* [89] και η μηχανή *Ludocore* [99], όπου σημασιολογίες παιχνιδιών κωδικοποιούνται μέσα σε δηλωτικά λογικά πλαίσια. Τώρα, σχετικά με αυτή την εργασία, το ASP συγκεκριμένα έχει δει εφαρμογές σε παιχνίδια. Έχουν αναπτυχθεί πράκτορες που μπορούν να επιλύσουν γρίφους [33, 103, 114] ή να παίξουν παιχνίδια όπως το *Angry Birds* [18]. Ο εργασίες [4] (δημιουργία πιστών για το παιχνίδι *Portal 2*) και [99] (κατασκευή λαβυρίνθων) εξετάζουν την εφαρμογή του ASP για τη δημιουργία περιεχομένου, επικεντρώνοντας στη δημιουργία puzzle levels ενώ ταυτόχρονα υπογραμμίζεται πώς το ASP μπορεί να λειτουργήσει ως ένα εκφραστικό εργαλείο για τη δημιουργία παραγωγών περιεχομένου παιχνιδιών με χρονικά αποδοτικό τρόπο.

Αυτή η εργασία αναπτύσσει ένα πλαίσιο για την ανάπτυξη παιχνιδιών αξιοποιώντας την παραδοχή ASP και παρουσιάζει την προστιθέμενη αξία του στο πλαίσιο συγκεκριμένων περιπτώσεων χρήσης στον προγραμματισμό παιχνιδιών. Συγκεκριμένα, παρουσιάζουμε αποτελέσματα αξιολόγησης που υπογραμμίζουν ότι η χρήση δηλωτικών εργαλείων μπορεί να επιταχύνει σημαντικά τον χρόνο

ανάπτυξης και να απαλλάξει τους προγραμματιστές παιχνιδιών από την ανάγκη κατανόησης και υλοποίησης πολύπλοκων αλγορίθμων, οδηγώντας σε πιο ευέλικτο και επαναχρησιμοποιήσιμο κώδικα, ενώ ταυτόχρονα επιτρέπει μια διαφορετική δημιουργική προσέγγιση στην ανάπτυξη παιχνιδιών.

Κίνητρο και Συνεισφορά

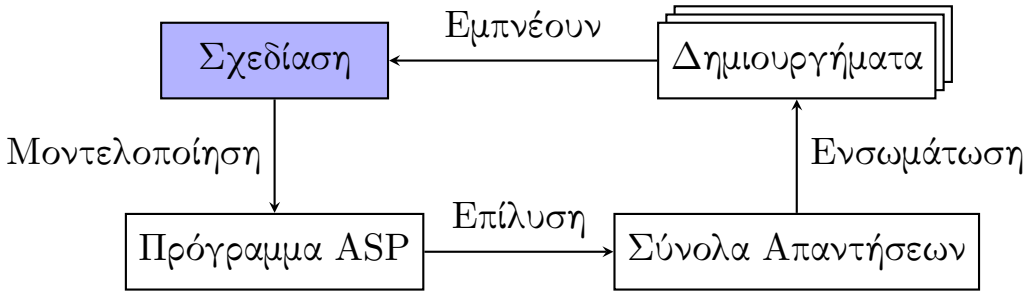
Σε μικρές ομάδες ανάπτυξης παιχνιδιών (1-5 άτομα), ο ρόλος του προγραμματιστή και του σχεδιαστή συχνά συγχωνεύονται. Αυτό σημαίνει ότι ο σχεδιαστής του παιχνιδιού συχνά πρέπει να διακόψει τη δημιουργική διαδικασία της τελειοποίησης και δοκιμής μιας ιδέας για να υλοποιήσει πολύπλοκη λογική. Υπάρχει ακόμα έλλειψη εργαλείων κατάλληλων για την ταχεία πρωτότυπη ανάπτυξη. Ως λύση, προτείνουμε ένα πλαίσιο για τη χρήση του ASP στα παιχνίδια, σχολιάζουμε πτυχές της ανάπτυξης παιχνιδιών που είναι κατάλληλες για υλοποίηση με ASP και παρουσιάζουμε μερικές μελέτες περιπτώσεων. Επίσης, πραγματοποιούμε αξιολόγηση του ASP με προγραμματιστές σε διάφορα επίπεδα εξοικείωσης με το παράδειγμα προγραμματισμού. Όσο γνωρίζουμε, μια μελέτη χρηστών για τα πλεονεκτήματα της χρήσης του ASP για τη δημιουργία παιχνιδιών δεν έχει πραγματοποιηθεί προηγουμένως.

Η εργασία είναι οργανωμένη ως εξής: Πρώτα παρουσιάζουμε τις βασικές γνώσεις για την κατανόηση της δηλωτικής παραδοχής προγραμματισμού ASP. Στη συνέχεια, παρουσιάζουμε το προτεινόμενο πλαίσιο που μπορεί να χρησιμοποιηθεί από προγραμματιστές παιχνιδιών και, τέλος, παρουσιάζουμε τα αποτελέσματα της μελέτης αξιολόγησης.

Θεωρία Προγραμματισμού Συνόλων Απαντήσεων

Ο Προγραμματισμούς Συνόλων Απαντήσεων (Answer Set Programming - ASP) [31] είναι ένα παράδειγμα επίλυσης προβλημάτων με ρίζες στο λογικό προγραμματισμό και τον μη-μονοτονικό συλλογισμό. Η εργασία του Gelfond [42] διατύπωσε για πρώτη φορά τη σημασιολογία των σταθερών μοντέλων και τον πυρήνα της γλώσσας ASP. Όπως φαίνεται στο σχήμα 1, το προγραμματιστικό μοντέλο του ASP είναι ένα όπου ο προγραμματιστής μοντελοποιεί το πεδίο του προβλήματος, με τη λύση να αναλαμβάνεται από ένα πρόγραμμα επίλυσης. Η προγραμματιστική διαδικασία γίνεται σε μια οικογένεια γλωσσών συχνά αποκαλούμενες ως *AnsProlog* [41]. Στην εργασία μας, θα χρησιμοποιήσουμε τη γλώσσα εισόδου του *Clingo* [37], ενός συστήματος το οποίο περιλαμβάνει έναν αποδοτικό επιλυτή με μια πλούσια συλλογή βιβλιοθηκών που βοηθούν στην ένταξή του με εξωτερικά εργαλεία.

Η σύνταξη είναι παρόμοια με αυτήν της *Prolog*, μιας δημοφιλούς γλώσσας λογικού προγραμματισμού. Υπάρχει μια ενοποιημένη προσέγγιση για να αναπα-



Σχήμα 1: Ροή ανάπτυξης παιχνιδιού με τη βοήθεια των εργαλείων ASP. Ο σχεδιαστής ξεκινά με έναν αρχικό στόχο, τον σχεδιασμό ενός μηχανισμού/συμπεριφοράς/συνόλου τεχνουργημάτων, που οδηγεί σε μια προδιαγραφή σε μορφή ενός προγράμματος ASP. Οι λύσεις του προγράμματος μπορούν να βοηθήσουν στην περαιτέρω βελτίωση του αρχικού σχεδιασμού, καθώς ανεπιθύμητες ή απούσες πτυχές γίνονται εμφανείς μετά την ένταξη των δημιουργημάτων με το υπόλοιπο παιχνίδι [98].

ραστήσει ο προγραμματιστής τόσο γνωστικές όσο και δεδομένου τύπου γνώσεις μέσω λογικών όρων. Οι όροι αυτοί μπορούν να είναι ατομικά στοιχεία όπως ονομασμένα σύμβολα, αριθμοί, συμβολοσειρές ή σύνθετα στοιχεία που αποτελούνται από έναν φορέα (ένα σύμβολο) και μια λίστα λογικών όρων ως ορίσματα. Χρησιμοποιώντας συλλογές λογικών όρων (καταχώρηση 1), μπορεί να αναπαρασταθεί εύκολα οποιαδήποτε δομή δεδομένων που σχετίζεται με την κατάσταση ενός παιχνιδιού.

Καταχώρηση 1: Ένα σύνολο γεγονότων που περιγράφουν στοιχεία του παιχνιδιού/κατάσταση του παιχνιδιού.

```
object(house).
position(player, vec3(1, 0, 1)).
size(house, vec3(4, 4, 4)).
tile(1, 1, water).
move(player, left).
object(orc).
object(frog).
```

Καταχώρηση 2: Λογικοί κανόνες που περιγράφουν τις σχέσεις μεταξύ οντοτήτων και τη συμπεριφορά τους.

```
damaged(player) :- attacked(player).
damaged(player) :- fall(player).
hostile(X) :- enemy(X).
friend(X) :- object(X), not hostile(X).
pos(player,X+1,Y,T+1) :- pos(player,X,Y,T),
                         move(player,right).
```

Στο παράδειγμα της καταχώρισης 1, παρουσιάζεται ένα σύνολο γεγονότων που περιγράφουν την κατάσταση του παιχνιδιού. Για πιο περίπλοκη συλλογιστική, ο συγγραφέας του ASP προγράμματος μπορεί να προσθέσει λογικούς κανόνες που μπορούν να εκφραστούν χρησιμοποιώντας τον τελεστή :-. Το αριστερό μέρος ενός κανόνα ονομάζεται "κεφάλι" (head) και το δεξί μέρος "σώμα" (body). Το κεφάλι ενός κανόνα είναι αληθές αν το σώμα του είναι αληθές. Εντός ενός κανόνα, τα κόμματα μεταξύ ατομικών στοιχείων υποδηλώνουν τη λογική λειτουργία "και" (and), ενώ η επανάληψη του ίδιου κεφαλαίου κανόνα με διαφορετικά σώματα υποδηλώνει τη λογική λειτουργία "ή" (or). Οι κανόνες μπορούν να χρησιμοποιηθούν για να ορίσουν τις επιπτώσεις ενεργειών ή για να εξάγουν ιδιότητες αντικειμένων του παιχνιδιού. Στους κανόνες, ένα ατομικό στοιχείο

που αρχίζει με κεφαλαίο γράμμα υποδηλώνει μια μεταβλητή. Στο παράδειγμα της καταχώρισης 2, παρουσιάζονται λογικοί κανόνες που περιγράφουν τις σχέσεις μεταξύ των οντοτήτων και τη συμπεριφορά τους.

Οι δυνατότητες επιλογής/δημιουργίας του ASP προέρχονται από τη δυνατότητα του συγγραφέα του προγράμματος να επιτρέπει στον επιλυτή ASP να πραγματοποιεί επιλογές μεταξύ ενός συνόλου ατομικών στοιχείων. Αυτά μπορούν να κωδικοποιηθούν χρησιμοποιώντας "κανόνες επιλογής".

Καταχώρηση 3: Ένας κανόνας επιλογής.

```
{chosen(X,Y) : person(X)} :-
    house(Y).
```

Καταχώρηση 4: Ένας κανόνας ακεραιότητας.

```
:- chosen(X, Y), chosen(Z, Y),
     X == Y.
```

Καταχώρηση 5: Οδηγία βελτιστοποίησης.

```
#minimize{C : cost(E,C)}.
```

Ο κανόνας επιλογής στην καταχώρηση 3 μεταφράζεται ως "για κάθε σπίτι Y, επέλεξε ανάμεσα στο σύνολο ατομικών στοιχείων $chosen(X, Y)$, όπου X είναι ένα πρόσωπο". Αυτό το πρόγραμμα μπορεί να παράγει πολλά σύνολα απαντήσεων, μία για κάθε πιθανή ανάθεση μεταβλητής. Ωστόσο, μερικές από τις δημιουργηθείσες απαντήσεις είναι μη έγκυρες στο πλαίσιο του προβλήματός μας. Για παράδειγμα, δε θα πρέπει να επιτρέπεται δύο άνθρωποι να έχουν επιλέξει το ίδιο σπίτι. Τέτοιοι κανόνες μπορούν να κωδικοποιηθούν με τη μορφή "περιορισμών ακεραιότητας" (integrity constraints) (καταχώρηση 4), οι οποίοι υποδεικνύουν τι δεν επιτρέπεται να ισχύει στα παραγόμενα σύνολα απαντήσεων. Αυτό παρέχει ένα μηχανισμό για το "φιλτράρισμα" μη-επιθυμητών απαντήσεων. Σε προβλήματα όπου μπορεί να υπάρχουν πολλές έγκυρες απαντήσεις, μπορούμε επίσης να προσθέσουμε οδηγίες βελτιστοποίησης που καθοδηγούν τον επιλυτή να εξάγει βέλτιστες απαντήσεις με βάση συγκεκριμένες μεταβλητές. Ο συγγραφέας του προγράμματος μπορεί να χρησιμοποιήσει τις οδηγίες `#minimize` και `#maximize` (καταχώρηση 5).

Για τον υπολογισμό των συνόλων απαντήσεων, τα προγράμματα ASP εισάγονται σε επίλυτες ASP. Αυτοί οι επίλυτες παρέχουν υψηλής απόδοσης μηχανισμούς για την παραγωγή του συνόλου των έγκυρων απαντήσεων για το συγκεκριμένο πρόβλημα. Ένας επίλυτης ASP μπορεί να θεωρηθεί ως ένα μαύρο κουτί, με τους επιλυτές να είναι αντικαταστάσιμοι, υπό την προϋπόθεση ότι οι σημασιολογία της γλώσσας εισόδου παραμένει ίδια.

Ευρετικές Εφαρμοσιμότητας και Μέθοδος για την Εφαρμογή του ASP στη Διαδικασία Ανάπτυξης Παιχνιδιών

Ένα σημαντικό στοιχείο του προτεινόμενου πλαισίου είναι ο καθορισμός συγκεκριμένων ευρετικών σχεδιασμού παιχνιδιών που αποδεικνύουν την καταλληλότητα των στοιχείων παιχνιδιού για προγραμματιστικές προσεγγίσεις με ASP. Προτείνουμε τις ακόλουθες αρχές/ευρετικές εφαρμογής:

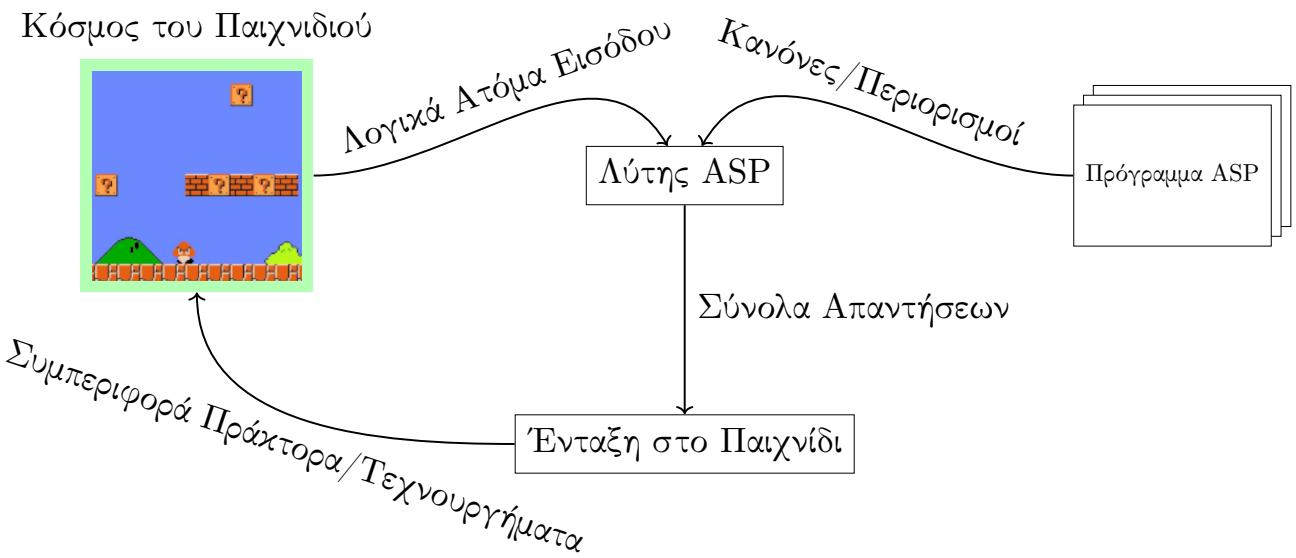
- ***Ευρετική Εφαρμογής ASP (Α)* Συντομία**: Ο ASP (και ο δηλωτικός προγραμματισμός γενικότερα) μπορεί να μειώσει την πολυπλοκότητα του λογισμικού [100], οδηγώντας σε πιο συνοπτικό κώδικα [12]. Ωστόσο, αυτό απαιτεί ότι κατά τον σχεδιασμό ενός μηχανισμού παιχνιδιού, κωδικοποιούνται μόνο τα σημαντικά του στοιχεία. Για παράδειγμα, σε ένα παιχνίδι λαβύρινθου θα περιλαμβάνονται μόνο τα ουσιώδη στοιχεία όπως η διάταξη του λαβυρίνθου, η αρχική θέση, η θέση του θησαυρού και οι κανόνες κίνησης.

- ***Ευρετική Εφαρμογής ASP (Β)* Σχετικά Μικρός Χώρος Λύσεων**: Πρέπει να αποφεύγονται περιπτώσεις όπου ο χρόνος επίλυσης γίνεται πολύ μεγάλος. Ένας μικρός χώρος λύσεων προκύπτει όταν η γεννήτρια (generator)/πράκτορας (agent) έχει έναν περιορισμένο αριθμό επιλογών για κάθε κανόνα επιλογής του προγράμματος ASP. Για παράδειγμα, ένας σχεδιαστής θα πρέπει να επιλέξει έναν πράκτορα να κινηθεί σε μία από τις τέσσερις κατευθύνσεις (πάνω, κάτω, αριστερά, δεξιά), αντί για το πλήρες εύρος κινήσεων. Έπειτα, μπορεί να χρησιμοποιηθεί μια φυσική προσομοίωση μέσω της μηχανής παιχνιδιού για να γίνει πιο φυσισκό-φανής η κίνηση.

  Οι σχεδιαστές μπορούν επίσης να αντιμετωπίσουν αυτόν τον περιορισμό διαμερίζοντας το πρόβλημα σε μικρότερα υπο-προβλήματα. Στην εργασία [19], οι συγγραφείς διαχώρισαν τη ρουτίνα δημιουργία της τοπολογίας ενός μπουντρουμιού από την τοποθέτηση περιεχομένου στο κάθε δωματίου, μειώνοντας έτσι τους χρόνους δημιουργίας, ενώ στο [86] οι διάφορες καταστάσεις ενός πράκτορα (τρώει, κρύβεται, δρα) διαιρέθηκαν σε μικρότερες μονάδες προγραμμάτων ASP, χρησιμοποιώντας μετα-σκέψη για να αποφασίζεται ποια σχετικά μέρη της βάσης γνώσης θα χρησιμοποιηθούν για την επίλυση την κάθε χρονική στιγμή.

- ***Ευρετική Εφαρμογής ASP (Γ)* Αναδυόμενη Πολυπλοκότητα**: Καταλληλότερα είναι σενάρια όπου παρατηρούνται ενδιαφέρουσες συμπεριφορές όταν οι πράκτορες αλληλεπιδρούν μεταξύ τους και με το περιβάλλον εντός του κόσμου του παιχνιδιού ή όταν τα παραγόμενα αντικείμενα εκδηλώνουν ενδιαφέροντα μοτίβα που δεν έχουν κωδικοποιηθεί ρητά στο πρόγραμμα ASP. Στο [82], όπου προστέθηκε ένα στρώμα δηλωτικού σχεδιασμού στους πράκτορες του παιχνιδιού *F.E.A.R*, προέκυψαν πολύπλοκες συμπεριφορές από τον συνδυασμό απλών στόχων και ενεργειών μαζί με τη δυναμική κατάσταση του κόσμου του παιχνιδιού.

Επιπλέον, προτείνουμε μια τυποποιημένη μεθοδολογία ανάπτυξης (σχήμα 2) που θα καθοδηγήσει φιλόδοξους προγραμματιστές στο να εφαρμόσουν με επιτυχία το ASP στις εφαρμογές τους, παρέχοντας γενικές κατευθυντήριες γραμμές προγραμματισμού που σχετίζονται με το μοντελοποίηση ASP. Βασιζόμαστε στο παράδειγμα "μάντεψε και έλεγξε" [34].



Σχήμα 2: Συνολική επισκόπηση της ένταξης ASP στη δημιουργία ενός μηχανισμού παιχνιδιού. Ο *Κόσμος του Παιχνιδιού* περιλαμβάνει την τρέχουσα κατάσταση του παιχνιδιού και τις πληροφορίες όλων των οντοτήτων μέσα σε αυτόν. Τα λογικά άτομα εισόδου είναι τα δεδομένα που χρησιμοποιούνται για να περιγράψουν την τρέχουσα κατάσταση. Αυτά, μαζί με τους κανόνες και τους περιορισμούς του προγράμματος ASP, τροφοδοτούνται στον ASP επιλυτή, ο οποίος εξάγει σύνολα απαντήσεων. Αυτά περιγράφουν λογική όπως οι ενέργειες που θα πρέπει να πάρει ένας πράκτορας ή τη θέση πού θα πρέπει να τοποθετηθεί ένα αντικείμενο. Έπειτα, μέσω μηχανισμών ένταξης, αυτά τα σύνολα απαντήσεων χρησιμοποιούνται για να ενημερώσουν τον κόσμο του παιχνιδιού.

1. **Βήμα (α): Καθορισμός Ατόμων Εισόδου και Εξόδου**: Το σύνολο των ατόμων εισόδου παρέχει το πλαίσιο που απαιτείται για να παράξει το ASP πρόγραμμα σωστά αποτελέσματα. Αυτά είναι συνήθως δυναμικές πτυχές της εκτέλεσης του παιχνιδιού και αλλάζουν κατά την κάθε κλήση του λύτη ASP. Παραδείγματα αυτών είναι η αρχική θέση ενός πράκτορα ή η λίστα των αντικειμένων που πρέπει να τοποθετηθούν. Από την άλλη πλευρά, τα άτομα εξόδου κωδικοποιούν τα αποτελέσματα που παράγονται από τον λύτη και που θα ερμηνευθούν από την εκτέλεση του παιχνιδιού ως τεχνάσματα ή συμπεριφορά πράκτορα. Αυτά περιλαμβάνουν πράγματα όπως η κατεύθυνση στην οποία θα κινηθεί ένας πράκτορας στο επόμενο βήμα ή η θέση στην οποία θα πρέπει να τοποθετηθεί ένα αντικείμενο.

2. **Βήμα (β): Δημιουργία "Τυχαίων" Συνόλων Απαντήσεων**: Ο προγραμματιστής μπορεί να κατασκευάσει αρχικά ένα πρόγραμμα ASP που αποτελείται από κανόνες επιλογής για τη δημιουργία μερικώς τυχαίων αποτελεσμάτων, με βάση τα επιλαχόντα άτομα εξόδου. Παρόλο που τα παραγόμενα αποτελέσματα μπορεί να είναι ατελή ή ακατάλληλα, αυτή η προσέγγιση διευκολύνει τον εντοπισμό και την αντιμετώπιση δυνητικών

τεχνικών προβλημάτων. Σε αυτό το στάδιο περιλαμβάνεται επίσης η δημιουργία ενός οπτικοποιητή ή η ένταξη του λύτη με το παιχνίδι εκκινώντας έτσι τη διαδικασία αποσφαλμάτωσης. Προτείνεται μια αρχιτεκτονική λογισμικού για την ενσωμάτωση του ASP σε μηχανή παιχνιδιού στο [3].

3. **Βήμα (γ): Προσθήκη Περιορισμών Ακεραιότητας/Οδηγιών Βελτιστοποίησης**: Με βάση τον τρέχοντα τομέα του προβλήματος, είναι απαραίτητο να προστεθούν περιορισμοί ακεραιότητας και/ή οδηγίες βελτιστοποίησης. Οι περιορισμοί παρέχουν άμεσο έλεγχο στα παραγόμενα σύνολα απαντήσεων για να συμμορφώνονται τόσο με το σύνολο κανόνων του παιχνιδιού όσο και με τις ιδέες του σχεδιαστή. Ανάμεσά τους, αν χρειάζεται, ο λύτης μπορεί να παράγει τις πλέον βέλτιστες λύσεις βάσει μιας μεταβλητής χρησιμοποιώντας κανόνες βελτιστοποίησης.

Απόδειξη Έννοιας και Μελέτες Περίπτωσης

Για να επιδείξουμε τις προαναφερθείσες ευρετικές της εφαρμογής του ASP στην ανάπτυξη παιχνιδιών, σχεδιάσαμε μια σειρά από μελέτες περίπτωσης[1].

Μελέτη Περίπτωσης (α): Δημιουργία Φυσικού Εδάφους σε Επίπεδο Πλακιδίων

Για τη δημιουργία εδάφους, τα παιχνίδια συνήθως ακολουθούν μια προσέγγιση όπου μια συνάρτηση θορύβου όπως ο θόρυβος Perlin [84] χρησιμοποιείται για να καθορίσει τον τύπο του τοπίου που θα τοποθετηθεί σε μια συγκεκριμένη θέση (x, y). Αυτή η προσέγγιση έχει γνωρίσει μεγάλη υιοθέτηση στη βιομηχανία των παιχνιδιών καθώς χρησιμοποιείται σε μεγάλους τίτλους όπως το *Minecraft* [77]. Ωστόσο, αυτή η προσέγγιση, αν και αποδοτική, δεν επιτρέπει υψηλά επίπεδα ελέγχου. Για παράδειγμα, ένας σχεδιαστής δεν μπορεί να καθορίσει ότι θέλει να τοποθετηθεί ένας συγκεκριμένος αριθμός βουνών ή ότι ένα ποτάμι ρέει μέσα από μια συγκεκριμένη περιοχή. Χρησιμοποιώντας το ASP, μπορούμε να δημιουργήσουμε φυσικό-εμφανές τοπίο χρησιμοποιώντας μια υψηλής έκφρασης γλώσσα. Οι ευρετικές **συντομία** και **αναδυόμενη πολυπλοκότητα**, είναι παρούσες σε αυτό το παράδειγμα. Στο ASP πρόγραμμά μας, η τοποθέτηση των πλακιδίων εντός του πλέγματος μπορεί να κωδικοποιηθεί από έναν κανόνα επιλογής. Έπειτα, μέσω των περιορισμών ακεραιότητας, προσθέτουμε κανόνες που εξαπλώνονται σε ολόκληρο το πλέγμα, δημιουργώντας ενδιαφέροντα μοτίβα.

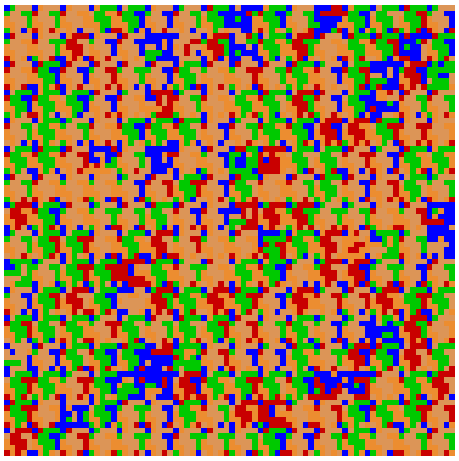
Η εφαρμογή της προτεινόμενης μεθοδολογίας μας λειτουργεί ως εξής:

[1] Ο πηγαίος κώδικας από τα παραδείγματά μας καθώς και τα έργα που αναπτύχθηκαν στη μελέτη χρηστών μπορούν να βρεθούν στο https://github.com/vagos/asp-games.

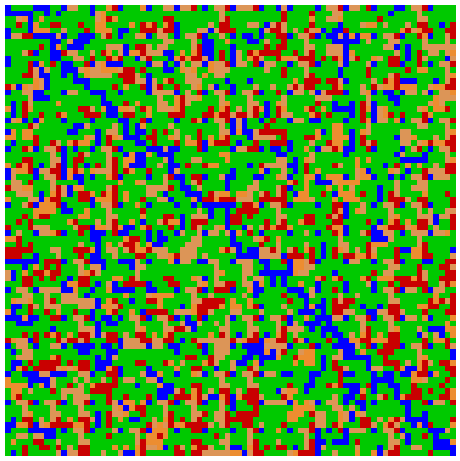
1. Καθορίζουμε τα λογικά στοιχεία εξόδου, τα οποία είναι ο τύπος του πλακιδίου σε κάθε θέση του πλέγματος (ένα άτομο της μορφής $tile(x, y, type)$). Τα ατομικά στοιχεία εισόδου αποτελούνται από προτάσεις που θα ελέγχουν συγκεκριμένες πτυχές της δημιουργίας. Για παράδειγμα, ο προγραμματιστής μπορεί να εισαγάγει ένα δεδομένο της μορφής $tile(1, 1, water)$ που θα αναγκάσει τον γεννήτορα να τοποθετήσει το συγκεκριμένο τύπο πλακιδίου σε αυτήν τη θέση.

2. Προσθέτουμε έναν κανόνα επιλογής που τοποθετεί ένα πλακίδιο τυχαίου τύπου σε κάθε θέση του πλέγματος. Σε αυτό το σημείο, αναπτύσσουμε επίσης ένα πρόγραμμα που παίρνει την έξοδο του επιλυτή και μεταφράζει τα ατομικά στοιχεία εξόδου σε χρωματισμένα εικονοστοιχεία.

3. Η έξοδος του γεννήτορα ελέγχεται με χρήση περιορισμών ακεραιότητας. Στο παράδειγμά μας, προσθέσαμε περιορισμούς όπου δεν μπορούν να αγγίζουν πλακίδια "νερό" και "λάβα" ενώ πρέπει να υπάρχει επίσης ένα ποτάμι που ρέει κατά μήκος της διαγωνίου.



(α′) Ένα τοπίο χωρίς κανόνες περιορισμού.



(β′) Ένα τοπίο στο οποίο τα πλακίδια νερού και λάβας δεν αγγίζουν ο ένας τον άλλον και υπάρχει ένας ποταμός που διασχίζει το πλέγμα.

Σχήμα 3: Παραδείγματα τοπίων που δημιουργήθηκαν από τον γεννήτορα.

Μελέτη περίπτωσης (β): Παιχνίδι Ποδοσφαίρου

Αναπτύξαμε ένα σενάριο παιχνιδιού που δείχνει πώς το ASP μπορεί να μοντελοποιήσει δύο ομάδες αντιπάλων πρακτόρων και πώς το αποτέλεσμα είναι ταυτόχρονα ενδιαφέρον και κατάλληλο. Η μέθοδός μας αποφεύγει την ανάγκη υλοποίησης αλγορίθμων εύρεσης διαδρομής όπως ο A* [94] ή η εφαρμογή τεχνικών Ενισχυτικής Μάθησης (Reinforcement Learning) [90], που μπορεί να είναι

δύσκολες στην ανάπτυξη και αποσφαλμάτωση κατά την προτυποποίηση του παιχνιδιού. Οι χαρακτηριστικές **σχετικά μικρός χώρος λύσεων** και **αναδυόμενη πολυπλοκότητα** είναι παρόντες σε αυτήν τη μελέτη περίπτωσης. Ο χώρος λύσεων μπορεί να ελεγχθεί με τη μείωση του αριθμού των χρονικών βημάτων που μπορεί να "δει" ο πράκτορας στο μέλλον. Η αναδυόμενη πολύπλοκη συμπεριφορά προέρχεται από το γεγονός ότι δεν θα ζητήσουμε ρητά από τον πράκτορα να κλωτσήσει την μπάλα για να σκοράρει γκολ. Αντίθετα, εξηγούμε μέσω λογικών κανόνων πώς μεταβάλλεται η τοποθεσία της μπάλας όταν κλωτσιέται και επιτρέπουμε στον επίλυτη του ASP να παράγει μια νικηφόρα στρατηγική βασιζόμενος σε αυτήν την πληροφορία.

1. Σε αυτήν τη μελέτη περίπτωσης, τα άτομα εισόδου είναι οι τοποθεσίες κρίσιμων αντικειμένων του παιχνιδιού, όπως η θέση της μπάλας, των δύο τερμάτων και των άλλων παικτών. Τα άτομα εξόδου θα είναι οι αποφάσεις του πράκτορα για το πού να κινηθεί, εάν θα κλωτσήσει την μπάλα και προς ποια κατεύθυνση.

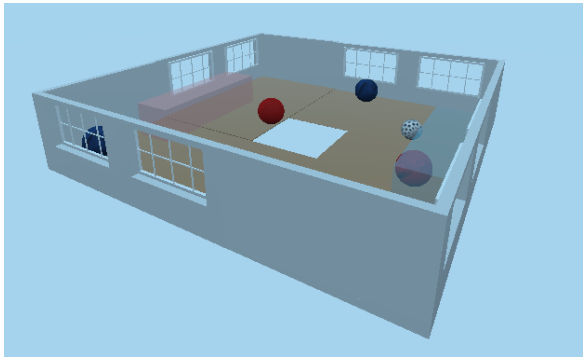
2. Προσθέτουμε κανόνες επιλογής για τις δυνατές ενέργειες του πράκτορα. Σε αυτό το σημείο, ενσωματώνουμε τον επίλυτη μέσα στη μηχανή παιχνιδιού *Godot* [47], μεταφράζοντας τα άτομα εξόδου σε ενέργειες εντός του παιχνιδιού.

3. Τέλος, προσθέτουμε περιορισμούς ακεραιότητας που κάνουν τους πράκτορες να αποφεύγουν τη σύγκρουση μεταξύ τους, καθώς και μια οδηγία βελτιστοποίησης που προσπαθεί να ελαχιστοποιήσει την απόσταση της μπάλας από το τέρμα του αντιπάλου. Η ακολουθία ενεργειών που θα ακολουθήσει ένας πράκτορας θα είναι αυτή που θα οδηγήσει την μπάλα να είναι πιο κοντά στο αντίπαλο τέρμα.

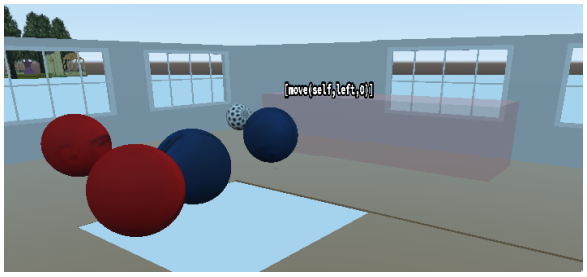
Εμπειρική μελέτη

Ερευνητικά ερωτήματα

Τα κύρια ερευνητικά ερωτήματα της εμπειρικής μελέτης ήταν να διερευνήσουμε: **EE1**) Εάν οι προαναφερθείσες ευρετικές εφαρμοσιμότητας μπορούν να επιβεβαιωθούν από τρίτους σχεδιαστές παιχνιδιών μετά από πειραματική εφαρμογή στα δικά τους σχέδια παιχνιδιών, **EE2**) εάν η προτεινόμενη μεθοδολογική προσέγγιση του ASP υποστηρίζει τη δημιουργικότητα στον σχεδιασμό παιχνιδιών και **EE3**) εάν οι τελικοί χρήστες αντιμετώπισαν δυσκολίες στην εφαρμογή της προτεινόμενης ροής εργασίας.



(α′) Ένα στιγμιότυπο οθόνης του γηπέδου.



(β′) Οι ποδοσφαιριστές. Το κείμενο που αιωρείται πάνω από τον πράκτορα υποδεικνύει την τελευταία "σκέψη" του πράκτορα, που είναι η επόμενη ενέργεια που θα εκτελέσει. Σε αυτήν την εικόνα, η πρόταση $move(self, left, 0)$ αποτελεί μέρος του βέλτιστου συνόλου απαντήσεων που παράχθηκε από τον επιλυτή ASP, πράγμα που σημαίνει ότι στο επόμενο χρονικό βήμα, ο πράκτορας θα μετακινηθεί προς τα αριστερά.

Σχήμα 4: Η υλοποίηση του παιχνιδιού ποδοσφαίρου.

Συμμετέχοντες

Συμμετείχαν συνολικά 8 συμμετέχοντες (2 γυναίκες και 6 άνδρες), οι οποίοι ήταν όλοι φοιτητές του τμήματος *Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών*. Όλοι είχαν εμπειρία προγραμματισμού με διαδικαστικές γλώσσες, με τρεις από αυτούς να έχουν εμπειρία με λογικές γλώσσες (είτε *Prolog* είτε *Clingo*). Όλοι εκτός από δύο από τους συμμετέχοντες είχαν προηγούμενη εμπειρία στην ανάπτυξη παιχνιδιών, στο πλαίσιο προσωπικών εργασιών. Οι συμμετέχοντες ενημερώθηκαν ότι δε συλλέχθηκαν προσωπικά δεδομένα πέρα από τις απαντήσεις τους στο μέρος συνέντευξης της μελέτης. Κάθε συμμετέχοντας συμμετείχε στη διαδικασία της μελέτης για διάρκεια 0,5 έως 3 ωρών, με αποτέλεσμα μια συνολική διάρκεια μελέτης περίπου είκοσι τεσσάρων ωρών.

Διαδικασία Μελέτης

Η διεξαγωγή της μελέτης ήταν σε μορφή ενός προς ένα, όπου κάθε συμμετέχοντας εργαζόταν ατομικά με τον ερευνητή. Η μελέτη χρησιμοποίησε όλα τα συλλεγμένα δεδομένα με ανώνυμο τρόπο και οι συμμετέχοντες είχαν την ελευθερία να αποσυρθούν από τη μελέτη οποιαδήποτε στιγμή της επιλογής τους.

- **Φάση Α - Εισαγωγή στο ASP.** Η μελέτη ξεκίνησε με μια σύντομη επισκόπηση της τεχνολογίας του Προγραμματισμού Συνόλου Απαντήσεων, της σύνταξης και σημασιολογίας της γλώσσας *Clingo*.

- **Φάση Β - Υλοποίηση Μηχανισμού Παιχνιδιού / Δημιουργία Περιεχομένου.** Στη συνέχεια, ζητήθηκε από τους συμμετέχοντες να σκεφτούν έναν μηχανισμό παιχνιδιού ή έναν δημιουργό περιεχομένου που θα ήθελαν να υλοποιήσουν χρησιμοποιώντας το ASP. Ενθαρρύναμε τους συμμετέχοντες να είναι δημιουργικοί και να σκεφτούν μοναδικές ή δύσκολες ιδέες. Αφού οι συμμετέχοντες είχαν μια ιδέα στο μυαλό τους, ο ερευνητής τους βοήθησε να δημιουργήσουν το λογικό πρόγραμμα για τον μηχανισμό του παιχνιδιού τους χρησιμοποιώντας ASP. Αποφεύγαμε να καθοδηγούμε τους συμμετέχοντες κατά τη διαδικασία μοντελοποίησης, όπου θα κατάρτιζαν τους λογικούς κανόνες για το πρόγραμμά τους, και περιορίζαμε την παρέμβασή μας στην επίλυση ζητημάτων που αφορούν τη σύνταξη της γλώσσας *Clingo*. Κατά τη διάρκεια της διαδικασίας δημιουργίας, ο ερευνητής ήταν διαθέσιμος για να απαντήσει σε ερωτήσεις και να παρέχει καθοδήγηση κατά τον απαιτούμενο βαθμό.

- **Φάση Γ - Συζήτηση.** Τέλος, διεξήγαμε μια ημι-δομημένη συνέντευξη για να λάβουμε ποιοτικά σχόλια και να εξάγουμε την άποψη του συμμετέχοντα σχετικά με την προτεινόμενη ροή εργασίας.

Η μελέτη παρείχε πολύτιμες πληροφορίες σχετικά με το πόσο καλά οι συμμετέχοντες μπόρεσαν να κατανοήσουν και να εφαρμόσουν το ASP για τη δημιουργία μηχανισμών παιχνιδιών, καθώς και για τα πλεονεκτήματα και τους περιορισμούς αυτής της προσέγγισης.

Ανάλυση Αποτελεσμάτων

EE1: Εφαρμοσιμότητα των προτεινόμενων ευρετικών ASP σε περιπτώσεις ανάπτυξης παιχνιδιών

Δεδομένου του αριθμού των συμμετεχόντων, η αξιολόγησή μας βασίζεται κυρίως σε μια ποιοτική παρά σε ποσοτική έρευνα. Οι συμμετέχοντες παρείχαν συγκεκριμένα αποτελέσματα για τα πλεονεκτήματα και τα μειονεκτήματα της προτεινόμενης ροής εργασίας. Επιπλέον, κατά τη διάρκεια της μελέτης, δημιουργήθηκε μια σειρά εφαρμογών. Ο μεγαλύτερος αριθμός των συμμετεχόντων επικεντρώθηκε στη δημιουργία προγραμμάτων παραγωγής περιεχομένου παρά σε μηχανισμούς συμπεριφοράς πράκτορα.

Οι συμμετέχοντες έδειξαν ικανότητα στην εφαρμογή των ευρετικών εφαρμοσιμότητας του ASP κατά τη διάρκεια της μελέτης. Οι δημιουργίες τους απέδειξαν την κατανόηση τουλάχιστον ενός από τα χαρακτηριστικά της συντομίας, του σχετικά μικρού συνόλου λύσεων και της δυνατότητας αναδυόμενης πολυπλοκότητας. Αυτές οι παρατηρήσεις επιβεβαιώνουν την πρακτικότητα και την αποτελεσματικότητα των ευρετικών στην καθοδήγηση των συμμετεχόντων προς επιτυχή υλοποίηση και αξιοποίηση του ASP στη διαδικασία ανάπτυξης παιχνιδιών.

Εφαρμογή	Περιγραφή	Χαρακτηριστικό Σχεδιασμού	Διάρκεια	Επαναλήψεις
Προσομοιωτής Κατεύθυνσης Ανέμου	Προσομοιώνει την κατεύθυνση του ανέμου σε ένα πλέγμα.	**Σ**, **M**	2,5 ώρες	3
Δημιουργός Λαφυρού	Δημιουργεί συνδυασμούς ανταμοιβής.	**Σ**, **M**	1 ώρα	2
Συνομιλητικός Πράκτορας	Προσομοιώνει μια συνομιλία.	**M**, **M**	2 ώρες	4
Πράκτορας Διάσχισης Χώρου	Πράκτορας που μπορεί να πλοηγηθεί σε έναν 2D χώρο.	**Σ**	1 ώρα	2
Δημιουργός Επιπέδων	Δημιουργεί επίπεδα παιχνιδιού.	**Σ**, **M**	2 ώρες	5
Επιλυτής *Ball Sort*	Λύνει/Δημιουργεί παραδείγματα παζλ παιχνιδιού.	**Σ**, **M**	1 ώρα	2
Επιλυτής *Futoshiki*	Λύνει/Δημιουργεί παραδείγματα παζλ παιχνιδιού.	**Σ**, **M**	0,5 ώρες	2
Δημιουργός Επιπέδων	Δημιουργεί επίπεδα με ελέγξιμη δυσκολία.	**Σ**, **M**	3 ώρες	6

Πίνακας 1: Οι εφαρμογές που δημιουργήθηκαν από τους συμμετέχοντες κατά τη διάρκεια της μελέτης και τα χαρακτηριστικά σχεδιασμού που ικανοποιούν για την καταλληλότητα του ASP. Αυτά είναι η συντομία (**Σ**), η σχετικά μικρός χώρος λύσεων (**M**), και η αναδυόμενη πολυπλοκότητα (**A**). Στη διάρκεια μετράται ο συνολικός χρόνος ανάπτυξης. Μια μεμονωμένη επανάληψη ορίζεται ως η διαδικασία επιθεώρησης των παραγόμενων συνόλων απαντήσεων του προγράμματος και την εκτέλεση ενός ή περισσότερων σημαντικών τροποποιήσεων στο πρόγραμμα.

EE2: Υποστηρίζει η προτεινόμενη μεθοδολογία ASP τη δημιουργικότητα στον σχεδιασμό παιχνιδιών;

Ρωτήσαμε τους συμμετέχοντες εάν η προτεινόμενη ροή εργασίας παρέχει έμπνευση για τον σχεδιασμό παιχνιδιών. Ένας συμμετέχοντας με εμπειρία στην ανάπτυξη παιχνιδιών ανέφερε ότι η ροή εργασίας παρέχει τη δυνατότητα να δημιουργηθούν εντελώς νέοι μηχανισμοί παιχνιδιών που αλλιώς θα παραμερίζονταν λόγω της δυσκολίας υλοποίησης τους.

Οι περισσότεροι συμμετέχοντες αναγνώρισαν ότι η κύρια αξία του ASP είναι η ικανότητά του να αντιμετωπίζει προβλήματα με συνοπτικό τρόπο. Εξέφρασαν την άποψη ότι ο ASP διευκολύνει τη συνοπτική επίλυση προβλημάτων και απλοποιεί τη διαδικασία ανάπτυξης. Επιπλέον, ένας συμμετέχοντας ανέφερε ειδικά ότι ο ASP προσφέρει πλεονεκτήματα για άπειρους προγραμματιστές, καθώς διευκολύνει την έκφραση κανόνων ως λογικών περιορισμών από άτομα χωρίς εκτεταμένη εμπειρία προγραμματισμού.

Οι περισσότεροι συμμετέχοντες, μετά την εισαγωγή στο παράδειγμα του ASP, μπόρεσαν να αναγνωρίσουν ενστικτωδώς σενάρια όπου μπορεί να εφαρμοστεί. Από τις σχεδιαστικές ευρετικές που προτείναμε, αυτή που αναφέρθηκε ως *σχετικά μικρός χώρος λύσεων* ήταν η πιο δύσκολη στην εφαρμογή και αυτή που οι συμμετέχοντες αντιλήφθηκαν ως περιοριστική για τον σχεδιασμό τους.

Αν και μερικοί συμμετέχοντες ανέφεραν ότι υπάρχει μια απότομη καμπύλη μάθησης, γενικά η μεθοδολογία ASP θεωρήθηκε ισχυρή και δημιουργική για τον σχεδιασμό παιχνιδιών. Ένα ζευγάρι σχεδιαστή/προγραμματιστή (ή ένας μόνο σχεδιαστής που μπορεί επίσης να προγραμματίζει με τη μεθοδολογία) μπορεί να περάσει γρήγορα από την ιδέα σε ένα λειτουργικό πρωτότυπο, με ελάχιστο χρόνο ανάμεσα στις επαναλήψεις.

-- *Συμμετέχοντας 1*: "Δίνει τη δυνατότητα να δημιουργήσεις εντελώς νέα game mechanics που δεν έμπαινες στον κόπο να αναπτύξεις αλλιώς λόγω της δυσκολίας του προγραμματισμού."

-- *Συμμετέχοντας 5*: "[Ένας προγραμματιστής παιχνιδιών] μπορεί να πει κάτι σαν "Α, αυτό μπορεί να κωδικοποιηθεί εύκολα χρησιμοποιώντας κανόνες". Τώρα είναι πιο εύκολο να σκεφτώ ένα μηχανισμό παιχνιδιού και να φτιάξω περιορισμούς για να τον υλοποιήσω."

Κατά τη διάρκεια της μελέτης, οι συμμετέχοντες προσθέτανε επαναληπτικά περιορισμούς. Αυτό οδήγησε σε μια διαδικασία τελειοποίησης όπου προσθέτανε σταδιακά πιο περίπλοκοι κανόνες και περιορισμοί. Τα παραγόμενα αποτελέσματα εξετάζονταν για να επιβεβαιωθεί η σωστή μοντελοποίηση και οι συμμετέχοντες συχνά προσθέτανε νέους κανόνες και περιορισμούς όταν παρατηρούσαν ανεπιθύμητα μοτίβα ή συμπεριφορές στην έξοδο. Αυτό επιβεβαιώνει εμπειρικά την εγκυρότητα του διαδικαστικού κύκλου στο fig. 1, όπου τα αποτελέσματα της μοντελοποίησης του σχεδιαστή μπορούν να τροφοδοτήσουν τη δημιουργική του διαδικασία.

EE3: Δυσκολίες στην προτεινόμενη εφαρμογή της ροής εργασίας ASP

Ρωτήσαμε τους συμμετέχοντες ποιες δυσκολίες αντιμετώπισαν κατά την εφαρμογή της προτεινόμενης ροής εργασίας. Μία κοινή δυσκολία που αντιμετώπισαν οι συμμετέχοντες ήταν η ασυνήθιστη σύνταξη της γλώσσας *Clingo*. Τη θεώρησαν παράξενη πράγμα που δυσκόλεψε τη συγγραφή και ανάγνωση των προγραμμάτων. Επιπλέον, η έλλειψη ενός εργαλείου εντοπισμού σφαλμάτων αποτέλεσε ένα σημαντικό εμπόδιο, ειδικά κατά τις αργότερες επαναλήψεις, όταν τα προγράμματα έγιναν πιο περίπλοκα με πολλούς κανόνες και περιορισμούς ακεραιότητας. Επίσης, αναγνωρίστηκαν η αργή επίλυση και οι περιορισμένες δυνατότητες κλιμάκωσης ως περιορισμοί.

Καταλήξαμε σε ένα σαφές συμπέρασμα σχετικά με τη σημασία της επένδυσης χρόνου στην ανάπτυξη μιας ολοκληρωμένης αντιστοίχισης μεταξύ των συνόλων απαντήσεων που παράγονται από τον επίλυτη ASP και της αναπαράστασής τους στο παιχνίδι. Έγινε εμφανές ότι η εξάρτηση αποκλειστικά από την επιθεώρηση της ακατέργαστης εξόδου του επιλυτή είναι επιρρεπής σε σφάλματα και μπορεί να οδηγήσει σε μια νοητικά απαιτητική εμπειρία για τους προγραμματιστές.

-- *Συμμετέχοντας 7*: "Θα ήταν ωραίο να υπάρχει μια γραφική διεπαφή που να εμφανίζει πώς ο επιλυτής φτάνει στις λύσεις."

-- *Συμμετέχοντας 8*: "[Η ροή εργασίας] θα μπορούσε να βελτιωθεί αν η Clingo είχε καλύτερη σύνταξη. Ίσως, μια αφαίρεση κατασκευασμένη από πάνω της."

Συμπεράσματα και Μελλοντική Εργασία

Ο στόχος της έρευνάς μας ήταν να εξερευνήσουμε τις ευρετικές εφαρμογής του Προγραμματισμού Συνόλου Απαντήσεων στο πλαίσιο της ανάπτυξης παιχνιδιών. Επικεντρωθήκαμε στον εντοπισμό πιθανών περιπτώσεων όπου ο ASP μπορεί να εφαρμοστεί αποτελεσματικά και διενεργήσαμε μια εμπειρική μελέτη για να αξιολογήσουμε την πρακτική αποτελεσματικότητά του.

Η προτεινόμενη διαδικασία απορρέει από την ανάγκη για αξιόπιστες διεπαφές υψηλού επιπέδου προγραμματισμού που μπορούν να βοηθήσουν στην ανάπτυξη περίπλοκων εφαρμογών όπως τα παιχνίδια. Προτείναμε μια μεθοδολογία για την αναγνώριση των μερών όπου η λογική του παιχνιδιού μπορεί να εκφραστεί με κομψότητα χρησιμοποιώντας την αναπαράσταση βασισμένη σε ASP.

Ένα ερευνητικό έργο που μπορεί να ξεπεράσει την περιορισμένη αποδοχή του ASP είναι η δημιουργία μιας γλώσσας που διατηρεί τη σημασιολογία της γλώσσας *AnsProlog* ενώ παρέχει μια πιο φιλική προς τον προγραμματιστή σύνταξη και δομή. Η μελλοντική εργασία θα πρέπει να περιλαμβάνει την εφαρμογή του ASP σε μεγαλύτερα έργα ανάπτυξης παιχνιδιών, εξερευνώντας πώς η προτεινόμενη ροή εργασίας μπορεί να ενσωματωθεί σε μακροχρόνια έργα ανάπτυξης παιχνιδιών.

Περιορισμοί

Εδώ, αναφερόμαστε στους περιορισμούς της έρευνάς μας. Βεβαίως, ένας περιορισμός της μελέτης μας είναι ότι το προφίλ των συμμετεχόντων περιορίστηκε σε φοιτητές αντί για έμπειρους προγραμματιστές παιχνιδιών. Επιπλέον, ο αριθμός των συμμετεχόντων ήταν σχετικά μικρός, με αποτέλεσμα να βασιζόμαστε σε ποιοτική ανάλυση για τα ευρήματα της έρευνας.

Abstract

The gaming industry is continuously growing and evolving, with new ways of creating games being developed. However, even with the availability of powerful game engines, developers are still forced to spend time and effort reimplementing common game features, such as basic AI, pathfinding, and simple scene variations. This can make it difficult for developers to focus on the game's core mechanics and content, especially if they are not experienced programmers or are uninterested in the development of complex algorithms. In addition, the implementation of such features is often closely coupled with the game's logic and architecture, making code reuse difficult.

The work in this thesis suggests an approach to game development where parts of the game logic are written in a declarative programming language. We do not reject the use of imperative programming but rather suggest tools that can be a part of and enhance the game development process. Specifically, this research focuses on the application of Answer Set Programming (ASP), a declarative programming paradigm, in video game development. The study demonstrates how ASP can be used as part of the game development process, by providing new ways of thinking about game mechanics, paving the way for new game design possibilities.

We highlight the adaptability of ASP to a wide range of problems and address the solver's ability to generate solutions in a reasonable amount of time. In addition, auxiliary tools are presented that can enhance the ASP programming experience.

A framework is illustrated that integrates an ASP solver into a game engine. Inside it, the designer can specify the rules that govern a specific game mechanic such as the placement of objects or the behavior of NPCs. The framework then generates the corresponding solutions which are interpreted by the game engine. The framework is designed to be modular, simple and extensible.

Finally, some concrete applications of ASP in the context of games are presented. We also assess the proposed workflow's practical feasibility and effectiveness by conducting an empirical user study.

Acknowledgements

Finally reaching the end of this journey I feel the need to express my gratitude to the people that helped me along the way.

First and foremost, I would like to extend my deepest appreciation to my supervisor, Christos Fidas. His guidance and unwavering support throughout this research journey have been invaluable. Your insightful feedback and constructive criticism have greatly shaped the direction of this work and helped me refine my ideas. I am truly grateful for your mentorship, for believing in my abilities.

I am also indebted to the members of my supervising committee, Pavlos Peppas and Christos Sintoris. Thank you for your time, dedication, and for providing valuable insights and recommendations during the thesis defense.

To all those who have supported me, directly or indirectly, I extend my heartfelt gratitude. Your contributions, in various forms, have played an integral part in making me the person I am. Thank you for being a part of my life and for making this accomplishment possible. I would like to thank, in random order Ioannis P, Konstantinos K, Orestis M, Ioannis T, Ioannis K, Georgia M, Nikolas F, Angelos A, Markela B, Apostolos P, Eleftherios P, Georgios M, Virginia A.

To my loving partner, Ioanna, thank you for your endless patience, understanding, and unwavering belief in me. Your constant support, encouragement, and unwavering presence have been my pillar of strength during the highs and lows of life. You are my best friend.

Lastly, I would like to express my deepest appreciation to my family. To my parents, Nikos and Giota, thank you for instilling in me the values of hard work, dedication, and perseverance. Your unconditional love, sacrifices, and unwavering belief in my abilities have been a driving force behind my accomplishments. To my younger brother, Nikolas, I wish that you find happiness and I will be by your side for any help and support you might need.

Evangelos Lamprou
Patras, June 2023

Contents

1. Introduction

The game industry is continuously growing and evolving. As with other industries, new ways of creating games are being developed. Game engines offer a plethora of tools and features that can aid game designers in bringing their ideas to life [1].

Answer Set Programming (ASP) [31] is a declarative programming paradigm that has shown promise in solving complex reasoning problems in various fields such as employee assignment [87], legal reasoning [5] and even automatic music composition [9]. This thesis explores the application of ASP techniques in video game development. We will show that using declarative tools can significantly speed up development time and relieve game programmers from the burden of understanding and implementing complex algorithms, leading to flexible and reusable code while also enabling for a different approach to game development.

1.1 Problem Description

In recent years, video game development has increasingly relied on the integration of artificial intelligence (AI) to create more immersive and engaging experiences for players. These include believable agents, content generation and player profiling [111]. However, even with the advancement of modern tools, programmers are still tasked with reimplementing basic game functionality such as path finding, decision making and game level variation, in their game engine of choice's native language. Oftentimes, these tasks serve as a bottleneck in the development process as they take a significant amount of time to implement, interrupting the creative process of game design. In addition, the final implementations of these features are often closely tied to the game's logic and architecture, making code reuse a challenge. Even recently developed tools that can generate code automatically [72,115], might generate results that are incorrect and hard to debug. Furthermore, these tools are often hard to steer towards a desired result [10], while game developers require a high degree of control when developing systems [97]. Nevertheless, even if those models were able to produce perfect code, the problem of complexity remains, as a large amount of code, even if generated by a machine, needs to be maintained and possibly adapted if changes to the application's structure

occur.

1.2 Goals and Objectives

In this work, we seek to highlight the adaptability of ASP to a wide range of problems. We will show how simple ASP modules can be written and integrated into any game, while exploring the paradigm's potential to create new game mechanics[1] suitable to exist as part of a game experience.

We will propose a methodology and heuristics for succesfully applying ASP to game development. Our goal is to develop a framework for developing game mechanics that serve as complementary elements to enhance the overall gameplay experience. We emphasize the potential of ASP to create game mechanics or simply replace the traditional implementation of mechanics with ones running inside a symbolic AI engine.

1.3 Contribution and Scope

To accomplish this, we have developed a generic framework that integrates an ASP solver into a game runtime. In our implementation, we integrate the *clingo* [37] ASP solver into the *Godot* game engine [47]. *Godot* is an open source production-ready engine capable of producing both 2D and 3D games. Our solution also takes into consideration the slow speed of ASP reasoning, making the system suitable for real-time applications by using an architecture where game logic and ASP reasoning are run in parallel. We propose a pragmatic approach for this incorporation, taking into account the complexity of the final framework.

We develop a few demonstrations, showing how new game mechanics can be added and modified in a tutorial fashion. The framework we propose offers flexibility and extensibility, opening up possibilities for game designers and developers to easily experiment with and implement novel gameplay features. Our findings contribute to the advancement of game development practices, providing insights into the efficient implementation of new game mechanics, and offering practical examples that demonstrate the ease and effectiveness of such enhancements in game design and gameplay experiences. We also conduct an evaluation of the ASP paradigm with developers at various levels of familiarity with the paradigm.

[1] By game mechanics we refer to the rules that govern the the logic of a game world entity. This can include entities the player will interact with e.g. enemies, items, etc. as well as abstract entities that are used to control the game's flow such as level generation systems.

1.4 Structure of the Thesis

In Chapter 2, there is a brief overview of various approaches to the problems of automatic object placement and intelligent agents throughout the literature, highlighting the challenges that these problems pose and how declarative programming is relevant to solving them. In addition, previous attempts to integrate declarative programming and answer set programming into game development are also presented. In Chapter 3, we present the background knowledge required to understand the rest of the thesis. This includes a brief overview of ASP and some of it's applications and extensions, as well as a description of the game engine architecture. In Chapter 4 we discuss the specifics of our approach, some implementation details and the structure of the user study. Some concrete examples as well as results from the user study are discussed in Chapter 5. Finally, in Chapter 6 we comment on our contributions and propose directions for future research.

2. Related Work

In this section, we give a brief overview of the research landscape of common game development related tasks while also explaining how answer set programming fits in as a solution. In addition, we show certain attempts at integrating answer set programming into a modern game development workflow.

2.1 Automatic Object Placement/Level Generation

In recent years, there have been many different approaches to solving the problem of automatic object placement in virtual environments. Some of these approaches include rule-based methods, optimization-based techniques, physics-based approaches and machine learning-based algorithms. Answer Set Programming (ASP) is a relatively new declarative programming paradigm that has shown promise in solving combinatorial problems, including object placement.

Rule-based methods have been a popular approach to solving the problem of automatic object placement. In this approach, object placement is done manually by the designer inside a modeling tool, with their actions being limited by a set of rules [95]. The rules are a set of predefined constraints that try to ensure that objects placed retain geometric and physical consistency. Such rules can be things like "no two objects can overlap" and "objects should remain at rest". This kind of solution can help by limiting the designer's degrees of freedom to a manageable subset, but the designer is still forced to interact with each object individually. In addition, the system is not well suited to handle constraints that are not geometric in nature such as "objects should be placed in a visually appealing way".

Combining rule-based methods with semantic information can help to overcome some of these limitations. In [113], the authors present a powerful system (*CAPS*) which offers a plethora of features to the designer. *CAPS* uses geometric rules as well as a semantic database to first create a plausible scene by automatically placing objects one at a time. A pseudo physics engine is also employed (similar to [20]) to ensure that the objects placed are physically

consistent. In addition, the user can further refine the scene by interacting with the placed objects and getting recommendations. The final system can speed up the creation of scenes, but it is not clear how the semantic database is constructed or how it can be extended to handle more complex constraints or richer object metadata.

The work in [91, 101] showcases the use of text-based descriptions to generate 3D environments. Generating scenes from natural language can eliminate the need for a designer to interact with the scene. However, the inherent ambiguity of natural language makes it difficult for the result to be controllable, or at least to be steered in a particular direction. In addition, the tools developed in these papers are not publicly available, making it difficult to assess their performance.

Machine learning-based approaches have also been used to solve the problem of automatic object placement. In [88], the authors present a system that can generate indoor scenes using a combination of convolutional neural networks wherein each step of the generation process, the system predicts the next object to place, its position, and orientation. As with all machine learning-based approaches, the system is only as good as the training data it is given. Generating a wider variety of scenes would require a larger training set, which can be difficult to obtain. The controllability of the generated scenes is also limited since the user is not able to provide any additional information to the system.

A machine learning approach that is not based on neural networks is presented in [59] where the Wave Function Collapse algorithm is used to generate tile-based scenes. The final scene generated is locally similar to a given training sample. Thus, there is a higher coloration between the final scene and the given training sample, allowing a designer to more accurately predict and control the final result.

The application of answer set programming to the problem of automatic object placement is relatively new. In [4, 80, 99], answer set programming is employed to generate game levels. It is shown that using ASP can offer a great amount of controllability and expressiveness to the designer. In addition, constraints such as "a maze that needs to be solvable" can be easily conveyed. Performance, however, is a major issue, with most of the work in this area focusing on generating small scenes or levels, with work being done to improve performance by partitioning the generation process into multiple steps [19].

2.2 Agent AI

Creating high-quality agent AI, even for background characters has been a great challenge for the video game development field. A review of the state of the art in this field is presented in [32]. Established techniques include

behavior trees, finite state machines, and machine learning techniques such as reinforcement learning. These solutions can be very powerful, with plugins like [57, 66] (decision trees and finite state machines), enabling a designer to quickly develop functioning AI agents often without requiring extensive programming knowledge. However, in both of those cases, the developer needs to have a thorough understanding of the possible states and possible decisions an agent can make. If the agent's AI needs to be extended, the designer will need to manually insert the new possible states and their transitions, resulting in a heavy cognitive load.

Declarative programming has been employed in the past for the development of intelligent agents for games. The General Game Playing Description Language (GDL) [43] attempts to provide a common declarative language for the description of games on top of which intelligent agents can be developed. The *Prolog* programming language has also been used to develop bots for real-time video games. In [56], the *SWI Prolog* interpreter is used to develop bot agents for the game *Counter-Strike*. The *Prolog* runtime handles low-level actions such as movement and shooting as well as high-level actions such as motivation and goal planning. The use of code as data is also mentioned, modeling the communication between multiple bot agents as sharing *Prolog* predicates. The work however provides minimal details on the game state representation and the actual implementation of the AI, making the separation between high and low-level actions unclear.

A notable example that popularised the declarative idiom in the game development community is the video game *F.E.A.R.* [82] where the developers used an action planning system to control the behavior of the enemies in the game. However, as is often the case with commercial games, the implementation details of the system are not publicly available, making it hard to critique the quality of the design.

Answer set programming (ASP) has been used in the development of intelligent agents for games to achieve autonomous decision-making in dynamic and uncertain environments. Previous research has focused on applying ASP to games such as Sudoku and Connect-Four [33, 103, 114] where agents use logical reasoning to make decisions and solve puzzles. These kinds of applications showcase how ASP can develop elegant and short solutions to game-playing problems. However, the works are limited to turn-based games where the game state is static and the agent often has full knowledge of it. ASP has been used in the development of strategic game-playing agents that employ reasoning about the game state by combining smaller ASP modules. [86] In [24], a more general architecture is presented where agents become a combination of ASP modules, where the overall agent behavior is a combination of facts and rule sets. This work further highlights the extensibility of encoding agent logic inside a declarative environment. Applications are not limited to strictly goal-oriented games, as there have been attempts to develop an

Angry Birds agent that uses ASP [18] to reason about the game state with external information based on a physics simulation. In the same work, there is also the aspect of planning at a higher level, as the agent tackles the problem of finding the next best level to play and which levels to retry. These developments highlight the potential of ASP in developing intelligent agents for games, which can handle complex and dynamic environments.

Most of the works mentioned have applied ASP to the development of 2D games. There are not many examples of the application of ASP to 3D environments. The work done in [4] where ASP is used to generate 3D levels for the game *Portal* shows that the ASP solver can quickly reach large solving times when the size of the rooms and the number of objects in the scene increases. This is also the case when dealing with agents that must make decisions inside 3D space such as movement. There exist solutions to this issue of increasing complexity[1]. The rules and restrictions of the game can be encoded in a few lines of *clingo* code, making it a powerful tool for reasoning, especially when the solution space is kept relatively small.

2.3 Integrating ASP with Game Engines

An important aspect of the work presented in this thesis is the integration of ASP with a game engine.

In [99] the authors present the *Ludocore* engine, which is a game development environment that attempts to add a solely declarative interface to game design. Event calculus [93] is used to represent and reason about the game state. Bridging the gap between the declarative and procedural paradigms, especially in the context of game design, is a very interesting problem.

The work in this thesis is most similar to [3], where the authors present *ThinkEngine*, a system that integrates ASP with the *Unity* game engine. In this work, the authors give a clear and well thought-out the architecture of a system that will allow in-game agents to reason about long-term reasoning goals as well as short-term actions. The result is that of enriching existing game tools with declarative programming, rather than replacing them. In our work, we follow a similar approach.

[1] We give solutions to the issue of large solution space in the context of specific use-cases in chapter 5.

3. Background

> The only true wisdom is in knowing you know nothing.
>
> *Socrates*

In this chapter, we fist present the necessary background for understanding the ASP paradigm. We show how ASP can be used to model problems using some illustrative examples and present some auxiliary tools that can be used to provide a better programming interface or to gain further insight into the solutions found by the solver. In addition, we present the architecture of a popular open source game engine as an example of a game development environment. Finally, through a simple example, we show how ASP can fit into a traditional game development workflow.

3.1 Answer Set Programming

Answer Set Programming (ASP) [31] is a declarative problem solving paradigm with roots in logic programming and nonmonotonic reasoning. The work of [42] first formalized the semantics of stable models and the ASP core language. Programming using this paradigm is done in a family of languages sometimes called *AnsProlog* [41].

The idea behind ASP is to model a problem as a set of rules and facts and then use a solver to find it's solution(s). Solutions are represented by stable models also known as answer sets. Rules, facts and constraints which describe the problem are the elements of the program. The program is then fed to a solver which will find one or more solutions.

In traditional (imperative) programming, getting from problem to solution involves the programmer understanding the given problem and producing a program which when given an instance of the problem will produce output which will then be interpreted as the solution (fig. 3.1).

In answer set programming, getting from problem statement to a set of solutions involves the following steps (fig. 3.2) [36].

1. **Modelling**: A problem is modelled in ASP syntax.

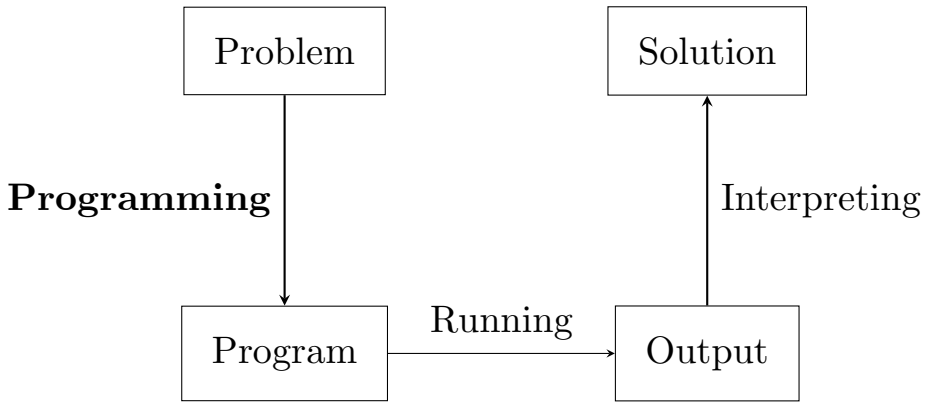


Figure 3.1: The process from problem to solution using imperative programming.

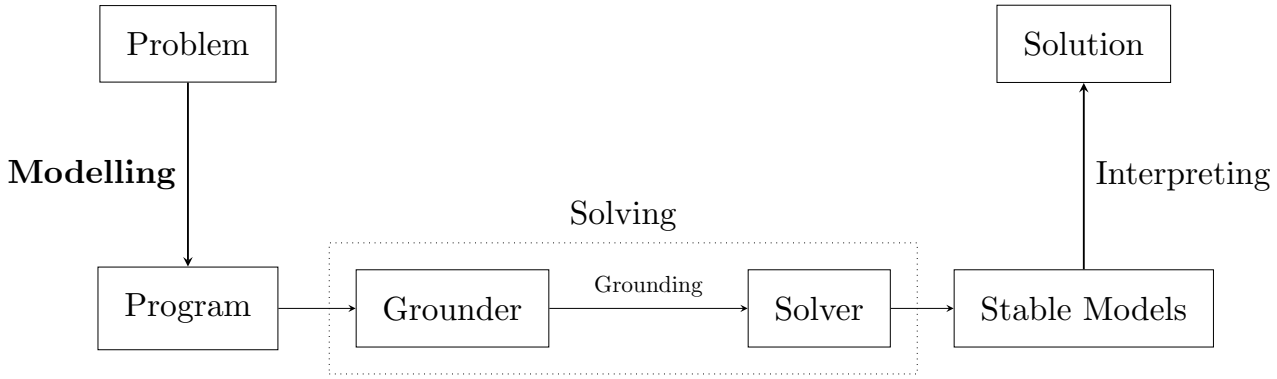


Figure 3.2: The Answer Set Programming process from problem to solution.

2. **Grounding**: A grounder (e.g. *gringo*) transforms the ASP program into a set of ground rules and facts.

3. **Solving**: A solver (e.g. *clasp*) finds a solution to the problem by computing the set of stable models (answer sets).

There have been a number of industrially applicable solvers developed for ASP such as *DLV* [31]. In this thesis, we will be applying the ASP system *Clingo* [37] which combines the *gringo* grounder with the *clasp* [53] solver into a single application, while also providing a powerful *Python* (and *Lua*) API for embedding the solver in other applications.

Using answer set programming, most of the development time is placed in the "modelling" phase, where the programmer has to describe the problem using a set of facts and rules. A discussion on methodology for this process can be found in [11] and is presented later in this thesis.

```
% Facts
name("John").
age("John", 99).

% Rule
old(X) :- name(X), age(X, Y), Y > 90.
```

```
$ clingo example.pl
Solving...
Answer: 1
old("John") age("John", 99) name("John")
```

Listing 1: A simple ASP program.

3.1.1 Syntax

Following is a brief introduction to Answer Set programming's syntax with focus on the particularities of the *Clingo*[1]. Some syntax elements not relevant to our work are omitted for brevity[2].

Terms

Terms are the basic building blocks of ASP programs. They can be integers, strings, constants, variables, or functions. Integers and strings are similar to their counterparts in other programming languages. Constants are identifiers which start with a lower case letter, while variables start with an upper case letter.

A variable is a placeholder for a value which is not yet known. A variable takes the same value if it appears multiple times in the same rule or fact. A special type of variable is the anonymous variable which is represented by the underscore character `_`. Anonymous variables are used to match a term without binding it to a variable, effectively ignoring it.

Functions are used to represent complex terms. As an example, the term `at("John", time(12), P)` is a function consisting of three arguments, the string `"John"`, the function `time(12)` and the variable `P`.

Normal Programs and Integrity Constraints

Terms can be atoms (named symbols, strings and numbers) or compounds consisting of a symbol and a list of logical terms as arguments. These com-

[1] *clingo* system from version 4.0 onwards adheres to the ASP language standard. [107].

[2] Official documentation for *Clingo* can be found in [34].

$$
\begin{array}{lcll}
\mathcal{A} & ::= & T. & \textit{Atom} \\
\mathcal{F} & ::= & A. & \textit{Fact} \\
\mathcal{L} & ::= & A. & \textit{Literal} \\
 & | & \textit{not } A. & \\
\mathcal{R} & ::= & A \ :- \ L_1, \ldots, L_n. & \textit{Rule} \\
\mathcal{C} & ::= & :- \ L_1, \ldots, L_n. & \textit{Integrity Constraint} \\
\mathcal{S} & ::= & l \ \{L_1; \ldots; L_n\} \ u \ :- R & \textit{Choice Atom} \\
\mathcal{G} & ::= & l \ op_{aggregate}\{L_1 = w_1; \ldots; L_n = w_n\} \ u & \textit{Aggregate} \\
\mathcal{O} & ::= & op_{optimize}\{L_1 = w_1; \ldots; L_n = w_n\} & \textit{Optimization}
\end{array}
$$

Grammar 3.1: A normal program grammar.

pounds can be thought of as function-like entities, of the form $f(t_1, \ldots, t_n) \rightarrow \{True, False\}$.

Rules constitute more complex logical sentences comprising of a head (the left part of the rule) and a body (the right part of the rule). If the body part of the rule can be derived, then the head part is also true. A comma between literals is equivalent to an "and" and the reuse of a rule with different body terms constitutes an "or".

The *not* keyword designates "negation as failure", where the literal *not a* evaluates to true if the literal a cannot be derived.

Integrity constraints specify that certain conditions should never hold. When we introduce choice atoms, this will allow us to "filter" the generated answer sets to the ones that are valid in our modelling context.

Choice Atoms

The generative and decision-making capabilities of Answer Set Programming are enabled by choice atoms (or choice rules). A choice atom entails that the ASP solver is free to choose one of the literals L_i to be true.

Aggregates

An aggregate has an lower bound l and an upper bound u, a multiset of literals L_i where each literal is assigned to a weight. If the weight is omitted, it defaults to one (1). The aggregate function $op_{aggregate}$ is one of:

- `#sum`: The sum of the weights.
- `#max`: The maximum weight.
- `#min`: The minimum weight.
- `#avg`: The average of all the weights.

```
person("John").
person("Bill").
person("Maria").

people(N) :- N = #count { X : person(X) }.
```

```
$ clingo aggregate-example.lp
Solving...
Answer: 1
person("John") person("Bill") person("Maria") people(3)
```

Listing 2: An ASP program with an aggregate, which counts the number of people in the knowledge base. We use the `count` aggregate function.

- `#count`: The number of unique literals.

Optimization

Answer set programming allows for optimization across multiple answer sets. Optimization can be used to find the answer set with the smallest or largest value of a certain function, which is achieved using the `#minimize` or `#maximize` directive respectively.

3.1.2 Semantics

Let $\mathbf{P}$ be an ASP program. The programs under consideration are sets of rules of the form

$$a \leftarrow l_1, \ldots, l_n \tag{3.1}$$

where a is an atom and $l_1, \ldots, l_n$ are literals as shown in 3.1 (i.e., atoms or negated atoms). The *Herbrand universe* of a first-order language $\mathbf{L}$ is the set of all ground terms of $\mathbf{L}$. The set of all predicate atoms which can be constructed by combining predicate names appearing in $\mathbf{P}$, with elements of the Herbrand universe of $\mathbf{P}$, is called *Herbrand base* of $\mathbf{P}$. A *(Herbrand) interpretation* $\mathcal{I}$ is a subset of the *Herbrand base* of $\mathbf{P}$ for which all atoms in $\mathbf{P}$ are true.

A rule is ground if it contains no variables. The ground program $\mathbf{P}^g$ of $\mathbf{P}$ is the set of all ground rules obtained by replacing all variables each of the rules in $\mathbf{P}$ by all combinations. of the *Herbrand universe*'s constants. We can think of the program $\mathbf{P}$ as a concise representation of $\mathbf{P}^g$.

```
person("John").
person("Bill").
person("Maria").

0 { invite(X) } 1 :- person(X).

invited(N) :- N = #count { X : invite(X) }.

#maximize { X : invited(X) }.
```

```
$ clingo invite.pl
Solving...
person("John") person("Bill") person("Maria")
invite("John") invite("Bill")
invite("Maria") invited(3)
Optimization: -3
OPTIMUM FOUND

Models       : 4
  Optimum    : yes
Optimization : -3
```

Listing 3: An ASP program with an optimization directive, which finds the answer set with the largest number of invited people. Obviously, the optimum answer set is the one where everyone is invited.

Let $\mathcal{I}$ be a Herbrand interpretation of $\mathbf{P}$. For a variable-free predicate atom a, $\mathcal{I} \models a$ iff $a \in \mathcal{I}$. For a default negated literal a, $\mathcal{I} \models not a$ iff $a \not\models \mathcal{I}$, and for a classically negated literal a, $\mathcal{I} \models \neg a$ iff $\neg a \in \mathcal{I}$. For a choice atom $l\ \{a_1; \ldots; a_n\}\ u$, $\mathcal{I} \models l\ \{a_1; \ldots; a_n\}\ u$ iff $l \leq |\{a_k : \mathcal{I} \models a_k \text{ and } 0 \leq k \leq n\}| \leq u$.

A rule of $\mathbf{P}$ is said to be satisfied by $\mathcal{I}$ if, $\mathcal{I} \models a$ whenever $\mathcal{I} \models l_1, \ldots, l_n$. The rules for which $\mathcal{I} \models l_1, \ldots, l_n$ constitute the reduct $\mathbf{p}^{\rangle}$ with respect to $\mathcal{I}$.[3] The reduct $\mathbf{p}^{\rangle}$ can be thought of as the essential parts of program $\mathbf{P}$.

In essence, a logic program $\mathbf{P}$ is satisfied by an interpretation $\mathcal{I}$ iff, all rules of $\mathbf{P}$ are satisfied by $\mathcal{I}$.

An interpretation $\mathcal{I}$ that satisfies program $\mathbf{P}$ is called a model of $\mathbf{P}$. A model for which no proper subset is a model is called a *minimal model* of $\mathbf{P}$.

Definition 1 (Answer Set)**.** An interpretation $\mathcal{I}$ is an ***answer set*** or *stable model* of $\mathbf{P}$ iff, $\mathcal{I}$ is a minimal model of the reduct $\mathbf{P}^{\mathcal{I}}$.

Consider the program

$$\begin{aligned} p(1). \\ q(2). \\ q(x) \leftarrow p(x). \end{aligned} \tag{3.2}$$

Two valid models are $\{p(1), q(2), q(1)\}$ and $\{p(1), p, (2), q(1), q(2)\}$. However, of the two models, only the first one is a stable model.

3.1.3 Grounding and Solving

The programmer can treat the solver as a black box [99], but it is useful to understand the process of grounding and solving in order to be able to debug and optimize the program. *Clingo* combines the grounding and solving process into a single step but the programs *gringo* and *clasp* can be invoked separately for us to see the intermediate representation of ground programs. Newer versions of *Clingo* allow the programmer to control the solving and grounding process via an embedded scripting language (*Lua* and *Python*) [40].

Here, we will present briefly what is involved in the grounding and solving process. We show the intermediate ground representation of an answer set program and provide an algorithm used for producing all of it's stable models.

Grounding

Grounding (or instantiation) in the context of answer set programming involves converting a logic program into a propositional format by systematically replacing all variables by variable-free terms. [58] The resulting program has no variables but has the same answer sets as the original program.

[3] The interested reader can find a more detailed presentation of the reduct algebraic structure in [13].

The grounding problem's complexity can be considered as polynomial ($O(n^a),\ a > 1$) when dealing with fixed non ground programs. However, when variable programs are considered, the complexity becomes exponential ($O(2^n)$) [60]. Work is being done to improve or eliminate the grounding process by means of lazy grounding [109].

Consider the following logic program

$$\begin{aligned} p(1).p(2).p(3). \\ q(3) \leftarrow not\ r(3). \\ r(X) \leftarrow p(X) \wedge not\ q(X). \end{aligned} \tag{3.3}$$

The grounding process produces the following ground program

```
$ gringo -t program.pl

p(1).
p(2).
p(3).
r(1):-not q(1).
r(2):-not q(2).
r(3):-not q(3).
q(3):-not r(3).
```

The second rule of our logic program was replaced by three (3) rules, each replacing variable X with a different possible value $(1, 2, 3)$. Notice how the predicate $p(X)$ has been optimized out of the rule.

Solving

An answer set solver is a program takes the propositional program representation provided by the grounder and computes it's answer sets. The process is similar in concept to feeding problems to a SAT solver, which produces one or more truth assignments that satisfy the problem. There are many different solvers [30, 39, 71] that support the *AnsProlog* syntax and can used interchangeably, producing the same answer sets for the same ASP program.

An Answer Set Solver Algorithm We will now show a possible implementation for an answer set solver [21]. Our solver will take as input a logic program and have as output all of it's answer sets, or the empty set if none exist.

Our program will be a set of rules (r) of the form

$$a \leftarrow p_1 \wedge p_2 \wedge \ldots \wedge p_n \wedge not\ n_1 \wedge not\ n_2 \wedge \cdots \wedge not\ n_n \tag{3.4}$$

```
Function ComputeAnswerSets(P)
    Data: A logic program P
    Result: P's Answer Sets or ∅
    return Solve(P, ∅, ∅)
end
```

1

```
Function Solve(P, CS, CN)
    if Expand(P, CS, CN) = false then
        return ∅
    end
    ⟨CS, CN⟩ ⟵ Expand(P, CS, CN)
    Select an atom a ∉ CS ∪ CN
    return Solve(P, CS ∪ {a}, CN) ∪ Solve(P, CS, CN ∪ {a})
end
```

2

```
Function Expand(P, CS, CN)
    Data: A logic program P, a set of atoms CS, a set of atoms CN
    repeat
        change ⟵ false
        forall rule r in P do
            if r_positive ⊆ CS and r_negative ⊆ CN then
                add r_head to CS
                change ⟵ true
            end
        end
        forall rule r in P do
            if r_positive ∩ CS ≠ ∅ and r_negative ∩ CN ≠ ∅ then
                add head(r) to CN
                change ⟵ true
            end
        end
    until change is false;
    if CS ∩ CN = ∅ then
        return ⟨CS, CN⟩
    else
        return false
    end
end
```

Algorithm 1: A basic answer set solving algorithm [21].

We define $r_{positive}$ to be the set of positive atoms in r, $r_{negative}$ to be the set of negative atoms in r and r_{head} to be atom a in the head of r.

In order to obtain all the answer sets using algorithm 1, the main function `ComputeAnswerSets` invokes the `Solve` function with an initially empty answer set. Within the `Solve` call, the answer set is iteratively expanded by deducing all possible atoms that can be inferred from the current atoms in the answer set, and subsequently adding one new atom at a time. The process continues until all answer sets have been computed.

Many of the concepts present in ASP solvers were first introduced in the field of satisfiability testing [8]. Modern ASP solvers use more advanced techniques. Most can be placed into three categories based on the algorithm or mapping they use: depth first search (DPLL) such as *Smodels* [81], mapping to SAT such as *ASSAT* and *Cmodels* [44, 71] and hybrid approaches such as *clasp* [39] which use conflict driven algorithms inspired by SAT solvers. In the context of conflict resolution, conflicts are examined and "noted", and decisions are made based on conflict scores. Genetic algorithms have also been proposed for this purpose, while ant colony optimization has been demonstrated as a core algorithm for small-scale answer set solving examples.

3.1.4 Event Calculus

Here, we will give a formal presentation of the event calculus formalism and how it relates to our work. In following sections, when building logic programs for incorporating into a game, we will not consider or point out rules relating to events as axioms of event calculus but instead as obvious and intuitive parts of the modelling process of a time-variant environment. Nevertheless, a formal presentation can often help with the development of our intuition.

Event Calculus (EC) [65, 93] is a formal logical framework that is commonly used in artificial intelligence and knowledge representation to model and reason about dynamic and temporal phenomena. It provides a way to represent events, states, and actions, along with their relationships and dependencies, using formal logic. It allows for the modeling of causality, time, and change, and provides a powerful tool for capturing and reasoning about the dynamics of events and their effects over time.

When developing agents inside game environments, the time domain is often present, requiring us to take into account the sequence of events occurring as well as the connections between action and result.

Ontology and Predicates of Event Calculus

Using these predicates, one can describe complex time-variant systems.

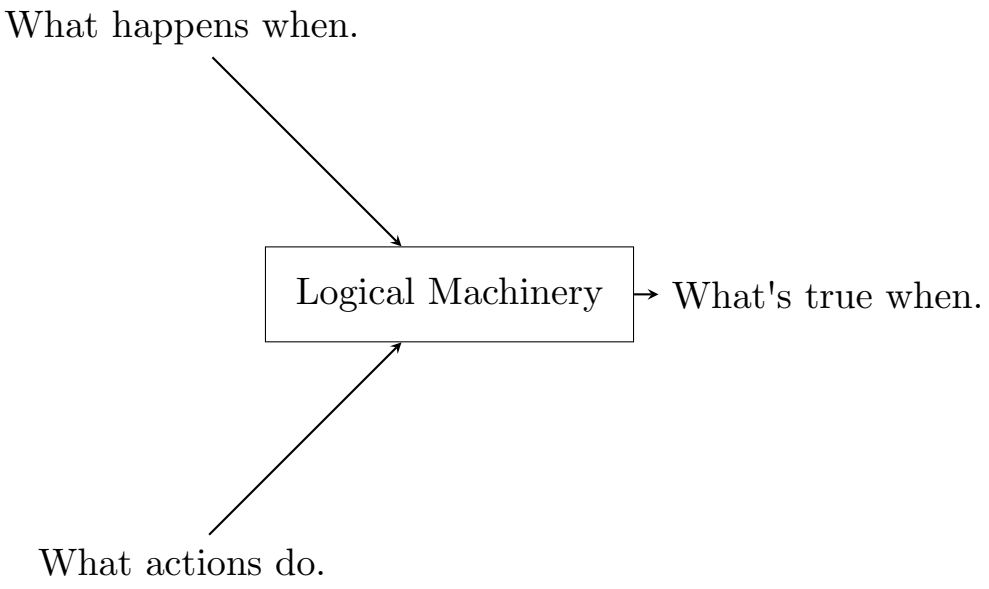


Figure 3.3: How the Event Calculus functions [93].

Predicate	**Meaning**
HoldsAt(b, t)	Fluent b holds (is true) at time t
Initiates(e, b, t)	Fluent b starts to hold after event e at time t
Terminates(e, b, t)	Fluent b ceases to hold after event e at time t
Happens(e, t)	Event a happens at time t
StartedIn(e, t_1, t_2)	Event e is initiated between times in the interval $[t_1, t_2]$
StoppedIn(e, t_1, t_2)	Event e is terminated between times in the interval $[t_1, t_2]$
Releases(a, b, t)	Fluent b stops being subject to inertia after action a at time t.

Table 3.1: Some Event Calculus predicates.

The Frame Problem in Event Calculus

A challenging aspect of modelling temporal domains in logic languages is that an action's non-effects also need to be represented. This is often referred to as the Frame Problem, first discussed in [75].

Solutions include using predicate completion, where the non-effects of actions and non-occurrence of events are made explicit by adding rules that represent the *completion* of the Initiates, Terminates, and Happens predicates.

Another approach is one of circumscription [74, 93]. Circumscription involves minimizing the extensions of certain named predicates in a formula. The Event Calculus is split into different parts, which are circumscribed separately to minimize the extensions of the predicates. The circumscription of a formula Φ yields a theory in which these predicates have the smallest extension allowable according to Φ. The circumscription of Φ minimising the predicate ρ is written,

$$CIRC[\Phi; \rho] \iff \Phi \wedge \neg\exists q[\Phi(q) \wedge q < \rho] \tag{3.5}$$

where $\Phi(q)$ is the formula Φ with all occurrences of ρ replaced by q.

Understanding the above formula, however is not necessary for our purposes.[4]

The Event Calculus and Answer Set Programming

Putting the theory of event calculus inside an answer set programming environment can be done by using the reformulation of the Event Calculus in Answer Set Programming by [69] in conjunction with the discrete Event Calculus (DEC) axioms[5] [79]. This can be thought of as including a "library" in our ASP program, allowing us to now employ the event calculus predicates as shown in table 3.1 to reason about dynamic event-based systems.

In addition, built on top of *Clingo*'s theory capabilities [35], the tool *Telingo* [17] can be used to model and reason about temporal domains using a slightly different interface.

Telingo offers a slightly better interface when encoding temporal domains because of it's understanding of effect axioms. Notice how there is no need to include a time argument in any of the predicates. These are rules that constitute the results of certain actions in our modelling phase. Nevertheless, "obvious" non-action effects, such as how items that are not moved stay where they were, still need to be explicitly authored.

3.1.5 Examples

Following are some motivating examples that showcase the use of ASP on some concrete cases. Through these examples, it is not our goal to appreciate the complexity of the given problems, but rather to prove how they can be efficiently modelled and solved succinctly using ASP.

Graph Coloring

Example 3.1.1. Consider a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. The goal is to color the nodes of a graph such that no two adjacent nodes have the same color. In the usual problem definition, we need to color the nodes with at most N colors. This problem in NP-complete for $N \geq 3$ [83].

We try to solve a more general case of the problem, where we need to find the *minimum* number of colors needed to color the graph, given the above constraint.

[4] The interested reader can find more information on circumscription in [70] and a history of it's relevance to the frame problem in [93].

[5] The axioms which are used in our examples are presented in appendix A.1.

We will first define our specific problem instance, representing the graph as a collection of facts of the form $edge(v, v'), v, v' \in V$. We will represent each vertex using a unique number $n \in \mathbb{Z}$. However, this is just a modelling choice and the vertices could have just as well been given unique alphanumeric names, possibly representing entities.

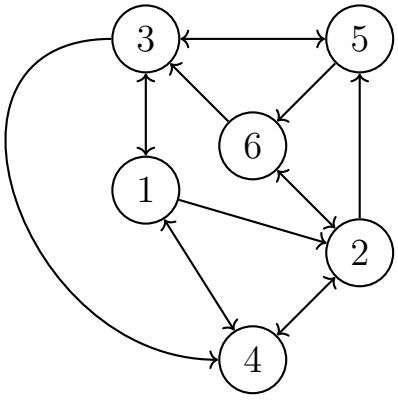


Figure 3.4: An example graph instance with six (6) vertices and seventeen (17) edges.

We encode the graph in fig. 3.4.

```
edge(1,2). edge(1,3). edge(1,4).
edge(2,5). edge(2,6). edge(2,4).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).
```

Defining the colors can be done similarly to the vertices, with a number $c \in \mathbb{Z}$ representing each different color. However, in the interest of semantic clarity, we decide to encode the possible vertex colors with facts of the form $color(c)$, where c is the color's name.

```
color(red).
color(blue).
color(green).
color(yellow).
color(purple).
color(cyan).
```

We now proceed with encoding the problem using non-ground rules which are independent of the specific problem instance. Notice how the *generation*

```
% Generate
1 { node_color(N, C) : color(C) } 1 :- node(N).

% Test
:- edge(I, J), node_color(I, C), node_color(J, C).

% Optimize
#minimize { 1, C : node_color(_, C) }.

#show node_color/2.
```

```
$ clingo ncolor.lp
Solving...
Answer: 2
node_color(1,yellow) node_color(2,green) node_color(3,green)
node_color(4,blue) node_color(5,yellow) node_color(6,blue)
Optimization: 3
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 3
```

part of the encoding is separate from the part enforcing the integrity constraints (typically known as *test* part).

In seven (7) lines of code, we have managed to model and solve the problem. We can see that the least amount of colors that can be used are three (3).

Minimum Path Robot Traversal

Example 3.1.2. Consider a robot that can move in four directions: up, down, left, and right. This robot is placed in a grid of size $n \times n$. Let g_{goal} be the goal position of the robot. We must find the shortest path from the robot's starting position to the goal position taking into account possible obstacles in the grid.

We first define the grid size and the facts the robot's starting position and obstacle positions.

```
blocked(0,1).
blocked(1,1).
```

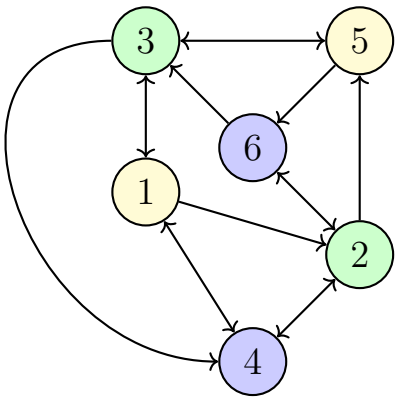


Figure 3.5: The solution to our N-coloring problem instance based on the optimum answer set.

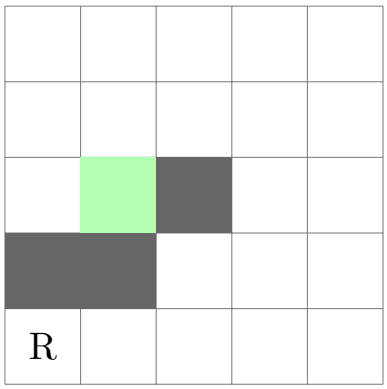


Figure 3.6: Our robot's grid-world.

```
3  blocked(2,2).
4
5  grid(0..cols - 1, 0..rows - 1).
6
7  robot(0,0,0).
8
9  goal :- robot(1, 2, _).
10 :- not goal.
```

Representing the robot's position is done using predicates $robot(x, y, t)$, where x and y are the coordinates of the robot and t is the time step. Our notation goes in line with the event calculus logical language [92]. Our robot will be able to move at one of the specified directions in each time step. We will use the following predicates to represent the robot's movement:

A shortcoming is how we need to specify a time range within which our robot will act (using the predicate $t(n), 0 \leq n \leq \mathtt{n}$). One has to choose a large enough `n` such that a solution will always be found or if a solution were to be found for larger `n`, it would not be of use. In this example we pick `n` to be

```
% Actions
t(0..n).
direction(up;down;left;right).
0 { move(D, T): direction(D) } 1 :- t(T).

action(T) :- move(_, T).

% Events
robot(X, Y + 1, T + 1) :- robot(X, Y, T), move(up, T).
robot(X, Y - 1, T + 1) :- robot(X, Y, T), move(down, T).
robot(X - 1, Y, T + 1) :- robot(X, Y, T), move(left, T).
robot(X + 1, Y, T + 1) :- robot(X, Y, T), move(right, T).

robot(X, Y, T+1) :- robot(X, Y, T), not action(T), t(T).

% Integrity constraints
:- robot(X, Y, _), not grid(X, Y).
:- robot(X, Y, T), blocked(X, Y), T=0..n.
```

Listing 4: The ASP encoding of the robot traversal problem.

equal to the number of cells in the grid. We make the assumption that the robot will reach it's goal in this time frame. Having an infinite time frame would be convenient for the programmer's peace of mind, but it would equate to an infinite time spent in the grounding phase.

An important part of the modelling phase for event-based problems (relating to the frame problem [27, 49]) is the last rule of the successor states (section 3.1.5, highlighted), which implies that if the robot does not move in a time step, it will stay in the same position. An intuitive explanation is that states that remain unchanged, should continue on to the next time-step.

With those rules in place, we have fully modelled the robot's world and the actions it can take inside of it. All that remains is to make the robot take the **shortest** path between the starting and final grid cell. Traditionally, a developer (especially in a game development context) would employ a graph search algorithm like A* [94]. However, in the context of ASP and *Clingo*, we can simply add an optimization directive.

```
last_action(TLast) :- TLast = #max { T : action(T) }.
#minimize {T : last_action(T) }.
```

```
$ clingo robot.pl
Solving...
Answer: 1
move(right,0) move(right,1) move(right,2) move(up,3)
move(up,4) move(up,5) move(left,6) move(left,7)
move(down,8)
Optimization: 8
OPTIMUM FOUND

Models       : 1
  Optimum    : yes
Optimization : 8
```

Listing 5: The output of *Clingo* for the robot traversal problem (fig. 3.6).

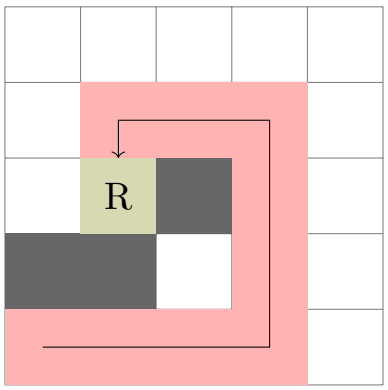


Figure 3.7: The robot's optimal path.

Menu Order Planning

Example 3.1.3. Let n be the total number of items to choose from (menu items), and let c_i denote the cost of item i. Let C be our target cost. The goal is to select a subset (S) of the items (with repetition) such that $\sum_{i \in S} c_i = C$. This problem is similar to the Knapsack problem [63], but with the items being repeatable and the total cost having to be exactly C.

First, we appose the facts relating to our specific problem instance. We will provide the cost of each item as a fact of the form $cost(i, c_i)$.

```
1 item(mixed_fruit).
2 cost(mixed_fruit, 215).
3 % ...
```

Because of *Clingo*'s lack of support for floating point numbers, we will

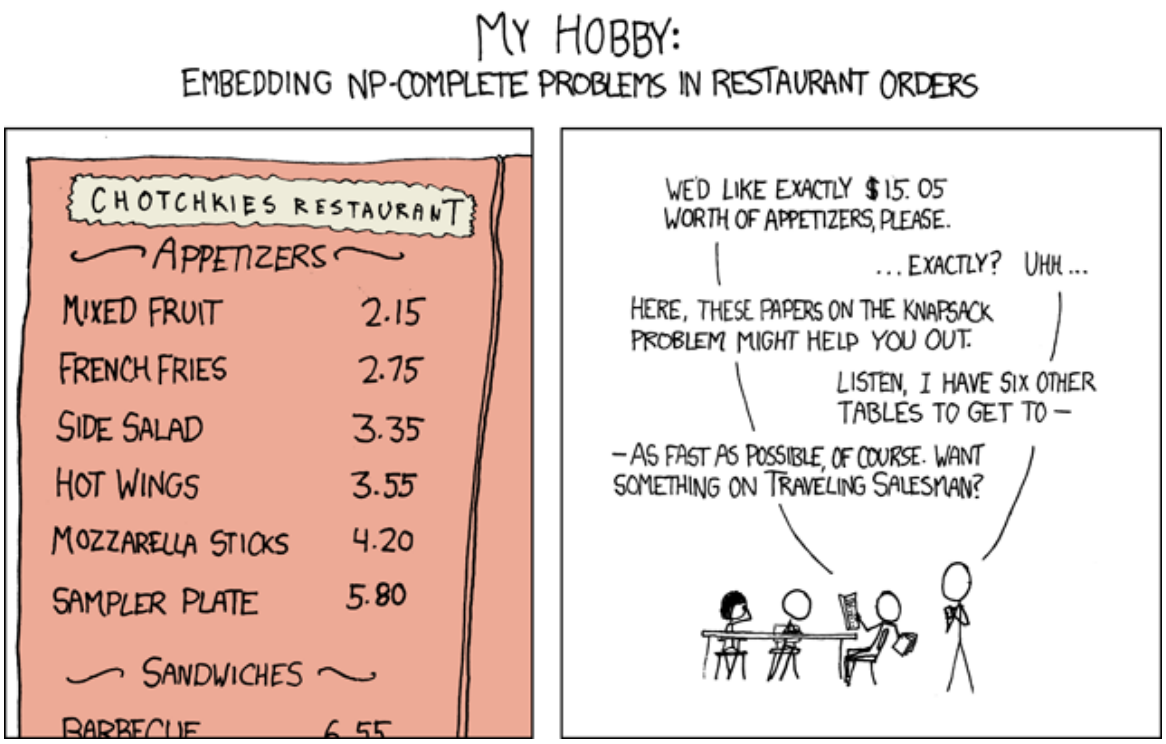


Figure 3.8: A relevant xkcd comic to example 3.1.3 [112].

have the cost of each item in cents. Later, we will provide an alternative implementation which will allow us to have each item's cost as a floating point number, intercepting the grounding process by leveraging *Clingo*'s *Python* API.

Modelling the "selection" of items might be a bit counter-intuitive. We will add a choice atom which will choose zero (0) or one (1) facts of the form $picked(c_i, n)$. We need to specify the amount of item's picked of a certain kind because we can't, for example, have two facts $picked(c_i), picked(c_i)$, designating that we chose item c_i two (2) times. These two facts would act as a single one.

Finally, we add the rule which enforces that the total amount of appetizers is equal to our goal amount.

```
#const goal = 1505.
n(1..10).

0 { picked(I, N) : n(N) } 1 :- item(I).

total_cost(I, C) :- cost(I, Ci), picked(I, N), C = N * Ci.
:- not #sum { C : total_cost(I, C), item(I) } == goal.

#show picked/2.
```

```
$ clingo menu.pl
Solving...
Answer: 1
picked(mixed_fruit,7)
SATISFIABLE

Models       : 1+
```

The solver, thus, recommends that we order seven (7) *mixed fruit* plates. However, that is a rather uninteresting order. We can see from `clingo`'s output that there are more valid models (answer sets). In the interest making an order with more variety, we can use an optimization directive to try and maximize the amount of different menu items in our order, while still adhering to the rest of the problem's constraints.

```
#maximize { 1,I : picked(I, _) }.
```

```
$ clingo menu.pl
Solving...
Answer: 1
picked(mixed_fruit,7)
Optimization: -1
Answer: 2
picked(mixed_fruit,1) picked(hot_wings,2) picked(sampler_plate,1)
Optimization: -3
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : -3
```

This order is a little more interesting, with one (1) *mixed fruit* plate, two (2) *hot wings* plates, and one (1) *sampler plate*.

Now, as promised, we will provide an alternative implementation which will allow us to have each item's cost as a floating point number.

```
#script(python)
import clingo
import math

def money(m):
    m = float(m.string)
```

```
    n = round(m * 100)
    return clingo.Number(n)

#end.

cost(mixed_fruit, @money("2.15")).
% ...
```

Here, we define a function call `money` which takes a `clingo` symbol (the money ammount) and returns a `clingo` number (the money ammount in cents). Our function runs as part of the grounding phase, effectively replacing in-place the string passed into it with the corresponding integer value. The rest of the problem encoding remains the same.

In conclusion, the examples presented in this section showcase the versatility and expressive power of Answer Set Programming (ASP) as a knowledge representation and reasoning paradigm. From modeling complex planning problems to solving puzzles and generating menu orders, ASP has demonstrated its ability to capture a wide range of real-world scenarios and provide efficient solutions. The ease of encoding domain-specific knowledge and the availability of powerful ASP solvers make it a valuable tool for addressing challenging problems in various domains, including artificial intelligence, robotics, constraint logic programming, and knowledge representation. The presented examples highlight the potential of ASP as a powerful and flexible tool for knowledge-based reasoning and decision-making, opening up new avenues for research and application development. It is not hard to think of ways to apply these examples in the context of a game. For instance, example 3.1.1 could be relevant in terrain generation. The terrain could be split into nodes where adjacent nodes need to be of a different biome. With ASP, we could derive the least amount of biomes necessary in order to make adjacent terrain nodes be of different type. In example 3.1.2, a game agent would be tasked with traversing a maze in order to reach the player character and in example 3.1.3, we could use ASP to generate restaurant orders for the player to prepare in a cooking game.

3.2 Explainable AI

The field of Explainable AI (XAI) has gained significant attention in recent years due to the increasing reliance on artificial intelligence (AI) systems in various domains. [108], it has become imperative to understand and explain their decision-making processes to ensure transparency, accountability, and trustworthiness. XAI seeks to bridge the gap between the complex decision-making capabilities of AI and the need for human understandable explana-

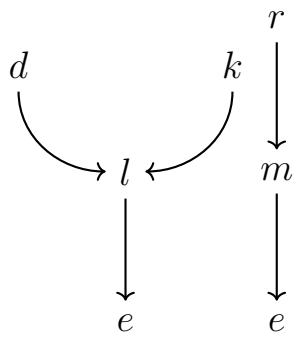


Figure 3.9: Derivation graph for the atom e in the program eq. (3.6).

tions, thereby enabling stakeholders to comprehend and trust the outcomes of AI-powered systems.

Explainability does only offer the ability to see how an AI system got it's results but also the ability to debug and improve it. Most research currently being done in developing deep learning frameworks for various tasks mostly involve trial and error [14] without clear correlation between specific choices being made in model architectures and the final results.

In this section, we show that using AI techniques rooted in logic can produce inherently explainable AI systems, that can also be useful in game contexts. Consider a game character not only being able to make smart decisions but also be capable of explaining how they reached a conclusion.

3.2.1 Explainability in ASP

Symbolic artificial intelligence has one major advantage over sub-symbolic approaches because of it's explainability. The relations between facts and conclusions made by a system running a symbolic engine are clearly defined. Thus, results from reasoning can be traced back. However, ASP solvers like *Clingo* does not provide[6] human-readable traces of their reasoning procedures.

The tool *xclingo*[7] [16] allows for justifications to be presented in relation to the results given by the *Clingo* solver. This is done by adding `trace` directives corresponding to the rules of our logic program. Giving a clear explanation of how the system reached specific conclusions can help with debugging the program as well as give the system's user some insight on it's output. The tool's functionality is based on the work in [15], where the causal graph structure is presented, a graph **G** whose vertices correspond to "rules involved in a derivation of a given atom (or formula)" and edges represent a partial ordering of application of rules that resulted in that derivation. The logic program's true atoms are associated with their justifications.

[6] As of the writing of this thesis and to my knowledge.

[7] https://github.com/bramucas/xclingo2

```
% A course was unsuccesful if the student
% received a grades less than 5.
-coursesFinished(X) :- student(X), course(C),
                       grade(X, C, G), G < 5.
coursesFinished(X) :- student(X), not -coursesFinished(X).
-thesisFinished(X) :- student(X), not thesisFinished(X).

degree(X) :- student(X), coursesFinished(X), thesisFinished(X).

-degree(X) :- student(X), -coursesFinished(X).
-degree(X) :- student(X), -thesisFinished(X).

#show degree/1.

%!show_trace degree(X).
%!show_trace -degree(X).
```

Listing 6: ASP program deciding whether a degree should be awarded to a student.

Consider the following logic program P that models a drunk driving scenario [15].

$$\begin{aligned} l &: punish \leftarrow drive \wedge drunk & r &: resist \\ d &: drive & e &: prison \leftarrow punish \\ m &: punish \leftarrow resist & k &: drunk \end{aligned} \tag{3.6}$$

Consider an ASP program which models the qualifications a university student will need to have in order to receive a degree. This example was inspired from the work in [5] where ASP was applied for legal reasoning. Explainability is especially important in contexts where AI system's decisions can affect human lives.

We have derived that “nikolas” should be awarded a degree. However, we can't tell without closely and meticulously inspecting the ASP program why “evangelos” can not graduate.

With the help of *xclingo*, without any modifications to the rules of our original program, we can get the reason for which a student might not be eligible for a degree.

The `auto-trace` option makes sure to trace every rule in program. Alternatively, the programmer could add `trace` directives on specific rules, also supplying human-readable messages to accompany the logic trace. The out-

```
course(databases). course(ai).

student(nikolas). student(evangelos).

grade(evangelos, databases, 9). grade(evangelos, ai, 3).
grade(nikolas, ai, 10). grade(nikolas, databases, 10).

thesisFinished(nikolas).
thesisFinished(evangelos).
```

```
$ clingo university.pl
Solving...

degree(nikolas)

SATISFIABLE
```

Listing 7: Our specific problem instance corresponding to program in listing 6.

```
$ xclingo university.pl --auto-trace all -n 0 0
Answer 1
  *
  |__-degree(evangelos)
  |   |__student(evangelos);student(evangelos)
  |   |__-coursesFinished(evangelos)
  |   |   |__student(evangelos);student(evangelos)
  |   |   |__course(ai)
  |   |   |__grade(evangelos,ai,3)

  *
  |__degree(nikolas)
  |   |__student(nikolas);student(nikolas)
  |   |__coursesFinished(nikolas)
  |   |   |__student(nikolas);student(nikolas)
  |   |__thesisFinished(nikolas)
```

Listing 8: Running our program with *xclingo* and the `auto-trace` option.

put includes a tree-like structure which shows the predicates responsible for specific conclusions for each of the rules. In our example, it becomes apparent that one of students has not completed their responsibilities for the "ai" course, receiving a grade 3 rather than the minimum of 5.

It's important to note that the *xclingo* system works only with classically negated predicates, requiring us to include the rules that lead to a student *not* getting a degree ($-degree(X)$). The system could be even friendlier for the program author if it could derive that a student should not be awarded a degree simply on the basis of negation as failure. For example, the reason for someone without a finished thesis not getting a degree would be $not\ thesisFinished(X)$. It's possible that we could answer "why not" questions without requiring the program author to add classical negation predicates by doing a program transformation during grounding. This is a possible continuation to the work in [16] but outside the scope of this thesis.

3.3 Game Engines/Game Tools

In this section we will present how, in general, a game development workflow is realized in relation to the game development tools available. We present the popular game engine *Godot* and comment on how answer set programming can fit into it.

3.3.1 The Godot Engine

The Godot Engine [47] is a free and open source game engine. It is written in *C++* while supporting a number of scripting languages such as *GDScript*, *C#*, and even *Python*, trough community developed modules. The engine is cross-platform and supports a number of platforms such as Windows[8], Linux[9], macOS[10], Android[11], iOS[12], HTML5[13], and UWP (Universal Windows Platform)[14]. The engine is designed to be extensible and flexible, while also being more lightweight than similar engines such as Unity [48, 106].

Godot's basic Concepts

In this section, we present the basic concepts of the *Godot* Engine. We will only dwell on the technical details to the extent where they are valuable in

[8] https://www.microsoft.com/en-us/windows
[9] https://www.kernel.org/
[10] https://www.apple.com/macos
[11] https://www.android.com/
[12] https://www.apple.com/ios
[13] The HTML5 standard can found in https://www.w3.org/TR/2011/WD-html5-20110405/
[14] https://learn.microsoft.com/en-us/windows/uwp

understanding our contributions.

Scenes In *Godot*, a game is composed of a number of scenes. A scene can be anything from a character, a level or a menu. Scenes are composed of nodes, which are the basic building blocks of a game inside the engine. Scenes can be nested, allowing for the creation of complex objects. For example, a character scene can be composed of the character's body and a weapon they are holding, which itself can be a separate scene.

Nodes A scene is composed by one or more nodes. A node can be a sprite, a mesh, a camera, a light, a collision shape. New nodes can be created by the programmer or downloaded from the *Godot Asset Library*[15].

Scene Tree The nodes of a scene are organized in a tree structure, called the *scene tree*. Nodes of the tree can have children, which are represented visually as branches of the tree. This parent-child relationship can be leveraged, for example, to ensure a character's weapon is always in the character's hand.

Signals Communication between nodes can be achieved in a minimal code manner by using *signals*. These are named events that can be emitted by a node and received by another. The programmer can connect a signal to a function, which will be called when the signal is emitted. For example, a button node can emit a *pressed* signal when it is clicked, or a football goal node can emit a *goal* signal when a ball enters it.

Scripts Nodes can be extended by scripts, which are pieces of code that can be attached to a node and can be written in a number of languages, such as *GDScript*, *C#* and *Python*. *Godot* has some predefined callback function which are callback at specific moments in the game's runtime. For example, the `_process` function is called every frame, while the `_ready` function is called when the node is added to the scene tree. This is where logic such as agent AI will be placed. Through this scripting API, we are allowed to invoke calls to the operating system, thus allowing for the integration of external tools such as an ASP solver.

Godot's Design Philosophy

The workflow provided by each different game engine is heavily intertwined with it's structure. Some game engines are highly specialized, providing tools for development of games in specific genres such as RPGs [26] and Interactive Fiction [62]. Godot strives to be a general purpose game engine. In this

[15] https://godotengine.org/asset-library/asset

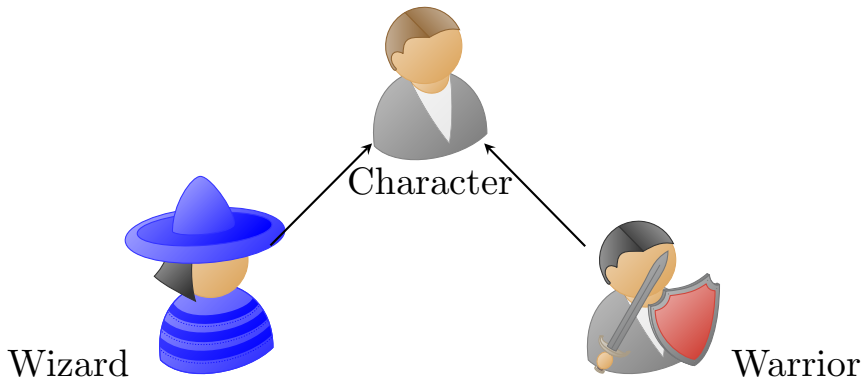


Figure 3.10: A character scene extended to a wizard and warrior scene.

section we will provide a brief overview of Godot's design philosophy, how it affects the programming experience and comment on some of it's weaknesses.

Object-oriented design and composition It can't be argued that object-oriented programming has dominated software engineering from the early to mid 1990s [51]. Object orientation lends itself well to game development since there is often a one-to-one mapping between the objects in the program's memory with the objects inside the game world.

Godot allows the developer to compose and aggregate scenes. For example, one might create a *Sword* scene and a *Flaming Sword* scene which itself uses the *Sword* scene, adding flames as a particle component on it of it. Using this kind of design, if the developer decided to change the original *Sword* scene (for example by swapping it's texture for another one), all other scenes that use it will be updated as well. Godot also offers an inheritance mechanism, which allows for scenes to be extended, adding on new functionality or modifying their in-world appearance. With a rudimentary involvement with object-oriented software, one can see the point of this approach, as the project structure tries to match the game's design [46]. One shortcoming of this approach is that it is not always clear how a project should be structured. For example, an inexperienced developer, or someone that has not worked with or studied object oriented principles, might fail to detect inherent similarities between objects. For example, given a wizard and a warrior scene, they might resort to create two separate scenes, rather than a single *Character* scene that is extended by both the *Wizard* and *Warrior* scenes. While this issue of user experience is outside of the scope of this thesis, it is worth noting that even tools that present themselves as non-developer friendly still require a basic understanding of software engineering principles.

In *Godot*, most game development, aside from behavior and game logic is done by navigating and interacting with Godot's menus. The programmer does not need to write boilerplate code in order to define a new scene or inherit

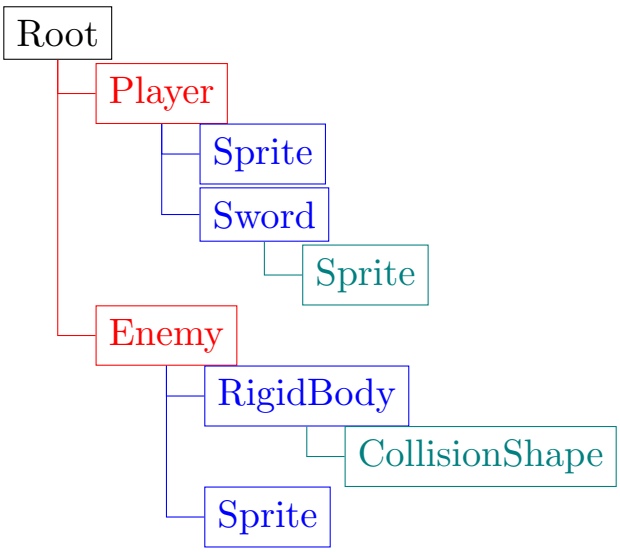


Figure 3.11: The tree structure the example scene is composed of.

from an existing one. In addition, Godot provides a number of built-in nodes that can be used to create game objects. These nodes are organized in a tree structure, where each node can have a number of children nodes. The set of nodes a base node has as children can be thought of as it's "capabilities". For example, a *Sprite* node can be used to display a texture on the screen, while a *CollisionShape* node can be used to define a collision shape for a *RigidBody* node, which can make it's parent node react to collisions and gravity.

In fig. 3.11, we have a scene comprising of a player and an enemy. These entities can be construed as abstract nodes that represent the respective characters. The player node, for instance, consists of a sprite node that is responsible for rendering the appearance of the player. The sword node, serves as a plausible representation of a sword that the player might be wielding. The positioning of the sword, as a child of the player node, is dependent on the parent's position. The enemy node, has a rigid body node as a child, which imparts physics properties such as mass and velocity. Additionally, a collision shape node is required to be added as a child of the rigid body node to enable the calculation of collisions.

Extensibility Godot is designed to be extensible. Developers have the ability to create their own plugins and modules, which can be used to extend the engine's functionality. These can be used to add new nodes, new editor functionality or even new scripting languages. In this thesis, we will be developing a module which will allow for the integration of ASP with Godot.

Godot's Architecture

In this section we will provide an overview of Godot's architecture. Godot is a complex system consisting of many moving parts. As of the writing of this thesis, the project spans almost three million[16] lines of code. However, ample documentation as well as quality design choices have made it possible to reason about the engine's structure.

The architecture as seen in fig. 3.12 consists of four core distinct pieces:

- **Drivers**: This is the part of the engine responsible for providing wrappers around common operating system functions like threading and process creation, file input and output, audio and low-level communication with the GPU.
- **Servers**: These are the engine's subsystems each corresponding to a major part of the game's logic. Examples of such servers are the *PhysicsServer* and the *VisualServer*, responsible for 3D Physics calculations and screen drawing, respectively.
- **Scene**: This part of the engine is the one a developer will likely be most familiar with. It involves the main engine abstractions such as Resources, the *SceneTree* and Node. Each part of this subsystem interacts directly with one or more servers. For example, a Node representing a moving ball interacts both with the Physics server for it's movement calculations as well as the *VisualServer* for getting drawn onto the screen.

Interacting with Shared Libraries

Shared libraries [29, 64], also known as dynamic link libraries or shared objects, are essential components in modern software development. They are pre-compiled code modules that contain functions and resources that can be reused by multiple programs simultaneously, promoting code reuse and modularity. These libraries are loaded into memory at runtime and shared among multiple processes, reducing memory footprint and promoting efficient resource utilization. By separating commonly used code into shared libraries, developers can improve development efficiency, simplify maintenance, and enhance system performance. Additionally, shared libraries enable dynamic linking, allowing programs to link with the library's functions at runtime, enabling flexibility and the ability to update the library independently from the programs that rely on it. Overall, shared libraries play a crucial role in software development, providing a powerful mechanism for code reuse, modularity, and efficient resource management.

[16] Results were taken by running the `cloc` (`github.com/AlDanial/cloc`) utility on the project's main branch.

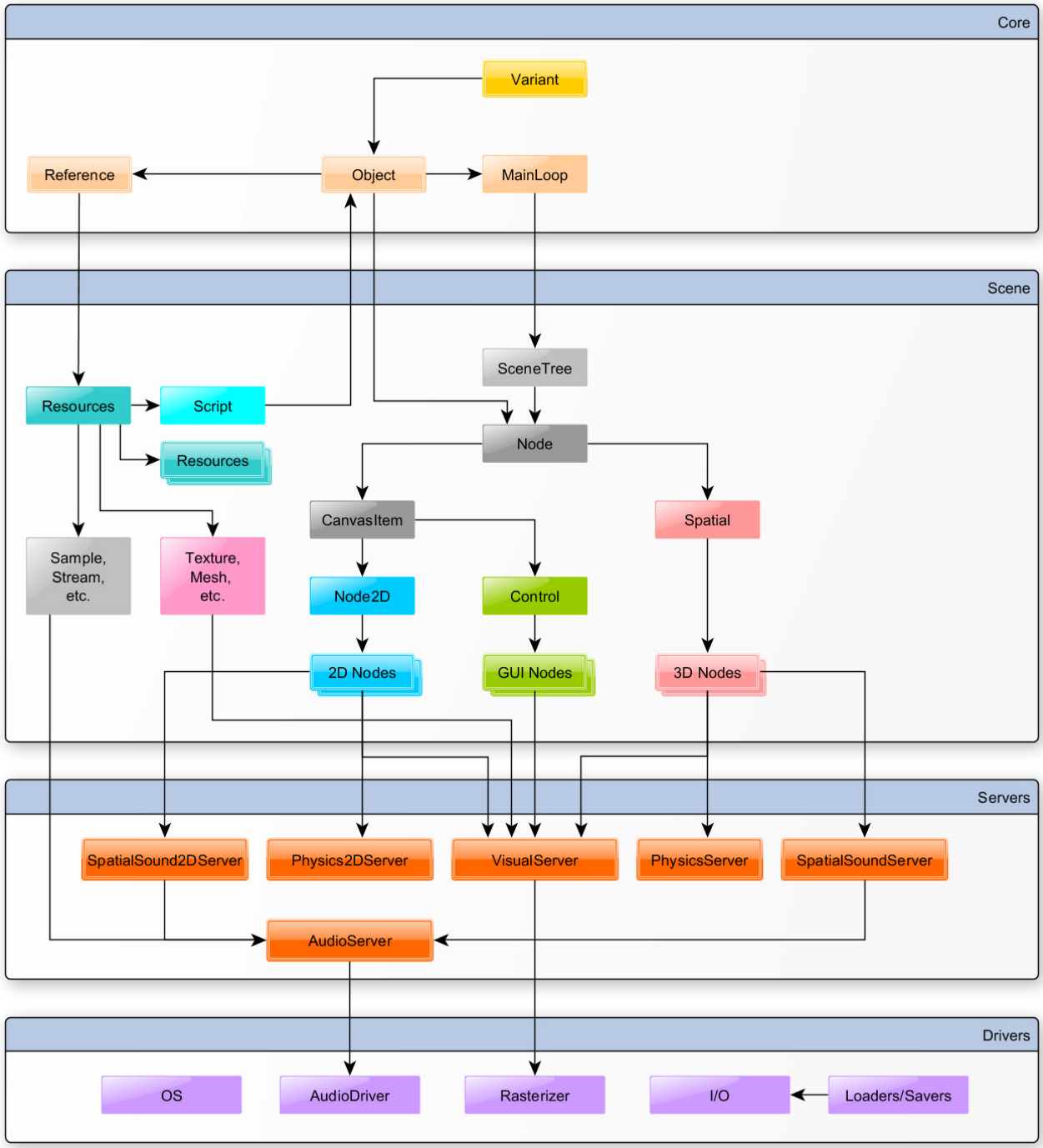


Figure 3.12: Godot's architecture diagram [45].

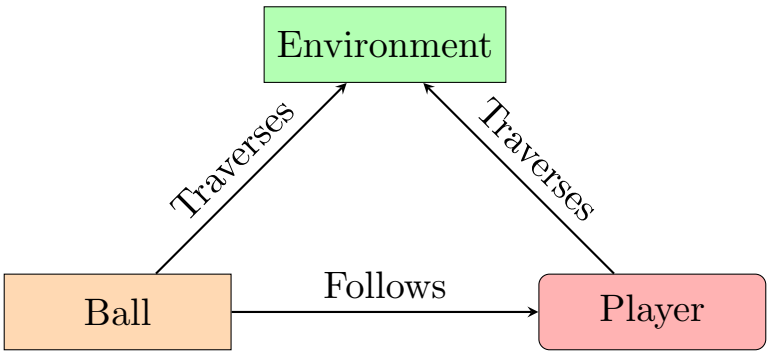


Figure 3.13: The game diagram.

In order for the *Godot* engine to interact with native shared libraries at run-time it uses the *GDNative* [23] technology, which has now been replaced by the equivalent API *GDExtension* [22]. It can be used to run native code without having to compile it with the engine. This allows for programmers to use languages not officially supported by the *Godot* engine to write game logic inside of. Bindings exist for languages like *Python*[17] and *Rust*[18]. Through these extensions, one can take advantage of modules written in another programming language for game object scripting inside the engine. We use this technology later in order to interface with the *Clingo* solver from inside *Godot* using *Python*.

3.3.2 Building a simple 3D game

In this section we will give a short tutorial on how a simple 3D game consisting of a player character and moving ball that can follow them. This serves as both as an introduction to the workflow that Godot offers as well as an opportunity to to show how logic programming can fit into it.

The list of features that will need to be implemented are:

- ☐ Create **Player** scene
  - ☐ Load player model and related assets
  - ☐ Make player a physics object
  - ☐ Program player control logic
- ☐ Create **Ball** scene
  - ☐ Load ball model and related assets
  - ☐ Make ball a physics object

[17] https://github.com/touilleMan/godot-python
[18] https://github.com/godot-rust/gdnative

☐ Program ball "player-following" logic

We begin by creating a new project in Godot. We then setup a `Main` scene where our game entities will be placed inside of. Afterwards, we create a scene with a *KinematicBody* representing the player character. For the player scene, we also add a *CollisionShape* node as a child of the player node in order for the player to collide with the environment as well as a *MeshInstance* node to represent the player's model. We then implement a script for the player character that enables it to respond to user input and move accordingly. This is a simple mapping between keyboard input and the player character's change in velocity. For this, we override the `_physics_process` method provided by the engine to all nodes that interact with the physics engine. This method is called at every step of the physics simulation. A *Camera* node is also added as a child to the player scene which follow the player's movement. This is used to render the scene from the player's perspective.

Next, we set up a scene with a *RigidBody* representing the ball and add a *CollisionShape* and *MeshInstance* node similarly to the player scene. We then develop a script for the ball that utilizes the Godot physics engine to calculate the direction and speed of the ball towards the player, allowing it to chase the player and bounce off of obstacles in the scene. This is done again by overriding the `_physics_process` method of the ball node. The ball's velocity normal vector can be calculated by subtracting the ball's position from the player's position. This vector can then be scaled by a constant speed factor to get the ball's velocity.

$$\vec{v_{ball}} = |\vec{p}_{player} - \vec{p}_{ball}| \cdot speed \tag{3.7}$$

Now, running the game will have a ball move towards the player around the scene. We can imagine that this ball could be some sort of enemy or a friendly pet that follows the player around.

This is a trivial problem and it would suffice to say that a game developer would not need to use logic programming to solve it. However, by implementing the ball's movement using ASP we can still highlight some of the paradigm's strengths.

The logic in listing 9 ends up being several times longer than the original script. This can hardly be considered an improvement. However, in the original program, the game developer had to understand the concept of vectors and how the difference between two position vectors can be used to calculate a direction vector. These of course are concepts any competent game developer should be familiar with. Nonetheless, when we deal with other game mechanics and scenarios, the list of required background knowledge required to implement the game logic begins to grow. In the ASP program, we don't ask the agent to move in any particular direction. Rather, after encoding the concepts of *distance* and *movement*, we allow the logic engine to figure

out how the agent should move. This simple example hopefully illustrates the motivation for considering ASP as a tool to ease the game development process.

3.3.3 Summary

In this chapter, we have introduced the concept of logic programming and the ASP paradigm in particular. We also presented a game engine and the process for game creation it provides. Finally, we discussed the motivation for using ASP in game development and how it can be used to solve a simple game development problem.

```
#const t_end = 5.
t(0..t_end).
direction(up;down;left;right;front;back).

object(target).
object(self).

% For every time step, the ball can move in any direction
0 { move(self, D, T) : direction(D) } 1 :- t(T).

position(O, vec3(X, Y - 1, Z), T + 1) :- object(O),
        position(O, vec3(X, Y, Z), T), move(O, up, T).
% ... movement rules for the other directions

% The ball stays in place if no movement is specified
position(O, vec3(X, Y, Z), T + 1) :-
        position(O, vec3(X, Y, Z), T),
        not move(O, _, T), t(T).

distance(D, T, O1, O2) :- position(O1, vec3(X1, Y1, Z1), T),
                          position(O2, vec3(X2, Y2, Z2), T),
                          D = |X1 - X2| + |Y1 - Y2| + |Z1 - Z2|.

% The ball tries to minimize the distance to the target
#minimize { D : distance(D, t_end, self, target) }.
```

Listing 9: The logic program responsible for the ball's movement written in *Clingo*. The programmer only needs to convey the relevant aspects of the game such as the agent's possible moves and how their distance to the target. Finally, a single `#minimize` directive results in answer sets that make the agent approach their target.

4. Methods

> My methodology is not knowing what I'm doing and making that work for me.
>
> *Stone Gossard*

This chapter serves as a comprehensive guide to the methodology employed for the integration of Answer Set Programming (ASP) into the domain of game development. The subsequent sections will delve into the various facets of this methodology, elucidating the steps taken to effectively apply ASP techniques within the context of game creation.

First, we will provide an in-depth overview of the software framework developed specifically for embedding an ASP solver into a game engine (section 4.1). Our design aims to be applicable to any game engine, given some specific technical assumptions. This framework, acts as the backbone for an easy-to-use integration of ASP principles into the game development process. In addition, we will outline a programming methodology tailored for the implementation of various game features using ASP (section 4.1.5). This methodology provides a systematic and structured approach to applying ASP techniques in a game development context. By following this methodology, game developers can quickly leverage the paradigm to start creating game mechanics and features. Furthermore, we propose applicability heuristics devised as part of this methodology (section 4.2.1). These heuristics serve as guidelines for determining whether a specific aspect of a game can be effectively implemented using ASP. By presenting these heuristics, we enable game developers to make informed decisions regarding the suitability of ASP in addressing particular challenges or requirements within their game design. This systematic approach ensures that the incorporation of ASP into game development is done judiciously, maximizing its potential benefits while minimizing potential limitations. Finally, we present the format of the user study we conducted (section 4.3), for evaluating the effectiveness and practicality of the proposed framework and methodology. Results of this study are discussed in section 5.4.

4.1 Integrating ASP into a Game Engine

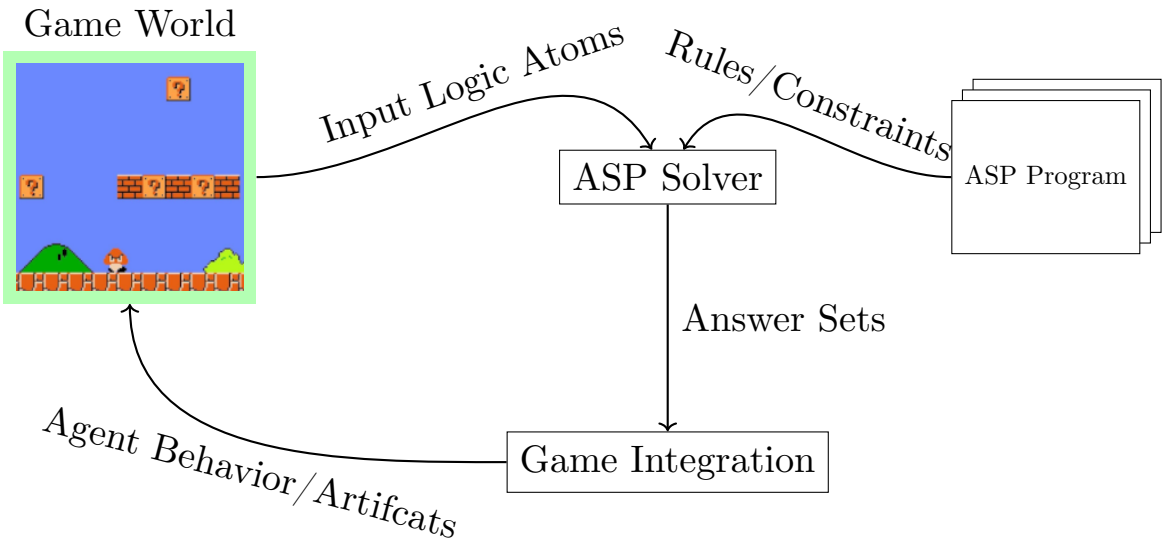


Figure 4.1: Integration of an ASP solver into a game engine. The *Game World* consists of the current game state and the information of all the game objects inside of it. Input logic atoms are the facts that are used to describe the current state of the game world. These, together with the rules and constraints of the ASP program, are fed into the ASP solver, which then outputs answer sets. These describe logic such as the actions that an agent should take or where an object should be placed. Through the *Game Integration* component, these answer sets are then used to update the game world.

4.1.1 Architecture

Here, we will present a generic design for integrating ASP into a game engine. The design is based on the following assumptions:

1. The game engine provides an API for creating sub processes and capturing their output or allows for the execution of programs in one the solver's supported languages (in `clingo`'s case `Lua` or `Python`).

2. The game engine provides an API for multi-threading.

Our architecture consists of the following parts:

- **Game World**: This represents the present game state. That includes all the data associated with objects inside the game world.

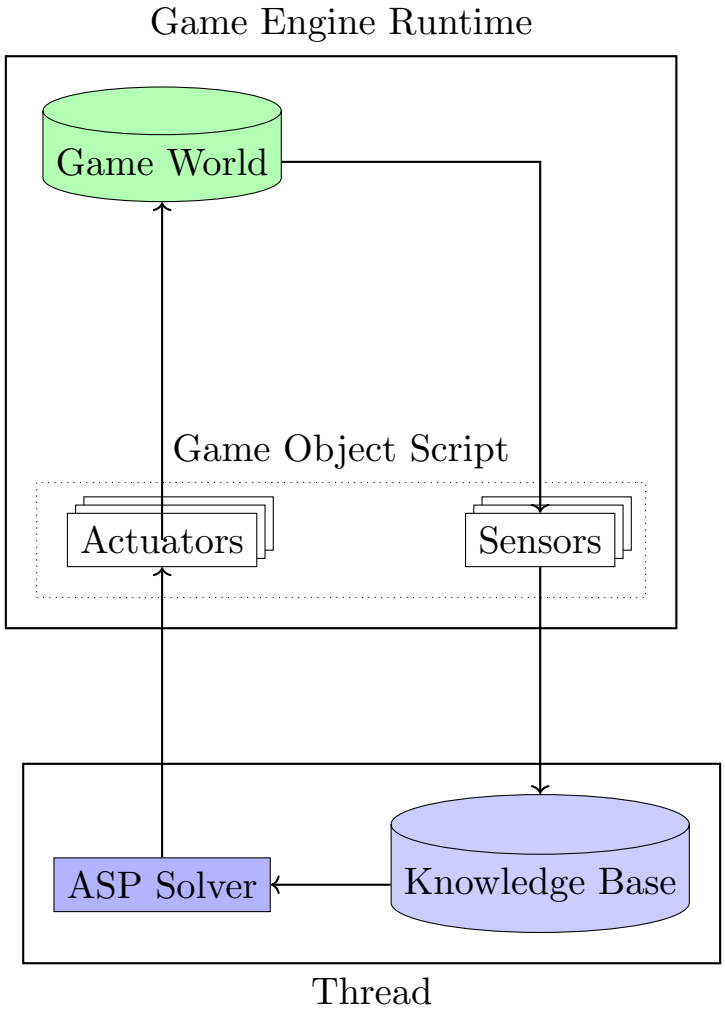


Figure 4.2: The generic framework's architecture.

- **Node Script**: This is the interface between the game programmer and *Game World*. Most engines provide callbacks that run per frame or per physics update which can be used to run supplementary logic inside. The node script code will invoke the `think` procedure periodically (for example every few milliseconds).
  - **Sensors**: This is the module responsible for converting the game state into ASP facts.
  - **Actuators**: This is the module responsible for converting facts returned from the ASP solver back into actions.
- **Knowledge Base**: This is the list of facts that the solver has access to. This knowledge base can include both long-term memory (facts that remain true throughout the game's runtime) and short-term memory (facts that will be true until the next *think step*).
- **Asynchronous Data Transfer**: To address situations where the game

engine's update times exceed the solver's solving times, it becomes necessary to implement a method for storing the output of the solver, allowing for game entities to "think" in parallel with their other procedures. For this, a data structure like a queue can be used [3].

This final design covers the vast majority of use cases where the ASP solver will be integrated in the game runtime. Further augmentation of this architecture might include the inclusion of two separate queues in order to accommodate for reactive parts of the AI code, where solving times are short and planning parts of the AI code where solving times can become large. We suggest that the game developer does not avert highly from the proposed design, since multiple levels of reasoning are better implemented using multiple instances of game object scripts, each corresponding to a different solver instance, helping with code modularity and separation of concerns.

In our work, we wanted to take advantage of a number of libraries that allow for a better programming interface with *clingo*, such as *clorm*. These libraries are only available[1] as modules for the *Python* programming language. This led to the need of integrating the *Python* language's runtime into the *Godot* engine for some of our demos.

4.1.2 Mapping the Game State to ASP

In this section we will describe how the game state can be mapped directly into answer set programming facts and rules. First, we will describe some common game data structures and how they can be mapped to ASP.

Data Structures

Integers and Strings: Integers and strings are supported by default by the `clingo` runtime. Floating point operations are not natively supported but can be encoded using `clingo`'s `Python` API.

Vectors: Inside a game engine, vectors are often used to represent positions, velocities, forces, etc. To represent a vector in ASP, we have a predicate of the form $vec3(X, Y, Z)$. Lower and higher dimension vectors can be similarly represented using predicates of the form $vecn(D_1, D_2, \ldots, D_n)$.

Axis-Aligned Bounding Boxes (AABBs): A 3D axis-aligned bounding box can be represented in ASP as two $vec3$ predicates. An example of such a predicate would be $aabb(vec3(x_1, y_1, z_1), vec3(x_2, y_2, z_2))$. An alternative modelling approach (which we used in section 5.2), is to first

[1] As of the writing of this thesis.

compute the AABB's size from inside the game engine and have as separate predicates the AABB's size as well as its position. These two options are semantically equivalent, but some program authors might prefer one over the other

Matrices: Matrices are often used to represent transformations (rotation, scaling, etc.) Matrices can be represented similar to vectors, by predicates of the form $matrix_{r\times c}(a, b, \dots)$.

Events

Events occurring or planned to occur inside the game world can be modelled by using predicates of the form $event(o, t)$ where o is the object that initiated the event and t the time instance where the event happened relative to a selected initial state. The initial state can either be the game start or more practically, the last *thinking* step, where the ASP solver was last called. It is often easier to reason about an agent's plan by simply using the *current* state as the temporal starting point, rather the game run-time's initial position.

4.1.3 Discretization of the 3D Space

This restriction over the solution space can be beneficial from the gameplay perspective. That kind of discrepancy between the actual game world and the discretized world inside the logic engine makes the agent prone to making small mistakes which can make the playing experience more fun for the player. This side of AI-design is sometimes referred to as Artificial Stupidity [25, 96, 105, 110].

However, since the actual game world is continuous or at least has a space delta that is much smaller than the one we deal with inside our logic programs, we can bridge that gap by applying a physics simulation to the agent's movement, making their traversal inside the game world seem natural and similar to an agent that is capable of moving in the full range of directions.

4.1.4 Object Relational Mapping (ORM)

Often times, the format in which a program's data is stored and retrieved does not match well with how a programmer would like to represent and deal with them. For example, in an application where users are stored in a database, a user can either be represented as a row inside a database table or an object with attributes and methods associated with it. This inconsistency is often referred to as the object-relational Impedance Mismatch Problem (IMP) [73]. Result of this is hard to extend and maintain code as their needs to be "translation" between the two forms [54, 104].

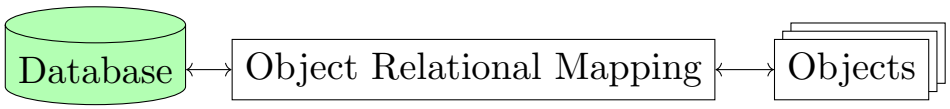


Figure 4.3: ORM as a bridge between a database and objects.

```
class Person(clorm.Predicate):
    name = clorm.ConstantField()
    age = clorm.IntegerField()

facts = [
    Person("Bob", 25),
    Person("Alice", 65)
]

problem_instance = clorm.FactBase(facts)
control.add_facts(problem_instance)
```

Listing 10: Encoding *clingo* predicates as *Python* objects and adding facts relating to our problem instance to the solver using the API provided by the *clorm* library.

Object relational mapping (ORM) is a method for converting data between a relational database and memory of a programming language, often providing an object-oriented interface.

The *clorm Python* library.

A popular Python library providing an ORM interface to *clingo* is available. The *clorm* [85] module allows for the mapping of facts to objects as well as querying the fact database using an interface similar to other Python ORM libraries [7].

The class-table matching is done by defining the schema. In the case of answer set programming, we define the form in which the ASP predicates appear inside our program. For example, a predicate of the form `person(name, age)`, can be encoded with *clorm* as follows:

It's important to note that when defining this mapping, we need to respect the order in which arguments appear inside the predicate (in our case, `name` is first and `age` second). However, when querying from the solution that clingo produces, order of arguments is irrelevant since we refer to each argument by its name (in our example `name` and `age`). Now, adding facts to the fact base can be done in a more programmer-friendly and readable way.

```python
if symbol.name == "young_person":
    name = str(symbol.arguments[0])
    age = str(symbol.arguments[1])
    print(name, age)
```

(a) Using the clingo Python API.

```python
for p in solution.query(Young_Person).all():
    print(p.name, p.age)
```

(b) Using the API provided by the *clorm* library (after specifying the schema).

Figure 4.4: Comparison of the *clorm* library usage when querying the fact database for solutions compared to clingo's default Python API.

Using this interface, the game developer can add new facts to the *clingo* fact database as well as interact with the solutions provided by the solver in more human-readable and less error-prone manner.

As can be seen in fig. 4.4, the *clorm* library provides a convenient interface for interacting with the fact database. Each fact argument can be associated with a name and a type, rather than the index where the argument appears in the atom. This allows for the use of a more natural syntax when querying the fact database and avoids breaking the code when the order of the arguments is changed in the ASP program.

It is important to note that object relational mapping is not a requirement for using ASP in game development, rather a tool for improving the interface imposed to the programmer.

4.1.5 ASP Development

We propose a standardized development methodology in order to guide aspiring developers to successfully apply ASP to their applications by providing some general programming guidelines relating to ASP modelling (fig. 4.1).

1. **Determine Input and Output Atoms**: The set of input atoms provide the context required for the ASP program to give correct results. These are usually dynamic aspects of the game's runtime and change at each invocation of the ASP solver. Examples of this are the starting location of an agent inside a game world or the list objects that need to be placed. On the other hand, output atoms encode the results produced by the solver and which will be interpreted by the game runtime

as artifacts or agent behavior. These include things like the direction at which an agent will move in the next timestep or the location an object should be placed at.

2. **Generate "Random" Answer Sets**: Based on the desired output atoms, the programmer can quickly create an ASP program that is a collection of choice rules that generates semi-random results. The artifacts or behavior produced this way will be incorrect or unsuitable but will provide an easy way to debug possible technical issues. This is the phase of development where some kind of visualizer or integration of the solver with the game runtime is created.

3. **Add Integrity Constraints/Optimization Directives**: Based on the current problem's domain, it is necessary to add integrity constraints and/or optimization directives. Constraints provide direct control over the produced answer sets for them to comply both with game's ruleset and the designer's ideas. Among them the designer can pick the most optimal ones based on some variable using optimization rules.

4.2 Design of ASP-Based Game Mechanics

In this section, we propose a methodology for applying ASP to game development and provide a set of guidelines for detecting game mechanics where the paradigm can be applied successfully.

4.2.1 ASP Applicability Heuristics

One important aspect of our work is the clarification of how specific game mechanics can pair well with ASP. These applicability heuristics stem from studying the related literature and analyzing the ways in which ASP has been used in game development so far. We have chosen to implement specific demos which possess the following characteristics:

- **Brevity**: Highlight the ways in which complex logic can be encoded briefly in an ASP program. ASP (and declarative programming in general) can reduce software complexity, [100] as well as the final program's size Even when the problem domain increases in sophistication, the code length and programming effort are not required to follow, making programs written in the paradigm more concise [12].

- **Relatively Small Solution Space**: Avoid scenarios where solving times become very large. Because of the real-time nature of most games, results from an ASP solver need to be produced in a time scale based

either to the frame rate of the game's runtime or the desired frequency for new agent decisions inside the game world.

- **Emergent Complexity**: Create scenarios where interesting behavior emerges when agents are observed interacting with each other and the environment inside the game world.

Symmetrically, there exist a few characteristics which render a game mechanic unsuitable for ASP:

- **Complicated Logic**: Avoid cases where the logic required to implement the game mechanic is very complicated and many different edge cases of the problem have to be modelled explicitly.
- **Large Solution Space**: Avoid scenarios where solving times become very large because of the many possible choices that the solver needs to explore.
- **Information Hunger**: Avoid scenarios where the agents need to have access to a large amount of information in order to make decisions. Having agents make decisions based on a large amount of information can be computationally expensive while also making the implementation of the game mechanic more complicated and error-prone. Nevertheless, complex reasoning tasks which require access to a large knowledge base are better implemented as high level reasoning tasks that can run in the background.

4.2.2 Game Design

ASP can lend itself to exploratory game design, where the designer can easily make adjustments to the artifacts they are producing to test new ideas and see immediate results [98].

The types of game mechanics where ASP can be interleaved in the development process can be split into three general categories [3, 98].

- **Planning AI**: Where an agent makes plans about a complex action that might involve complex reasoning.
- **Reactive AI**: Where an agent takes immediate action based on information about his state and surroundings.
- **Generative AI**: Where the designer tries to create static or dynamic artifacts and place them inside the game world. Generation might also involve meta-generation of artifacts such as game rules [61].

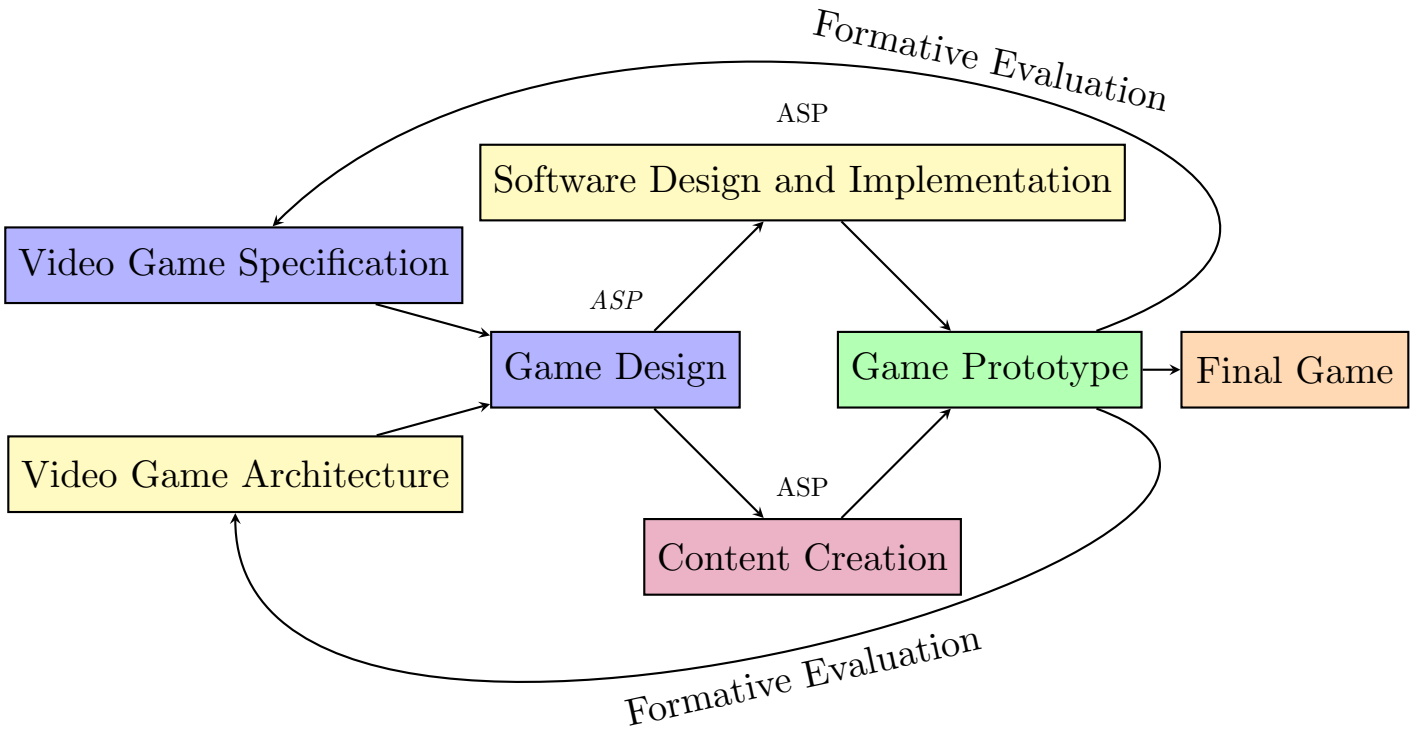


Figure 4.5: The various stages of game development encompass several points where the application of Answer Set Programming (ASP) can prove advantageous. The proposed framework facilitates the utilization of ASP in implementing game mechanics, presenting an interface that separates logic into distinct modules known as ASP Programs. This segregation contributes to improved software design and streamlines the implementation of game mechanics. ASP can be leveraged for the generation of game content. By specifying the desired content to the ASP program, designers can produce content automatically. This approach enhances the development process by enabling faster and easier iteration for both programmers and designers. This iterative workflow fosters greater possibilities for experimentation and exploration within the game's design space. As a result, the actual game design process is impacted positively, leading to a more dynamic approach to game development.

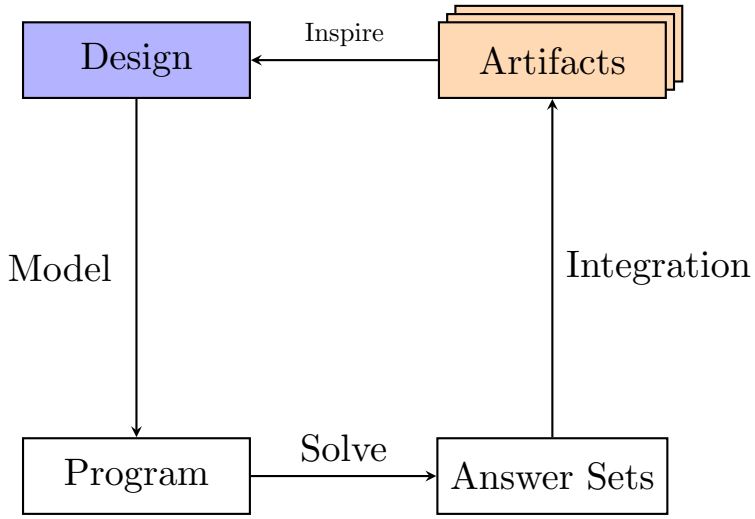


Figure 4.6: Modelling a game mechanic with the help of ASP tooling. The designer starts with an initial goal, the design of the wanted game mechanic/behavior/set of artifacts which leads to a specification in the form of an ASP program. The program's solutions can help to further refine the initial design as missing or unwanted aspects of it become apparent after it's integration with the rest of the game occurs.

The designer is free to add as much or as little complexity to their modelling. However, it is often more appropriate to model the core aspects of the world the end artifact will interact with. Often times, because of performance concerns, the designer will have to make some adjustments to their modelling in order to remove certain variables that might only act as noise to the solver, artificially enlarging the solving space without offering tangible improvement to the artifacts created. These kinds of limitations might seem destructive for the creative process but can often lead to more concrete and refined game scenarios, while also allowing the designer to approach the mechanics in a new eye, giving way to new ideas. In any case, changes to an ASP program are much easier to make because of the brevity of the language as well as its functional nature, making the code produced more modular and decoupled from the rest of the program [52]. A modification being a few lines of ASP away in comparison to a few hundred lines of code can make the difference between a designer being able to explore a new idea or not.

The designer starts from a hazy state, where they have a general idea of what they want to achieve. This process of free-thinking is interrupted by the need to model the program which when run, will result in a set of valid artifacts[2] close to the desired one. It is of major importance to keep the modelling process short and concise, where each subsequent iteration has a tangible effect on the artifact produced [98]. The final artifact will then result

[2]In this context, an *artifact* can be anything from a game object to an agent's behavior.

into changes in the design space, as the creator will have a better understanding of whether their initial idea is suitable, or rather making modelling improvements. It is possible that programmer has introduced possible bugs in the ASP program, by either forgetting edge cases or misinterpreting the desired behavior [3]. Incremental development is a good way to avoid this, as the designer can gradually add complexity to their model, while also testing the correctness of the program [11].

4.3 Testing and Evaluation

In this section we describe how we conducted a user study in order to evaluate the effectiveness of using Answer Set Programming (ASP) to create game mechanics or content generation procedures. In this study, participants were recruited to design game mechanics or content generation tools and implement them using ASP. Their outputs were analyzed to assess the feasibility and usefulness of this approach. The study aimed to answer the research question of whether ASP is a suitable tool for such tasks and whether it can provide advantages over traditional methods. The findings of the study contribute to the understanding of the potential of ASP as a tool for game development and provide insights into its strengths, limitations and also inform future refinements.

We recruited a total of 8 participants (2 female and 6 male), all of whom were undergraduate *Electrical and Computer Engineering* students. All of them had experience programming with imperative languages, with 3 having experience with logical languages (either *Prolog* or *Clingo*). All but two of the participants had prior experience with game development, in the context of personal projects. Participants were informed that no personal data was collected aside from their answers to the interview part of the study. On average, each of the participants took part in the study procedure for a duration of 0.5 to 3 hours, resulting in a total study length of approximately twenty-four hours. The study conducted was a one-on-one user study, where each participant worked individually with the researcher. The study utilized all collected data in an anonymous manner, and participants had the freedom to withdraw from the study at any time of their choosing.

- **Phase A - Introduction to ASP** The study began with a brief overview of Answer Set Programming (ASP) technology and the *clingo* language's syntax and semantics.

- **Phase B - Implementation of Game Mechanic/Generation of Content** Participants were then asked to think of a game mechanic or content generator that they would like to implement using ASP. We encouraged participants to be creative and come up with unique or chal-

lenging ideas. After participants had an idea in mind, the researcher assisted them in creating the logic program for their game mechanic using ASP. We avoided instructing the participants during the modelling process, where they would come up with the logic rules for their program and limited our interference to helping with issues concerning *clingo*'s syntax. During the creation process, the researcher was available to answer questions and provide guidance as needed. After each participant completed their game mechanic, the researcher evaluated it with them to discuss the strengths and weaknesses of the approach taken, and provided feedback on how it could be improved. The evaluation of each participant's output was based on several criteria, such as how well the game mechanic functioned, its uniqueness, its complexity, and its potential for being implemented in an actual game.

- **Phase C - Discussion** Finally, we conducted a semi-structured interview to receive qualitative feedback and elicit the participant's likeability and comments with regard to the proposed workflow.

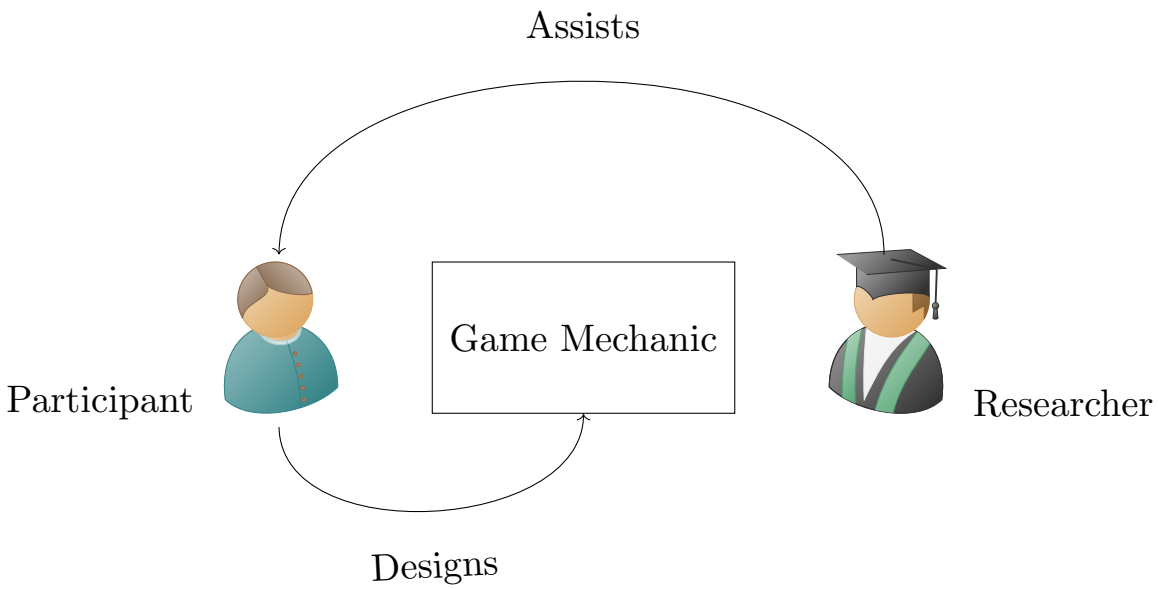


Figure 4.7: The experimental setup of the user study.

5. Case Studies

> Few things are harder to put up with than the annoyance of a good example.
>
> *Mark Twain*

The Case Studies section of this thesis presents some of the applications we developed using Answer Set Programming (ASP) in the context of games. These case studies demonstrate the potential of using ASP to create unique game mechanics. The goal is to show the versatility of ASP in game development, showing how it can be used to create different types of game mechanics and solve game development challenges. The use of ASP in game development allows for a more efficient and effective process as it allows for rapid prototyping and reduces the need for extensive programming and testing. The applications developed in these case studies provide a solid foundation for future research in the use of ASP in game development and highlight the paradigm's potential for practical application in the field.

5.1 Football (Soccer) Game

We developed a game scenario that showcases how ASP can model two teams of adversarial agents and how the end result is both interesting and appropriate. Our method avoids the need to implement path-finding algorithms like A* [94] or the application of reinforcement learning techniques [90] which can be hard to develop and debug during the stages of game prototyping. In this scenario, we give the agent's a limited number of actions per turn while also having multiple agents on each team, leading to emergent complexity in the final result, without requiring for us to explicitly model it. The characteristics of **relatively small solving space** and **emergent complexity** are present in this example. Our goal is not to develop AI that plays football "well", but rather to create intelligent agents that enhance the game experience for the human player. One possible application of our demo would be to have agents playing football in the game world while the human player is exploring the rest of environment. The player might have the choice to join in and play or simply stay an observer. Indirect interaction with the agents playing football

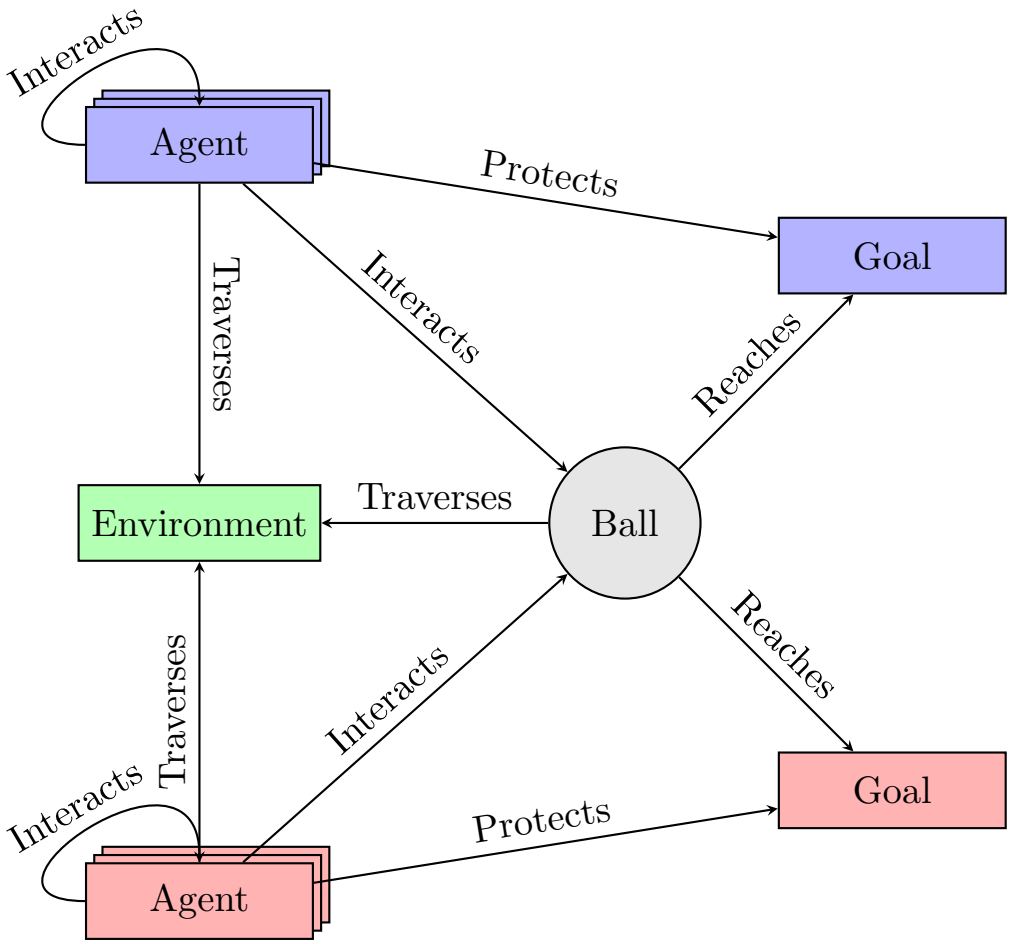


Figure 5.1: The football game diagram.

could be achieved by a betting mechanic where the player tries to predict the winning team. In most of these scenarios, the agent's skill level plays little role.

Football is a popular sport involving two teams that kick a ball to score a goal. The sport has had a close relationship with video games, with game titles that involved the player character playing football matches against AI opponents being developed since the beginning of the video game medium [50, 102].

5.1.1 The playing field

The playing field will consist of a flat terrain with two goals sitting opposite of each other. Each goal is modelled as a rectangle that detects collisions. Whenever the ball enters the collision shape, a point is scored for the team opposite of the one the goal belongs to.

5.1.2 Football-playing agents

The game is composed of two opposing teams, each of which has a goal. Intuitively, the agent's goal is to score as many goals as possible. Encoding this behavior in ASP can be done in multiple ways. We will show a possible modelling of the problem and discuss how it affects the final gameplay result as well as their pragmatic applications in a larger game experience.

We will show that the resulting logic program is not a monolithic array of statements but rather a collection of separable modules containing logic for sub problems [24]. These modules, when combined, produce an agent capable of "playing football".

We will follow an incremental approach to building the agent's behavior, similar to an approach we hope a game programmer to take. Debugging logic programs is often difficult because of their non-linear nature. Each constraint and fact affects the entirety of the program. Thus, it would be beneficial to the programmer and their sanity to triple check each subsection of the logic program to make sure the modelling satisfies the programmer's intents.

One of our logic "modules" is the one relating an object's movement. At each time step, the agent is allowed to move to one of eight (8) directions (up, down, left, right, front, back as mentioned in section 4.1.3)

$$position(O, P', T+1) \leftarrow position(O, P, T) \wedge move(O, D, T). \tag{5.1}$$

This rule models the agent's ability to move around the game world. It implies that any object $o \in O$ that is in position p and moves in direction $d \in D$ at any timestep $t \in T$ will end up in position p' at timestep $t+1$. The mapping between p and p' is based on an arbitrary mapping function $f_{NewPositionMove}(p, d)$.

We also need to model the "kicking" mechanic, where an object can kick another object and change its location.

$$position(O, P', T+1) \leftarrow position(O, P, T) \wedge kick(OO, O, up, T) \tag{5.2}$$

$$\bot \leftarrow kick(O, OO, , T) \wedge distance(O, OO, Distance, T) \wedge Distance > D_{max} \tag{5.3}$$

The variable OO refers to the object that is initiating the kick. The logic is similar to the one in eq. (5.1), where now we simply have a different mapping function between the positions p and p', $f_{NewPositionKick}$. However, an integrity constraint (eq. (5.3)) is necessary in order for the agent to be unable to kick the ball regardless of their relative position.

It becomes apparent how modelling the agent's logic using ASP can create smart agents in a very cost-efficient manner. However, the agent currently has no goal and running the above ASP program would yield an empty answer set. We now propose some goal states to motivate the ASP solver.

Now, an important question arises: What is the simplest goal an agent should have in order for their resulting behavior to resemble that of a football player? It is not far-fetched to think that a player's main objective is to get the ball as close as possible to the opponent's goal. This can be encoded in ASP in a single line of code.

$$\#minimize\{D : distance(ball, goal, D, t_{end} + 1)\}. \tag{5.4}$$

Using the modelling so far as a base, the designer can further flesh out the agent's behavior. For example, there could be a "goalkeeper" agent that tries to protect the goal and keep the ball far away from it.

$$\#maximize\&\{D : distance(ball, my\_goal, D, t_{end} + 1)\}. \tag{5.5}$$
$$\#minimize\&\{D : distance(player, my\_goal, D, t_{end} + 1)\}. \tag{5.6}$$

In essence, we are reusing eq. (5.4) with different objects as the *distance* function's arguments. The ASP solver will provide us with answer sets that optimize over all of the soft constraints.

The designer can mix and match these rules to easily create smart agents with different behaviors. We can think of the base football rules (like movement) as a separate ASP module which is combined with the rules that govern the agent's goal inside the playing field. The ASP program can be split into sub-programs using *clingo*'s `#program` directive. From inside the game engine API, the designer can choose the parts of the program to ground before commencing the solving procedure. That way, different aspects of the agent's AI can be "enabled" and "disabled" based on some conditions. This can allow for mixing and matching different agent behaviors, enabling more expressive experimentation and prototyping.

Finally, we added a feature where the agent's last "thought" is shows in the game world as a speech bubble (fig. 5.2b). This can be thought of as a debugging tool for the designer, allowing them to inspect the agent's reasoning process. It could be argued that something similar could be achieved if inside a traditional algorithm, some logging was added. However, using ASP, we get this feature "for free", without the need of adding error-prone logging code inside the agent's logic. In addition, the game developer has immediate access to the agent's last decision as a logical symbol, without having to translate some abstract structure, only relevant inside the underlying algorithm, like a graph's node into a human-readable format.

```
#program base.
% base rules ...

#program a.

#program b.

#program c.
```

Listing 11: Splitting the ASP program into modules. The programmer, from inside the game engine API can choose which parts of the program to ground.

5.2 Level Generation

In this section, we delve into another practical application of Answer Set Programming (ASP) in game development, specifically focusing on level generation. ASP offers a unique and flexible approach to generating game content, including game levels, by defining rules and constraints in a logical framework. We first try to develop a generic basis on which objects can be placed and later try to apply some semantic meaning to the objects in order to guide the solver to create appropriate artifacts. These examples provide insights into the potential of ASP as a valuable tool in game development, opening up new horizons for content creation and gameplay innovation. The characteristics of **brevity** and **emergent complexity** are present in these examples.

5.2.1 Generic Object Placement

One approach on modelling object placement is one where the designer has absolutely no preference of where to place each object, leaving it up to the solver to decide the x, y, z coordinates of each one. This approach, although it can be applicable in certain concepts, can lead to large solving times, with the majority of the time being spent on the grounding phase, because of the large amount of predicates generated. To limit grounding times, we put a limit on the solver's available space for placing objects, with a constant parameter d. The placing space will be $d \times d \times d$ units large.

The designer starts off with an asset library, which contains all the objects that can be placed. Our fact base will consist of the objects that will be placed as well as their sizes. Acquiring an object's size in the x, y, z directions is done through it's axis-aligned bounding-box (AABB) structure. This is information is enough for our solver to place the objects abiding to given constraints. Additional predicates can be added to add semantics to each

```
in(O, IX, IY, IZ) :- place(O, X, Y, Z), size(O, SX, SY, SZ),
                     X <= IX, X + SX > IX,
                     Y <= IY, Y + SY > IY,
                     Z <= IZ, Z + SZ > IZ,
                     dim(IX), dim(IY), dim(IZ).
```

Listing 12: The $in(O, X, Y, Z)$ rule defined in *Clingo*.

object and enable for more complex reasoning.

The generation part of our program consists of a single choice atom, which selects a tuple of (x, y, z) for each object to be placed.

$$\{place(O, X, Y, Z) : dim(X), dim(Y), dim(Z)\} \leftarrow object(O). \tag{5.7}$$

On our first iteration, we will simply ensure that no two objects overlap with each other. Instead of defining what an "overlap" is in our ASP program, we will use a helper predicate $in(O, X, Y, Z)$ which defines how an object O occupies a set of points when it placed.

Now, overlaps can be eliminated with a single integrity constraint (eq. (5.8)) where no two different objects can occupy the same space.

$$\bot \leftarrow in(O1, X, Y, Z) \wedge in(O2, X, Y, Z) \wedge O1 \neq O2 \tag{5.8}$$

Another constraint can be added in order to ensure that no object is floating in the air.

$$\bot \leftarrow place(O, X, Y, Z) \wedge Y > 0 \wedge 0\{in(OO, X, Y - 1, Z)\}0. \tag{5.9}$$

The constraint in eq. (5.9) states that if an object o is placed at (x, y, z) and $y > 0$ (above ground), then there must be another object oo placed at $(x, y - 1, z)$.

Continuing with modelling steadiness in our scene, it might be the case that large objects shouldn't be placed on top of smaller ones. For this, we add a $volume(o, v)$ rule which defines the volume of an object o, an $on(o1, o2)$ rule which defines that $o1$ is on top of $o2$ and a single integrity constraint.

$$volume(O, N)\& \leftarrow size(O, X, Y, Z) \wedge N = X \cdot Y \cdot Z. \tag{5.10}$$

$$on(O1, O2)\& \leftarrow in(O2, X, Y, Z) \wedge size(O2, SX, SY, SZ) \wedge in(O1, X, Y + SY, Z). \tag{5.11}$$

$$\bot\& \leftarrow on(O1, O2) \wedge volume(O1, V1) \wedge volume(O2, V2) \wedge V1 \geq V2. \tag{5.12}$$

```
#const n=5.

cellType(grass;wood;water;lava).

x(1..n).
y(1..n).
```

Listing 13: The grid size and cell types defined in *Clingo*.

After generating some placements, we can visualize the results in a 3D environment. For visualization, we use the *Godot* game engine.

5.2.2 Tile Level Terrain Generation

When it comes to terrain generation, most games usually follow an approach where some kind of noise function like Perlin noise [84] is used to determine the type of terrain to be placed at some specific (x, y) location based on the noise function's value. This approach has seen major adoption in the game industry as it is used in major titles like *Minecraft* [77]. However, this approach although performant, does not allow for high levels of controllability. For example, a designer cannot specify that they want a certain number of mountains to be placed, or a river that flows through a specific area. Using ASP, we can generate natural-looking terrain using a highly expressive language.

Modelling

First, we'll build an ASP program which decides the type of tile to be placed on a 2D grid. We need t define a grid size and the cell types that our generator will use.

Assigning a cell type to each cell is done by using a single choice rule. The predicate $cell(X, Y, T)$ assigns the cell type T to the cell at (X, Y).

$$1\{cell(X, Y, T) : cellType(T)\}1 \leftarrow x(X), y(Y). \tag{5.13}$$

The above generation is semantically identical to placing cells randomly. To add more meaning to our generation, we can add some constraints. First, we'll add a helper predicate $adjacent(X1, Y1, X2, Y2)$ which defines how two cells are adjacent to each other.

When lava touches water, it turns into stone. One way for our generator to respect this, is to add a constraint where a lava tile cannot be placed adjacent to a water tile.

```
adjacent(X, Y, X + 1, Y) :- x(X), y(Y).
adjacent(X, Y, X - 1, Y) :- x(X), y(Y).
adjacent(X, Y, X, Y + 1) :- x(X), y(Y).
adjacent(X, Y, X, Y - 1) :- x(X), y(Y).
```

Listing 14: The $adjacent(X1, Y1, X2, Y2)$ predicate defined in *Clingo*.

$$\bot \leftarrow cell(X1, Y1, lava) \wedge cell(X2, Y2, water) \wedge adjacent(X1, Y1, X2, Y2). \tag{5.14}$$

What if we want to generate a map with a river flowing through it? With the help of the $adjacent(X1, Y1, X2, Y2)$ predicate, we can define a $connected(X1, Y1, X2, Y2)$ predicate. Two tiles are considered connected if they are of the same type while also being adjacent to each other or connected to a third tile.

$$\begin{aligned} connected(X, Y, X1, Y1)\& &\leftarrow adjacent(X, Y, X1, Y1) \wedge cell(X, Y, T) \wedge cell(X1, Y1, T). \\ connected(X, Y, X1, Y1)\& &\leftarrow connected(X, Y, X2, Y2) \wedge connected(X1, Y1, X2, Y2). \end{aligned} \tag{5.15}$$

Finally, we can add a constraint where a river that runs from the top of the map to the bottom is generated.

$$\bot \leftarrow\sim connected(1, 1, n, n). \tag{5.16}$$

The described approach can be extended further to generate maps with more complex constraints.

Similar to the work in [19] where the authors divide the to-be-generated region into smaller sections and apply an ASP-based generator to each one separately. This can allow for the creation of larger maps without the solving times becoming unreasonably long. We follow a similar approach, where we generate multiple $n \times n$ (where $n \leq 7$) grids and "stitch" them together to create a larger map. We also apply what we call "outer loop randomization", where the parameters of the ASP solver per sub-grid are also randomized, in order to produce more diverse results.

Our final terrain generator consists of a *Python* script which calls the ASP solver *Clingo*, getting the generated map as an answer set, matching each tile type as a color. A final image is generated using the *Image Magick*[1] library, which represents the map as a grid of colored cells. Our final implementation can be found in appendix B.

[1] https://imagemagick.org/

Running Time

The solution space for the terrain generator can become large as the size of the grid increases. The total number of possible solutions is given by the following equation.

$$\text{Number of possible solutions} = \prod_{i=1}^{n}\prod_{j=1}^{n}\text{Number of cell types} = O(|T|^{n^2}) \tag{5.17}$$

Where T is the set of cell types and n is the size of the grid.

The running time of the terrain generator is highly dependent on the size of the grid. The *Clingo* solver allows us to enable parallel solving by using the `parallel-mode` flag. However, even with parallel solving enabled, larger grid sizes can take a long time to generate. To get over the large solving time constraints, we can use a divide and conquer approach, where we split the grid into smaller sub-grids.

When we apply our partitioning approach, the complexity of the generator becomes linear to the number of sub-grids ($O(N_s)$).

It can be seen that the total time for generating the entire grid increases exponentially. This is why applying the partitioning technique is essential if we wanted to generate larger maps. However, if we look at the generated terrain not being a tile by tile representation of our world but rather an instance where each square represents a different biome, we can see that ASP can be utilized to generate higher-level artifacts. For example, in our game we could have the logic determining what kind of biome a chunk of our world is written in ASP and apply a more traditional procedural generation technique to the tile by tile generation of each separate chunk.

5.3 Goal-Oriented Room Traversal

Our goal in this section is to create an agent which, given a goal will try to formulate a sequence of actions to achieve it. We will employ a modelling which is based on the theory of event calculus which was discussed in section 3.1.4. This is an example of high-level reasoning, where the agent formulates a long term plan consisting of high level actions, rather than some short term reactive behavior which results in immediate action. The characteristics of **relatively small solution space** and **emergent complexity** are present in this example.

We will create an agent who can move from one room to another and pick up objects. Our example is inspired from the work in [68].

Our code to make this agent functioning includes the following modules

1. The *Discrete Event Calculus* axioms (appendix A.1) module
2. Our problem's domain logic module

5.3.1 Actions and Events

As with our other examples, we need to encode some axioms which are derived from the modelling of our specific problem. An example domain dependent theory is the axiomatization of our scenario where a simple agent can move from room to room, pick up and let go of objects. We will use the two fluents $inRoom(o, r)$ (where object o is inside room r) and $holding(a, o)$ (where agent a holds object o) and three actions $walk(a, r_1, r_2)$, $pickUp(a, o)$, $letGoOf(a, o)$.

All the possible events/fluents are generated with the following rules

$$event(letGoOf(A, O))\& \leftarrow agent(A) \wedge object(O). \tag{5.18}$$

$$event(pickUp(A, O))\& \leftarrow agent(A) \wedge object(O). \tag{5.19}$$

$$event(walk(A, R1, R2))\& \leftarrow agent(A) \wedge room(R1) \wedge room(R2) \wedge R1 \neq R2. \tag{5.20}$$

$$fluent(inRoom(O, R))\& \leftarrow object(O) \wedge room(R). \tag{5.21}$$

$$fluent(holding(A, O))\& \leftarrow agent(A) \wedge object(O). \tag{5.22}$$

Our rules and integrity constraints will need to model certain aspects of our world. Most of these will seem obvious, but they still need to be encoded in our logic program since the *Event Calculus* axioms can only cover the parts of our theory relating to the temporal relationship between the events in our world and not topological relationships such as the simple fact that an object can not exist in two places at the same time. Once these rules are in place, however, we can have a robust

- An object can not be in two (or more) rooms at the same time.
- An agent that moves from room R_1 to room R_2 stops being in room R_1.
- An agent can pick up an object only if they and the object are in the same room.
- If an agent is holding one or more objects and moves to another room, the objects will move room as well.

- If an agent lets go of an object, they stop holding it.

With these rules inside our logic program, we can create agents that can make sound decisions inside our logic world.

5.3.2 Agent Choice

One of the interesting parts of this approach is that we separate the existence of an action predicate to whether it actually happened. For an action/event e to have been realized at time t the predicate $happens(e, t)$ needs to be true. This enables us to easily encode the agent's actions in a single choice rule.

$$1 \{ happens(E, T) : event(E) \} 1 \leftarrow time(T). \tag{5.23}$$

Our agent chooses an action to do for every time step t
The agent's goal is formulated as follows in first-order logic

$$goal \leftarrow holdsAt(inRoom(book, kitchen), maxtime) \\ \wedge holdsAt(inRoom(john, livingRoom), maxtime). \tag{5.24}$$

The rule in eq. (5.24) entails that our goal is for the agent "bob" to be inside the *livingRoom* and for the "book" to be inside the *kitchen* at time $t = maxtime$.

We run the program for `maxtime` $= 5$ time steps and format the output[2].

5.3.3 Locked Door Problem

Following, we have a slightly more complicated version of the problem, where the door between the two rooms is locked. A key is placed with the agent in the living room and the agent must pick up the key, unlock the door and then move the book. The final goal remains the same, but the agent needs to perform more actions to achieve it.

We need to modify one of the existing axioms to model the fact that a door can be locked and also add a new event $unlock(a, r)$ where agent a unlocks room r. The relevant code can be found in appendix B. With minimal change, our program can now handle more complicated problem domains.

5.3.4 Results

The agent is capable of formulating a plan to achieve its goal. The plan for the first problem instance is as follows

[2]The `format-output` utility can be found in https://decreasoner.sourceforge.net/csr/ecasp/index.html

1. Go from the kitchen to the living room
2. Pick up the book
3. Go to the kitchen
4. Let go of the book
5. Go back to the living room

The plan for the second problem instance

1. Pick up the key
2. Unlock the living room door
3. Go from the kitchen to the living room
4. Pick up the book
5. Go to the kitchen
6. Let go of the book
7. Let go of the key[3]
8. Go back to the living room

We can see that our agent is able to figure out that they must move the book from one room into another, in a discrete sequence of actions. Had we wanted the agent to execute an optimal plan, we only need to add a predicate that counts the number of actions and minimize over it.

$$eventCount(N)\& \leftarrow N = \#count\{E, T : happens(E, T)\}. \tag{5.25}$$
$$\#minimize\&\{N : eventCount(N)\}. \tag{5.26}$$

5.4 Study Participant Projects

In this section, we present the projects the study participants completed during the evaluation of our framework. Details of the participants and the study can be found in section 4.3.

During our evaluation study, participants, after being introduced to the ASP paradigm, were asked to design a game mechanic using the proposed

[3]This action is optional since the agent holding the key is not part of the goal formulation.

Application & Description	Time	Iterations	Design Characteristic
Wind Simulator & Simulate wind direction in grid.	2.5h	3	**B, E**
Loot Generator & Generates reward combinations.	2h	1	**B, S**
Conversation Agent & Simulate a conversation.	1h	3	**S, E**
Traversing Agent & Agent that can navigate a 2D space.	1h	2	**B**
Level Generator & Generates game levels	1.5h	2	**B, E**
Ball Sort Solver & Solves/Generates puzzles.	1h	2	**B, S**
Futoshiki Solver & Solves/Generates puzzles.	0.5h	2	**B, S**
Level Generator & Generate levels with controllable difficulty.	3h	6	**B, E**

Table 5.1: The applications created by the participants during the study and the design characteristics which they satisfy for ASP suitability. These are brevity (**B**), relatively small solution space (**S**) and emergent complexity (**E**). The assignment of characteristics is done based on the comparison of the resulting application with the characteristics described in section 4.2.1.

framework. Here we present the projects that were created by the participants and briefly comment on them and how they relate to our framework.

Despite, the small number of subjects, our assessment gave concrete results on the strengths and weaknesses of the proposed workflow. During the study, participants created a wide range of applications. Most participants gravitated towards creating generation programs rather than agent behavior mechanics. However, there was still a wide variety of creations.

- **Wind Simulator & Loot Generator**: The Wind Simulator and Loot Generator applications demonstrate the efficiency of ASP implementation, with relatively short development times of 2.5 hours and 2 hours, respectively. Both applications satisfy the design characteristic of brevity (B), showcasing the concise nature of their ASP solutions. Additionally, the Loot Generator aligns with the relatively small solution space (S) characteristic, while the Wind Simulator showcases emergent complexity (E) through its simulation of wind direction in a grid, resulting in complex outputs as small adjustments to the initial conditions of the grid (placement of wind sources, mountains that block air flow) can significantly affect the final result.

- **Conversation Agent & Traversing Agent**: The Conversation Agent and Traversing Agent applications exemplify the versatility of ASP in different domains. The Conversation Agent simulates conversations using ASP, while the Traversing Agent focuses on agent-based navigation in a 2D space. Both applications showcase efficient implementation, with development times of 1 hour and 1 hour, respectively. The Traversing Agent satisfies the brevity (B) characteristic, while the Conversation Agent exhibits both the relatively small solution space (S) and emergent complexity (E) characteristics.

- **Level Generator & Level Generator with Controllable Difficulty**: The Level Generator and Level Generator with Controllable Difficulty applications highlight the use of ASP in generating game levels. The Level Generator creates game levels efficiently within 1.5 hours, while the extended version allows for customizable difficulty and requires more development time (3 hours) and iterations (6). However, time per iterations remains low. Both applications fulfill the brevity (B) characteristic, and they demonstrate emergent complexity (E) through the generation of diverse and dynamic game levels.

- **Ball Sort Solver & Futoshiki Solver**: The Ball Sort Solver and Futoshiki Solver applications utilize ASP to solve and generate puzzles. Developed in 1 hour and 0.5 hours, respectively, both applications showcase the efficiency of ASP implementation. They satisfy the brevity (B) and relatively small solution space (S) characteristics, highlighting the suitability of ASP for puzzle-solving scenarios.

5.4.1 Participant Feedback

After participants completed their application, a short interview was conducted. The interview consisted of a series of questions regarding the participant's opinion on the applicability of the proposed workflow.

Would you use ASP again to create a game mechanic or content generator? Most participants answered positively to this question, with most mentioning that it would more plausible that they would use ASP in more logic-heavy and decision-making applications. All participants concluded that ASP can be very powerful, but that it requires a lot of practice to master.

-- *Participant 1*: "I believe it would take me a long time to learn the language. The syntax is strange, but I can create a mental model of how it works."

-- *Participant 5*: "I would use it again in simple scenarios."

-- *Participant 7*: "I would use Clingo again if what I wanted to build involved logic and decision making."

What did you find challenging about applying the proposed workflow and how could it be improved? A common issue that participants faced was the unusual syntax that *Clingo* uses. Some participants mentioned that the lack of a debugger made it difficult to find errors in their programs, especially in later iterations where our programs consisted of multiple rules and integrity constraints. One participant mentioned that the lack of a visual

representation of the output of the program made it difficult to understand the solutions produced. One participant gave the suggestion of "[...] a similar language that is specifically made for game design.", having premade axiomatic rules for common game mechanics and structures.

-- *Participant 7*: "It would be nice to have a graphical interface that shows how the solver arrives at solutions."

-- *Participant 8*: "[The workflow] could be improved if Clingo had better syntax. Maybe an abstraction layer built on top of it."

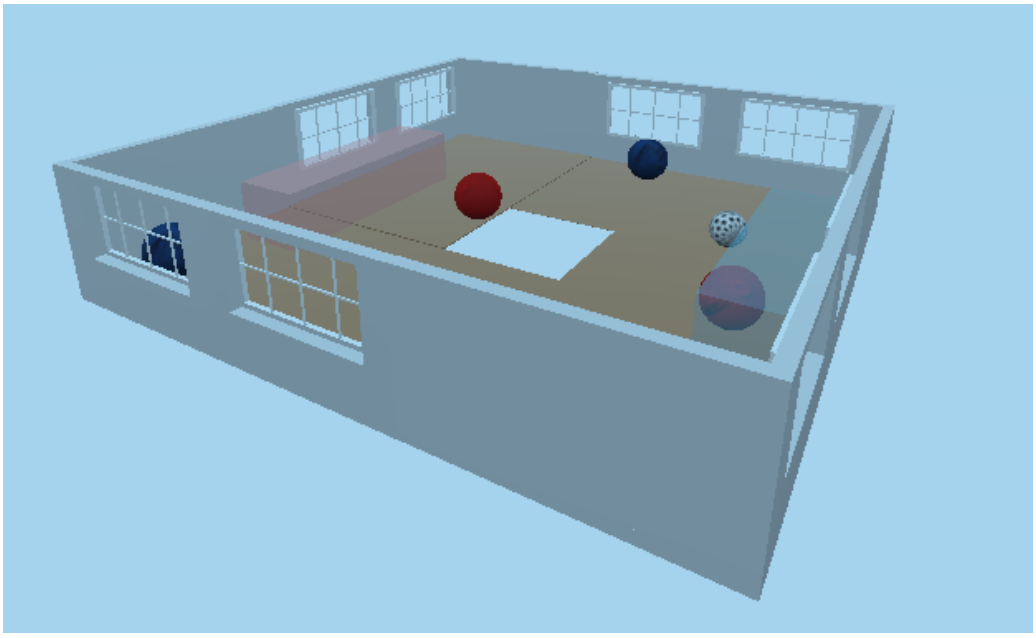
Does the proposed workflow provide design inspiration for developing games? This question was also answered positively. One participant with extensive experience in game development said that this workflow "[...] gives you the ability to create entirely new game mechanics that you wouldn't bother developing otherwise because of the programming difficulty". Most participants mentioned that the value of ASP lies in its ability to solve problems briefly and concisely, in comparison to traditional programming languages. One participant mentioned that "[...] it is easier for someone that is not a programmer to express their logic with constraints".

One participant who was familiar with cybersecurity but unfamiliar with ASP and game development mentioned that the paradigm could be used to make high level decisions applying attacks using *Metasploit* [78], a penetration testing framework.

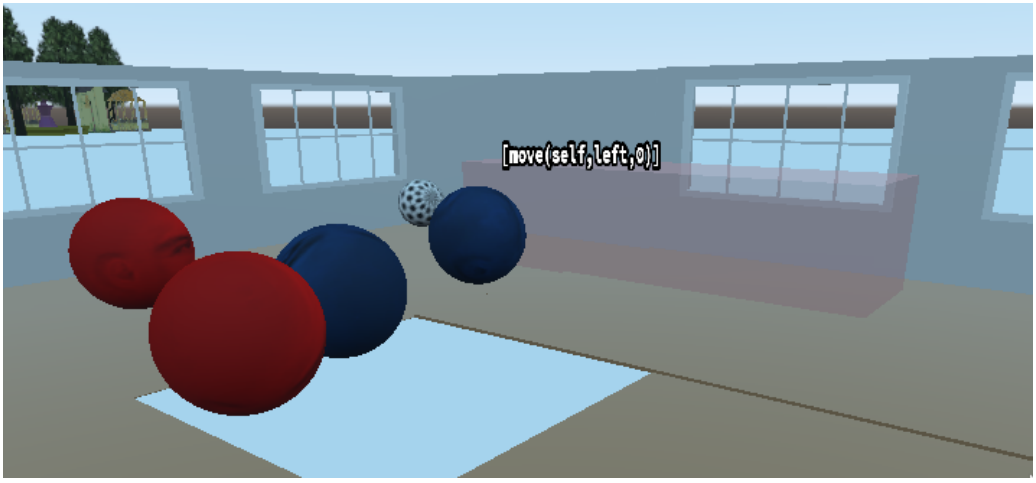
-- *Participant 1*: "It gives you the ability to create entirely new game mechanics that you wouldn't bother developing otherwise because of the programming difficulty."

-- *Participant 5*: "[A game developer] might say something like "Oh, this can be easily encoded using rules". Now, it's easier to think of a game mechanic and come up with constraints to create it."

In section 4.2 we presented a few general characteristics that constitute a game mechanic as suitable for being implemented in ASP. Those were **B**revity of the rules being used in final model, a relatively **S**mall solutions space and **E**mergent complexity. Most of the projects developed by the participants were able to satisfy at least two of these characteristics.



(a) A screenshot of the football playing field. It consists of two goals and four agents, two on each team. The ball colliding with one of the semi-transparent rectangles will result in a point being scored for the team opposite of the goal.



(b) The football players. Floating text above the agent indicates the agent's latest "thought", which is the action it will perform next. In this image the predicate $move(self, left, 0)$ is part of the optimal answer set produced by the ASP solver, meaning that in the next time-step, the agent will move to the left.

Figure 5.2: Screenshots of our football game implementation.

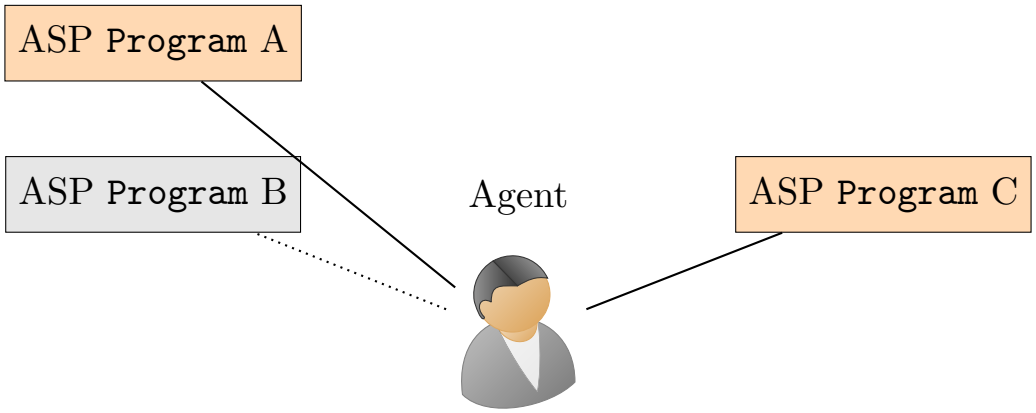


Figure 5.3: Changing the agent's behavior by enabling/disabling ASP modules.

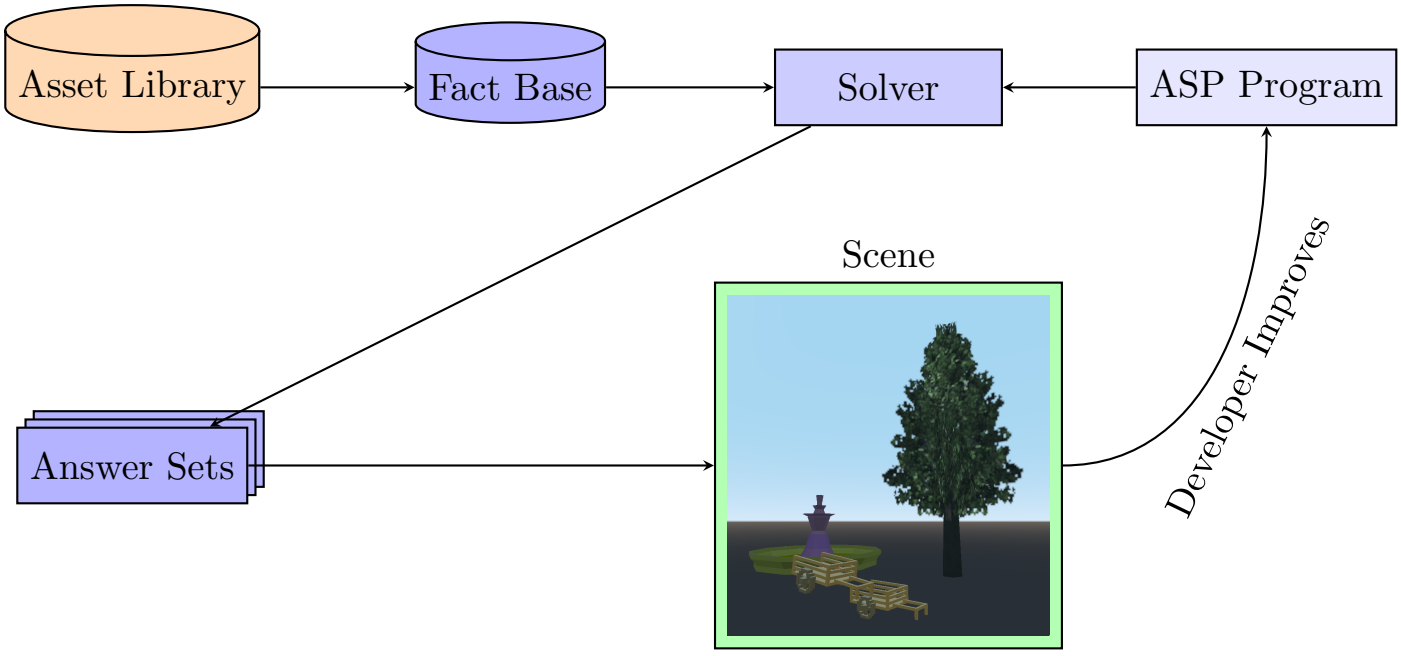
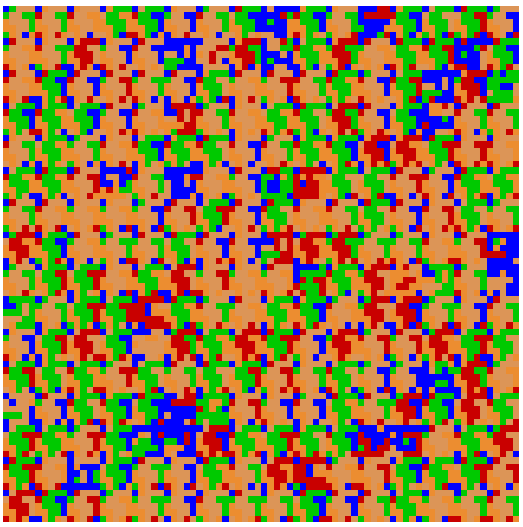
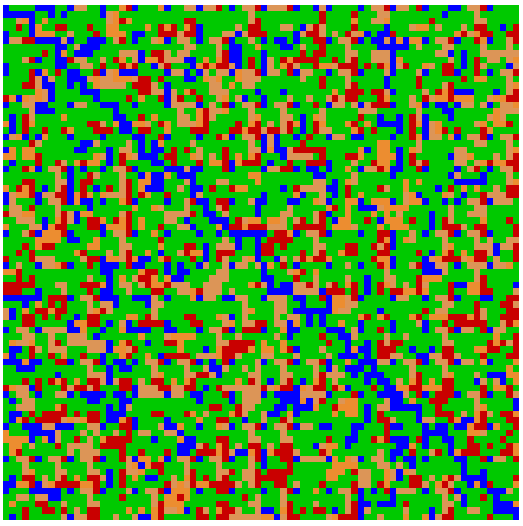


Figure 5.4: The workflow of our level generation system. An asset library consists of all the 3D assets the designer would like to place inside the scene. These assets are placed inside the *Fact Base* as predicates of the form $object(O)$. $size(O, X, Y, Z)$. These facts together with an ASP program are fed into the solver, which will generate answer sets. Answer sets which include facts of the form $place(O, X, Y, Z)$ are then interpreted by the game engine, placing the assets inside the scene. After the scene is generated, the designer can improve the ASP program, adding more constraints or changing the existing ones, iterating over the process.



(a) A generated map with no constraints encoded.



(b) A generated map with a river flowing through it and no water and lava tiles touching (inside the same sub-grid).

Figure 5.5: Examples of maps generated using our terrain generator.

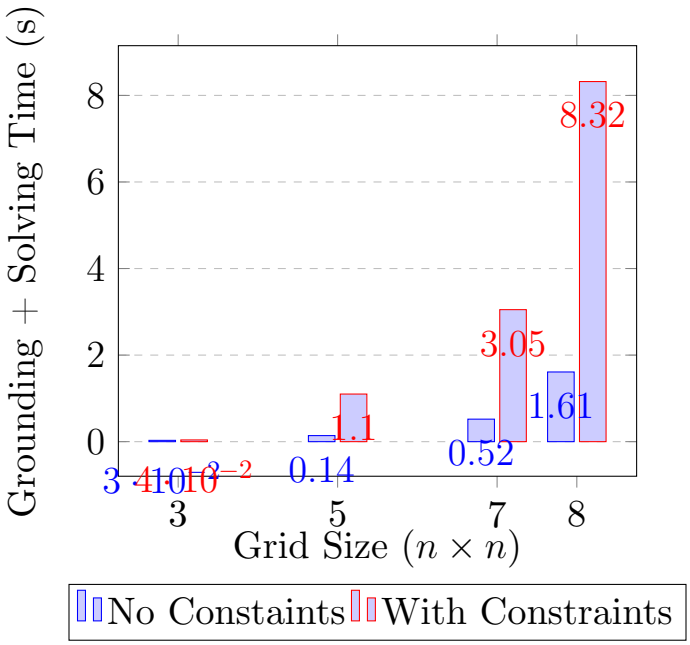
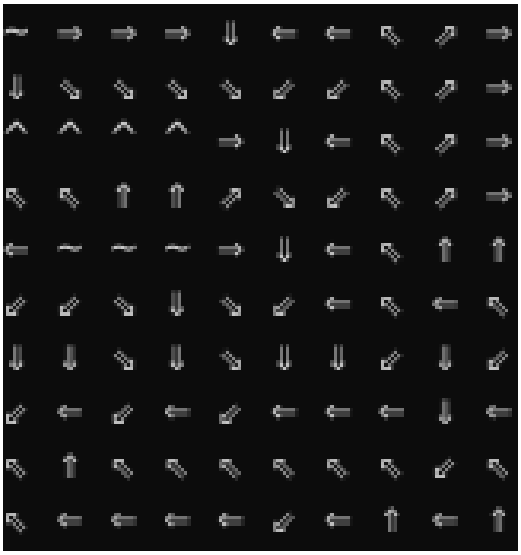
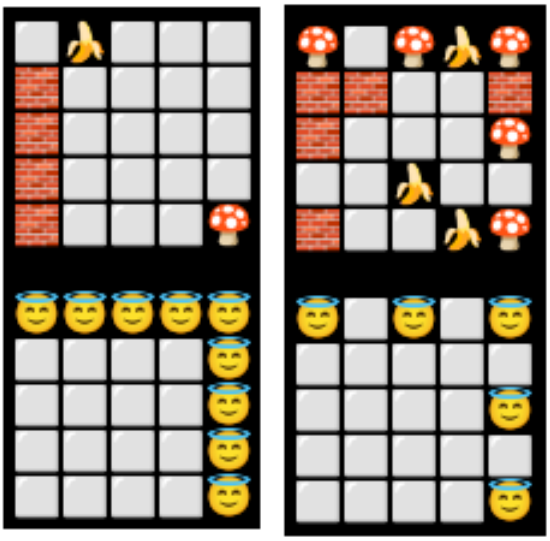


Figure 5.6: Indicative average generation times for different grid sizes. Comparison is made between the solving + grounding times with and without constraints. Measurements were made on a machine with an Intel Core i7-4770 CPU @ 3.40GHz.



(a) Wind direction simulator. Arrows designate wind direction while ^ and ~ characters designate mountains and wind sources respectively.



(b) Two instances of the level generator with difficulty control. Above is the level generated and below the optimal solution. Image to the right is the lower difficulty level.

(c) Four instances of the level generator

```
in_chest(diamond,3)
in_chest(gold,1)
in_chest(porkchop,9)

in_chest(diamond,3)
in_chest(gold,3)
in_chest(porkchop,1)

in_chest(diamond,3)
in_chest(gold,3)
in_chest(porkchop,9)
```

(d) Three instances of the loot generator.

Figure 5.7: The applications created by the participants during the study. A single iteration is defined as the process of inspecting the program's produced answer sets and making one or more significant modifications to it before the process is repeated.

6. Conclusions

6.1 Summary

In this thesis we have provided some formal design heuristics and practical implementations concerning the application of Answer Set Programming to game development. Our proposals stem from the need of robust high level programming interfaces to assist with the development of complex applications like video games.

- We provided a software framework for integrating ASP solvers into game engines, taking into account the specific requirements of the domain such as real-time performance and the need for modularity and extensibility. We explained how the game state can be mapped into ASP as collections of facts and rules commenting on compromises that need to be made for the approach to be viable in real applications. In addition, we presented auxilary tools and libraries for augmenting the developer experience when working with ASP.

- We proposed a set of design heuristics for recognising parts where game logic can be elegantly expressed using ASP. These heuristics are based on the analysis of the related work and serve as guidelines for developers and designers to determine when and how to use ASP in their projects.

- We proposed a methodology for authoring ASP programs in a way that is accessible to non-expert users. This is achieved by providing a number of distinct steps based on guidelines provided by ASP experts and tailored to the context of game development.

- We implemented a number of demos and presented them in tutorial fashion, demonstrating how the proposed methodology can be used to solve a variety of problems. The applications that we developed ranged from multi agent game playing systems to procedural content generation. In these examples we also commented on the limitations of the approach, mainly in the realm of performance, and provided suggestions on how to overcome them.

- We organised an empirical user study to evaluate the effectiveness of the proposed methodology. We asked participants to design and implement a game mechanic, with the help of the researcher. Then, an interview was conducted to gather feedback on the experience. The results of the study indicate that the proposed methodology is effective in assisting game developers in quickly implementing game logic while also enabling rapid game prototyping. The method's effectiveness by non-expert users, however, is limited by the idiosyncrasies of the ASP paradigm.

6.2 Future Work

There is a plethora of interesting extensions and continuations of our work, a number of which have been already discussed thtoughout this thesis. In this section we briefly summarize some of the more important avenues of future research and applications.

- *Clingo*'s input language, although concise and in-line with other logic programming languages, poses a steep learning curve to aspiring users. A language built on top of it where the syntax resembles one of traditional programming languages while sustaining the program's semantic properties would be stepping stone towards wider adoption of the paradigm. Such work has been made in [6] where a *Lisp* programming language dialect is given a *Python*-like syntax through a trans-pilation process.

- High performance ASP solvers are a necessity for the approach to be viable in larger scale applications. Taking advantage of GPU parallelism is a promising avenue of research, as demonstrated by [28]. Such advances would be especially useful in the context of video games, where it is not uncommon for users to have access to high performance GPU hardware.

- Finally, future work should involve the application of ASP in larger-scale video game projects, exploring how the proposed workflow can fit into long running game development cases.

Bibliography

[1] Andrade, A. Game engines: a survey. *EAI Endorsed Transactions on Game-Based Learning Endorsed Transactions on Game-Based Learning 2*, 6 (nov 2015), 150615.

[2] Angilica, D., Ianni, G., Lisi, F. A., and Pulina, L. AI and videogames: a "drosophila" for declarative methods. In *Proceedings of the 10th Italian workshop on Planning and Scheduling (IPS 2022), RCRA Incontri E Confronti (RiCeRcA 2022), and the workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (SPIRIT 2022) co-located with 21st International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022), November 28 - December 2, 2022, University of Udine, Udine, Italy* (2022), R. D. Benedictis, N. Gatti, M. Maratea, A. Micheli, A. Murano, E. Scala, L. Serafini, I. Serina, A. Umbrico, and M. Vallati, Eds., vol. 3345 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[3] Angilica, D., Ianni, G., and Pacenza, F. Declarative AI design in Unity using Answer Set Programming. In *2022 IEEE Conference on Games (CoG)* (aug 2022), IEEE.

[4] Antonova, E. Applying Answer Set Programming in Game Level Design. Master's thesis, Aalto University, 2015.

[5] Aravanis, T., Demiris, K., and Peppas, P. Legal Reasoning in Answer Set Programming. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)* (nov 2018), IEEE.

[6] Babenhauserheide, A. Wisp: Whitespace to lisp, 2013.

[7] Bayer, M. SQLAlchemy: The Database Toolkit for Python, 2023.

[8] Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. *Handbook of Satisfiability* (2009), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press.

[9] Boenn, G., Brain, M., Vos, M. D., and ffitch, J. Automatic music composition using answer set programming, 2010.

[10] Bowman, S. R. Eight Things to Know about Large Language Models, 2023.

[11] Brain, M., Cliffe, O., and De Vos, M. *A pragmatic programmer's guide to answer set programming.* Sept. 2009. Software Engineering for Answer Set Programming (SEA09) ; Conference date: 14-09-2009.

[12] Bratko, I. *Prolog programming for Artificial Intelligence.* Pearson education, 2012.

[13] Burris, S., and Sankappanavar, H. P. *A Course in Universal Algebra.* Graduate Texts in Mathematics. Springer, Berlin, Germany, 1981.

[14] Business Reporter. Trial and Error: The Human flaws in Machine Learning. *Business Reporter* (2020).

[15] Cabalar, P., Fandinno, J., and Fink, M. Causal graph justifications of logic programs, 2014.

[16] Cabalar, P., Fandinno, J., and Muñiz, B. A System for Explainable Answer Set Programming. *Electronic Proceedings in Theoretical Computer Science 325* (sep 2020), 124--136.

[17] Cabalar, P., Kaminski, R., Morkisch, P., and Schaub, T. telingo = ASP + Time. In *International Conference on Logic Programming and Non-Monotonic Reasoning* (2019).

[18] Calimeri, F., Fink, M., Germano, S., Humenberger, A., Ianni, G., Redl, C., Stepanova, D., Tucci, A., and Wimmer, A. Angry-HEX: An Artificial Player for Angry Birds Based on Declarative Knowledge Bases. *IEEE Trans. Comput. Intell. AI Games Transactions on Computational Intelligence and AI in Games 8*, 2 (jun 2016), 128--139.

[19] Calimeri, F., Germano, S., Ianni, G., Pacenza, F., Pezzimenti, A., and Tucci, A. Answer Set Programming for Declarative Content Specification: A Scalable Partitioning-Based Approach. 225--237.

[20] Cant, R., and Langensiepen, C. Methods for Automated Object Placement in Virtual Scenes. In *2009 11th International Conference on Computer Modelling and Simulation* (2009), IEEE.

[21] Chaudhri, V. Answer Set Programming, 2022.

[22] Contributors, G. E. Gdextension docs, 2021.

[23] Contributors, G. E. Gdnative docs, 2021.

[24] Costantini, S. Integrating Answer Set Modules into Agent Programs. 613--615.

[25] Das, I. Why Artificial Stupidity is the Next Frontier of Intelligence.

[26] Degica. Rpg maker, 2021.

[27] Dennett, D. Cognitive wheels: The frame problem of ai.

[28] Dovier, A., Formisano, A., and Vella, F. Gpu-based parallelism for asp-solving, 2019.

[29] Drepper, U. How to write shared libraries.

[30] Eiter, T., Faber, W., Leone, N., and Pfeifer, G. Declarative Problem-Solving Using the DLV System. In *Logic-Based Artificial Intelligence*. Springer US, 2000, pp. 79--103.

[31] Eiter, T., Ianni, G., and Krennwallner, T. Answer Set Programming: A Primer. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 40--110.

[32] Fan, X., Wu, J., and Tian, L. A Review of Artificial Intelligence for Games. In *Lecture Notes in Electrical Engineering*. Springer Singapore, 2020, pp. 298--303.

[33] Fuscà, D., Germano, S., Zangari, J., Calimeri, F., and Perri, S. Answer set programming and declarative problem solving in game AIs. *CEUR Workshop Proceedings 1107* (01 2013), 81--88.

[34] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. A user's guide to gringo, clasp, clingo, and iclingo, 2015.

[35] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. Theory Solving Made Easy with Clingo 5. In *International Conference on Logic Programming* (2016).

[36] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[37] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Clingo = ASP + control: Preliminary report. *CoRR abs/1405.3694* (2014).

[38] Gebser, M., Kaminski, R., and Schaub, T. aspcud: A linux package configuration tool based on answer set programming. *Electronic Proceedings in Theoretical Computer Science 65* (aug 2011), 12--25.

[39] Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. clasp: A Conflict-Driven Answer Set Solver. In *Logic Programming and Nonmonotonic Reasoning*. Springer Berlin Heidelberg, pp. 260--265.

[40] Gebser, M., Kaufmann, B., and Schaub, T. Conflict-driven answer set solving: From theory to practice. *Artif. Intell. 187* (2012), 52--89.

[41] Gelfond, M. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond*. Springer Berlin Heidelberg, 2002, pp. 413--451.

[42] Gelfond, M., and Lifschitz, V. The Stable Model Semantics For Logic Programming. *Logic Programming 2* (12 2000).

[43] Genesereth, M. R., Nilsson, N. J., and Pell, B. General game playing: Overview of the AAAI competition. *AI Magazine 26*, 2 (jun 2005), 62--72.

[44] Giunchiglia, E., Lierler, Y., Maratea, M., and Tacchella, A. Experiments with SAT-based Answer Set Programming. In *Search and Logic: Answer Set Programming and SAT, LaSh-06, A Workshop affiliated with ICLP, as part of FLoC* (2006).

[45] Godot Engine Team. Godot Architecture, 2022.

[46] Godot Engine Team. Godot Design Philosophy, 2022.

[47] Godot Engine Team. Godot Engine, 2023.

[48] Goled, S. Battle of Game Engines: Godot vs Unity.

[49] Gryz, J. The Frame Problem in Artificial Intelligence and Philosophy. *Filozofia Nauki 21* (06 2013), 15--30.

[50] Halst, M. V. EA SPORTS FIFA and beyond: An illustrated history of soccer video games.

[51] Hayes, B. The post-oop paradigm. *American Scientist 91*, 2.

[52] Hughes, J. Why Functional Programming Matters. *Computer Journal 32*, 2 (1989), 98--107.

[53] Hölldobler, S., and Schweizer, L. Answer Set Programming and CLASP - A Tutorial. In *Proceedings of the Young Scientists' International Workshop on Trends in Information Processing (YSIP) Colocated with the Sixth International Conference on Infocommunicational Technologies in Science, Production and Education (INFOCOM-6),*

Stavropol, Russian Federation, April 22-25, 2014 (2014), S. Hölldobler, A. Malikov, and C. Wernhard, Eds., vol. 1145 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 77--95.

[54] Ireland, C., Bowers, D., Newton, M., and Waugh, K. Understanding object-relational mapping: A framework based approach. *Int J Adv Softw 2* (01 2009).

[55] Isla, D. Halo 3 Objective Trees: A Declarative Approach to Multiagent Coordination, 2001.

[56] Ja´skiewicz, G. Prolog-Based Reasoning Layer for Counter-Strike Agents.

[57] Kagenashi. Godot Behavior Tree Asset, 2022.

[58] Kaminski, R., and Schaub, T. On the Foundations of Grounding in Answer Set Programming. *Theory and Practice of Logic Programming* (jul 2022), 1--60.

[59] Karth, I., and Smith, A. M. WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. *IEEE Transactions on Games 14*, 3 (2022), 364--376.

[60] Kaufmann, B., Leone, N., Perri, S., and Schaub, T. Grounding and Solving in Answer Set Programming. *AIMag Magazine 37*, 3 (oct 2016), 25--32.

[61] Khalifa, A., Green, M. C., Perez-Liebana, D., and Togelius, J. General video game rule generation. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (aug 2017), IEEE.

[62] Klimas, C. Twine, 2021.

[63] Korte, B., and Vygen, J. *Combinatorial optimization*, 4 ed. Algorithms and Combinatorics. Springer, Berlin, Germany, Oct. 2007.

[64] Kosnik, B. Dynamic shared objects: Survey and issues, 2006.

[65] Kowalski, R., and Sergot, M. A Logic-Based Calculus of Events. In *Topics in Information Systems*. Springer Berlin Heidelberg, 1989, pp. 23--55.

[66] kubecz3k. Godot Finite State Machine Asset, 2022.

[67] Lamprou, E., and Fidas, C. Investigating applicability heuristics of answer set programming in game development: Use cases and empirical study. Under Review.

[68] Lee, J., and Palla, R. Classical logic event calculus as answer set programming. pp. 119--133.

[69] Lee, J., and Palla, R. Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming. *CoRR abs/1401.4607* (2014).

[70] Lifschitz, V. *Circumscription*. Oxford University Press, Inc., USA, 1994, p. 297–352.

[71] Lin, F., and Zhao, Y. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157*, 1-2 (aug 2004), 115--137.

[72] Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., He, H., Li, A., He, M., Liu, Z., Wu, Z., Zhu, D., Li, X., Qiang, N., Shen, D., Liu, T., and Ge, B. Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models, 2023.

[73] Maier, D. Representing database programs as objects. In *Advances in database programming languages*. 1990, pp. 377--386.

[74] McCarthy, J. Circumscription—A form of non-monotonic reasoning. *Artificial Intelligence 13*, 1 (1980), 27--39. Special Issue on Non-Monotonic Logic.

[75] McCarthy, J., and Hayes, P. J. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, 1969, pp. 463--502. reprinted in McC90.

[76] Miller, R., and Shanahan, M. Some Alternative Formulations of the Event Calculus. In *Computational Logic: Logic Programming and Beyond*. Springer Berlin Heidelberg, 2002, pp. 452--490.

[77] Minecraft Wiki contributors. Minecraft Noise Generator, 2023.

[78] Moore, M. Penetration testing and metasploit.

[79] Mueller, E., and Sutcliffe, G. Reasoning in the event calculus using first-order automated theorem proving. pp. 840--841.

[80] Neufeld, X. Procedular Level Generation with Answer Set Programming for General Video Game Playing. Master's thesis, Institute of Knowledge and Language Engineering, Otto-von-Guericke University Magdeburg, 2015.

[81] Niemelä, Ilkka, Simons, Patrik, Syrjänen, and Tommi. Smodels: A System for Answer Set Programming. *CoRR cs.AI/0003033* (03 2000).

[82] Orkin, J. Three States and a Plan: The AI of F.E.A.R. GDC 2006.

[83] Papadimitriou, C. H. *Computational complexity.* Addison-Wesley, 1994.

[84] Perlin, K. Improving noise. *ACM Trans. Graph. Transactions on Graphics 21*, 3 (jul 2002), 681--682.

[85] Rajaratnam, D. clorm: A Python ORM-like interface for the Clingo Answer Set Programming (ASP) reasoner, 2023.

[86] Ribeiro, T., Inoue, K., and Bourgne, G. Combining Answer Set Programs for Adaptive and Reactive Reasoning.

[87] Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., and Leone, N. Team-building with answer set programming in the gioia-tauro seaport, 2011.

[88] Ritchie, D., Wang, K., and Lin, Y.-a. Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models, 2018.

[89] Schaul, T. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (2013), pp. 1--8.

[90] Scott, A., Fujii, K., and Onishi, M. How does ai play football? an analysis of rl and real-world football strategies, 2021.

[91] Seversky, L. M., and Yin, L. Real-time automatic 3D scene generation from natural language voice and text descriptions. In *Proceedings of the 14th ACM international conference on Multimedia* (oct 2006), ACM.

[92] Shanahan, M. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, Cambridge, MA, USA, 1997.

[93] Shanahan, M. The Event Calculus Explained. In *Artificial Intelligence Today.* Springer Berlin Heidelberg, 1999, pp. 409--430.

[94] Sherrod, A. *Data structures and algorithms for game developers.* Cengage Learning, South Melbourne, VIC, Australia, 2007.

[95] Shinya, M., and Forgue, M.-C. Laying out objects with geometric and physical constraints. *The Visual Computer 11*, 4 (apr 1995), 188--201.

[96] Simplicable. Artificial Stupidity.

[97] Skarupke, M. Why Video Game AI does not Use Machine Learning .

[98] Smith, A. M. *Mechanizing Exploratory Game Design*. PhD thesis, University of California, Santa Cruz, December 2012.

[99] Smith, A. M., and Mateas, M. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Trans. Comput. Intell. AI Games Transactions on Computational Intelligence and AI in Games 3*, 3 (sep 2011), 187--200.

[100] Spinellis, D. The Importance of Being Declarative. *IEEE Software 30*, 1 (January/February 2013), 90--91.

[101] Sproat, R. WordsEye: A Text-to-Scene Conversion System. In *Advances in Natural Language Processing*. Springer Berlin Heidelberg, 2002, pp. 1--1.

[102] Thermal, S. A Brief History of Soccer Video Games.

[103] Thielscher, M. Answer Set Programming for Single-Player Games in General Game Playing. In *Logic Programming*. Springer Berlin Heidelberg, 2009, pp. 327--341.

[104] Torres, A., Galante, R., Pimenta, M. S., and Martins, A. J. B. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology 82* (2017), 1--18.

[105] Trazzi, M., and Yampolskiy, R. V. Artificial Stupidity: Data We Need to Make Machines Our Equals. *Patterns 1*, 2 (may 2020), 100021.

[106] Unity Technologies. Unity, 2023.

[107] Universit`a della Calabria. ASP Standardization, 2013.

[108] Vilone, G., and Longo, L. Explainable Artificial Intelligence: a Systematic Review, 2020.

[109] Weinzierl, A., Taupe, R., and Friedrich, G. Advancing lazy-grounding asp solving techniques -- restarts, phase saving, heuristics, and more, 2020.

[110] West, M. Intelligent Mistakes: How to Incorporate Stupidity Into Your AI Code.

[111] Xia, B., Ye, X., and Abuassba, A. O. Recent Research on AI in Games. In *2020 International Wireless Communications and Mobile Computing (IWCMC)* (jun 2020), IEEE.

[112] xkcd. NP-Complete, 2022.

[113] Xu, K., Stewart, J., and Fiume, E. Constraint-based Automatic Placement for Scene Composition. *Proceedings - Graphics Interface* (06 2002).

[114] Zacarias, F., Cuapa, R., Jimenez, L., and Vazquez, N. Modelling of Intelligent Agents Using A–Prolog. *International Journal of Artificial Intelligence and Applications 10*, 2 (mar 2019), 1--11.

[115] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. A Survey of Large Language Models, 2023.

A. Axioms

In the context of our problems, we will be dealing with Discrete Event Calculus (DEC), since Answer Set Programming can only deal with discrete domains. Following, we present the twelve DEC axioms [76, 79].

A.1 Discrete Event Calculus

Axiom 1.

$$\forall T_1, F, T_2(stoppedIn(T_1, F, T_2) \Leftrightarrow \exists E, T(T_1 < T < T_2 \wedge happens(E, T) \wedge terminates(E, F, T))). \quad (A.1)$$

Axiom 2.

$$\forall T_1, F, T_2(startedIn(T_1, F, T_2) \Leftrightarrow \exists E, T(T_1 < T < T_2 \wedge happens(E, T) \wedge initiates(E, F, T))). \quad (A.2)$$

Axiom 3.

$$\forall E, T_1, T_1, T_2, T_2((happens(E, T_1) \wedge initiates(E, T_1, T_1) \wedge T_2 > 0 \wedge trajectory(T_1, T_1, T_2, T_2) \wedge \neg stoppedIn(T_1, T_1, T_1 + T_2)) \Rightarrow holdsAt(T_2, T_1 + T_2)). \quad (A.3)$$

Axiom 4.

$$\forall E, T_1, T_1, T_2, T_2((happens(E, T_1) \wedge terminates(E, T_1, T_1) \wedge T_2 > 0 \wedge antiTrajectory(T_1, T_1, T_2, T_2) \wedge \neg startedIn(T_1, T_1, T_1 + T_2)) \Rightarrow holdsAt(T_2, T_1 + T_2)). \quad (A.4)$$

Axiom 5.

$$\forall F, T((holdsAt(F, T) \wedge \neg releasedAt(F, T + 1) \wedge \neg\exists E(happens(E, T) \wedge terminates(E, F, T))) \Rightarrow holdsAt(F, T + 1)). \quad (A.5)$$

Axiom 6.

$$\forall F,T((\neg holdsAt(F,T) \wedge \neg releasedAt(F,T+1) \wedge \neg\exists E(happens(E,T) \wedge initiates(E,F,T))) \Rightarrow \neg holdsAt(F,T+1)). \quad \text{(A.6)}$$

Axiom 7.

$$\forall F,T((releasedAt(F,T) \wedge \neg\exists E(happens(E,T) \wedge (initiates(E,F,T) \vee terminates(E,F,T)))) \Rightarrow releasedAt(F,T+1)). \quad \text{(A.7)}$$

Axiom 8.

$$\forall F,T((\neg releasedAt(F,T) \wedge \neg\exists E(happens(E,T) \wedge releases(E,F,T))) \Rightarrow \neg releasedAt(F,T+1)). \quad \text{(A.8)}$$

Axiom 9.

$$\forall E,T,F((happens(E,T) \wedge initiates(E,F,T)) \Rightarrow holdsAt(F,T+1)). \quad \text{(A.9)}$$

Axiom 10.

$$\forall E,T,F((happens(E,T) \wedge terminates(E,F,T)) \Rightarrow \neg holdsAt(F,T+1)). \quad \text{(A.10)}$$

Axiom 11.

$$\forall E,T,F((happens(E,T) \wedge releases(E,F,T)) \Rightarrow releasedAt(F,T+1)). \quad \text{(A.11)}$$

Axiom 12.

$$\forall E,T,F((happens(E,T) \wedge (initiates(E,F,T) \vee terminates(E,F,T))) \Rightarrow \neg releasedAt(F,T+1)). \quad \text{(A.12)}$$

B. Code Listings

Full listings of the code used in this thesis are provided in this git repository.

C. Article

The work in this thesis resulted in a publication [67]. The article is included in this chapter.

Investigating Applicability Heuristics of Answer Set Programming in Game Development: Use Cases and Empirical Study

Evangelos Lamprou, Christos Fidas

C.1 Abstract

The game industry is continuously growing and evolving, with new ways of creating games being developed. However, even with the availability of powerful game engines, developers are still forced to spend time and effort implementing common game features, such as basic AI, path-finding, and simple scene variations. This can become a serious detriment for indie game developers. The present research focuses on the application of Answer Set Programming (ASP) methods within the game development process, aiming to support rapid and cost-effective game prototyping for indie game developers. Specifically, we present a pragmatic approach to the usage of ASP for game development within certain use cases and we report on evaluation results based on feedback that was received from end-users. Analysis of results demonstrates how ASP can be used, providing new ways of thinking about game mechanics and content creation, and eventually paving the way for new game design frameworks and possibilities. On the downside, adoption of the suggested method can be difficult due to unfamiliarity with the ASP programming paradigm.

C.2 Introduction

The game industry is continuously growing and evolving. As with other industries, new ways of creating games are being developed. Game engines offer a plethora of tools and features that can aid game designers in bringing their

ideas to life [1]. In most game development engines, however, game programmers are presented with imperative languages. Traditional imperative programming is a programming paradigm that specifies step-by-step instructions for the computer to execute using variables, loops and conditional statements. Declarative programming focuses on what a program should accomplish, using high-level abstractions to define the problem domain, allowing the computer to reason about the solution.

Answer Set Programming (ASP) [31] is a declarative programming paradigm that has shown promise in solving complex problems in various fields such as employee assignment [87], legal reasoning [5], resolution of software package configuration errors [38] and automatic music composition [9].

Declarative techniques have already been applied in the context of games [2], with examples ranging from commercial grade software such as the game *F.E.A.R* [82] and *Halo 3* [55] to more research-centric approaches such as the *Video Game Playing Description Language (GDL)* [89] and the *Ludocore* engine [99], where game semantics are encoded inside declarative logical frameworks. Now, more closely relating to our work, ASP has also seen application in games. Agents have been developed that can solve puzzles [33, 103, 114] or play games like *Angry Birds* [18]. The work in [4] and [99] explores the application of ASP for procedural content generation, focusing on the creation of puzzle levels while also highlighting how ASP can act as a highly expressive tool for creating game content generators in a time-effective manner.

This paper elaborates on an ASP framework for game development and presents its added value in the context of specific use cases in game programming. Specifically, we present evaluation results that highlight that using declarative tools can significantly speed up development time and relieve game programmers from the burden of understanding and implementing complex algorithms, leading to more flexible and reusable code while also enabling a different creative approach to game development.

Motivation and Contribution. In small game development teams (1 - 5 people) the role of developer and designer are often intertwined. That means that a game designer often has to interrupt the creative process of refining and testing a game idea to implement complex game logic. There is still a lack of tools suitable for rapid prototyping. As a solution, we propose a framework for using ASP in games, comment on aspects of game development suitable to be implemented in ASP and present some case studies. We also conduct an evaluation of ASP with developers at various levels of familiarity with the paradigm. To our knowledge, a user study on the merits of using ASP for game creation has not been previously conducted.

The paper is structured as follows. First we present background knowledge for understanding the ASP programming paradigm. Next, we present the suggested framework that can be utilized by indie game developers and finally, we present the results of the evaluation study.

C.3 Background Theory on Answer Set Programming

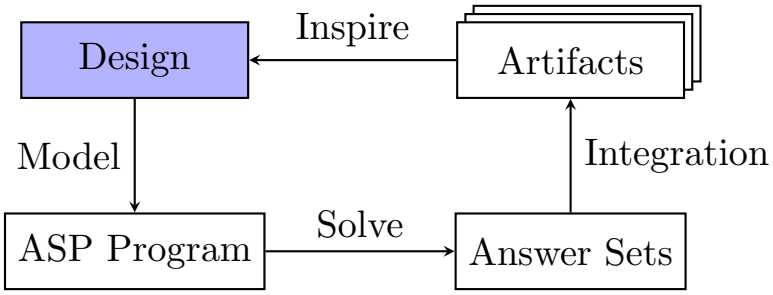


Figure C.1: Game development workflow with the help of ASP tooling. The designer starts with an initial goal, the design of a game mechanic/behavior/set of artifacts which leads to a specification in the form of an ASP program. The program's solutions can help to further refine the initial design as missing or unwanted aspects of it become apparent after its integration with the rest of the game [98].

Answer Set Programming (ASP) [31] is a problem-solving paradigm with roots in logic programming and non-monotonic reasoning. The work of [42] first formalized the semantics of stable models and the ASP core language. As shown in fig. C.1, the programming model of ASP is one where the programmer models the problem domain, with the solution being handled by a solver program. Programming using this paradigm is done in a family of languages sometimes called *AnsProlog* [41]. In our work, we will be using the input language of *Clingo* [37], which is a high-performance integrated solver with a large collection of libraries and bindings helpful in integrating it with external tools.

The syntax is similar to the one of *Prolog*, a popular deductive logic programming language. It employs a unified approach to represent both code-like and data-like knowledge through logical terms. These terms can be atoms, which are named symbols, numbers, strings, or compounds that consist of a functor (a symbol) and a list of logical terms serving as arguments. By utilizing collections of logical terms (listing C.1), any data structure relating to the state of the game world can be easily represented.

Listing C.1: A set of facts describing game elements/game state.

```
object(house).
position(player, vec3(1, 0, 1)).
size(house, vec3(4, 4, 4)).
tile(1, 1, water).
move(player, left).
object(orc).
object(frog).
```

Listing C.2: Logic rules describing relationships between entities and connection between action and effect.

```
damaged(player) :- attacked(player).
damaged(player) :- fall(player).
hostile(X) :- enemy(X).
friend(X) :- object(X), not hostile(X).
pos(player,X+1,Y,T+1) :- pos(player,X,Y,T),
                         move(player,right).
```

For more complex reasoning, the program author can add logical rules which can be expressed using the `:-` operator. The left side of a rule is called its "head" and the right side its "body". Simply put, the head of a rule is true if its body is true. Inside a single rule, commas between atoms signify the "and" logical operation while repeating the same rule head with different bodies signifies the "or" operation. Rules can be used to specify the effects of actions or derive properties of game objects. In rules, an atom starting with a capital letter designates a variable.

The choice/generation capabilities of Answer Set Programming come from the ability of the program author to allow the ASP solver to make choices among a set of atoms. These can be encoded using "choice rules".

Listing C.3: A choice rule.

```
{chosen(X,Y) : person(X)} :-
    house(Y).
```

Listing C.4: An integrity constraint.

```
:- chosen(X, Y), chosen(Z, Y),
     X == Y.
```

Listing C.5: An optimization directive.

```
#minimize{C : cost(E,C)}.
```

The choice rule in listing C.3 is translated as "for each house Y, choose among the set of atoms $chosen(X, Y)$, where X is a person". This can produce several answer sets, one for each possible variable assignment. However, some of the generated answer sets are invalid in the context of the problem. For example, it should not be allowed for two people to have chosen the same house. Such rules can be encoded elegantly in the form of "integrity constraints" (listing C.4) which entail what is not allowed to hold in the answer sets generated. This provides a mechanism for "filtering out" unwanted answer sets. In problems where there might be multiple valid answer sets, we can also add optimization directives, instructing the solver to output optimal answer sets over certain variables. The program author can use the `#minimize` and `#maximize` directives (listing C.5).

To compute answer sets, ASP programs are inputted into ASP solvers. These solvers provide efficient mechanisms to generate the set of valid answers to the given problem. An ASP solver can be thought of as a black box, with them being interchangeable as long as the input language semantics remain the same.

C.4 Heuristics and Method for Applying ASP in the Game Development Process

An important aspect of the suggested framework is the determination of specific game design heuristics pointing towards game components suitable for ASP programming approaches. We suggest the following applicability principles/heuristics:

- ***ASP Applicability Heuristic (A).*** **Brevity**: ASP (and declarative programming in general) can reduce software complexity, [100] allowing for concise code [12]. This however requires that when modelling a game mechanic, only its important aspects are encoded. For example, in a maze game, only essential elements like maze layout, starting point, treasure location, and movement rules would be included.

- ***ASP Applicability Heuristic (B).*** **Relatively Small Solution Space**: Avoid scenarios where solving times become very large. A small solution space involves the generator/agent having a limited amount of options encoded in each of the ASP program's choice rules. For example, a designer should opt to have an agent move in one of four directions (up, down, left, right), rather than the full range of motion. A physics simulation can then be employed through the game engine to introduce natural-looking movement

  Designers can mitigate this limitation by breaking down the problem into smaller sub-problems. In [19], the authors decoupled the generation of a dungeon's topology from the content of each of the rooms, reducing generation times, while in [86], an agent's different states (eat, hiding, action) are split into smaller ASP program modules, using meta-reasoning to decide which relevant parts of the knowledge base will be used for solving.

- ***ASP Applicability Heuristic (C).*** **Emergent Complexity**: Create scenarios where interesting behavior emerges when agents are observed interacting with each other and the environment inside the game world or when the generated artifacts exhibit interesting patterns that were not explicitly modeled in the ASP program. In [82], where a declarative planning layer was added to agents in the game *F.E.A.R*, complex behaviors emerged from a combination of simple goals and actions together with the dynamic state of the game world. The *Portal* game levels generated using ASP in [4] were complex and challenging, while modeling only involved specifying how a level is solvable and ensuring its topological integrity.

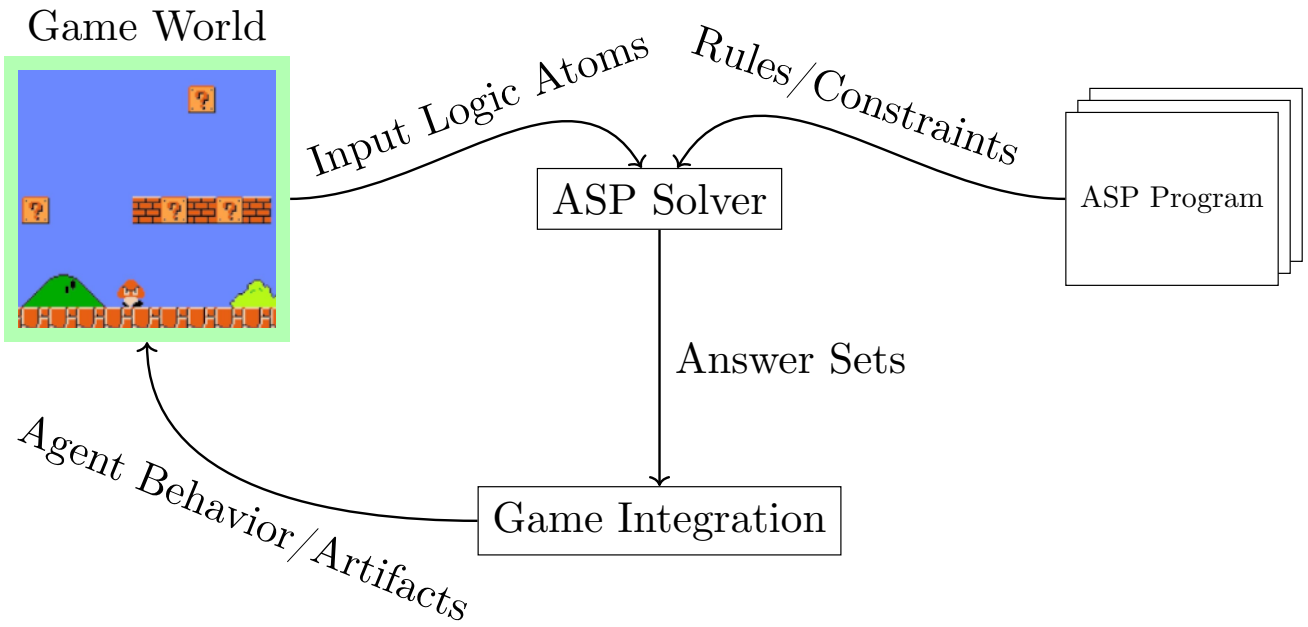


Figure C.2: High level overview of ASP integration into a game engine. The *Game World* consists of the current game state and the information of all the game objects inside of it. Input logic atoms are the facts used to describe the current state of the game world. These, together with the rules and constraints of the ASP program, are fed into the ASP solver, which then outputs answer sets. These describe logic such as the actions that an agent should take or where an object should be placed. Through the *Game Integration* component, these answer sets are used to update the game world.

Furthermore, we propose a standardized development methodology (fig. C.2) to guide aspiring developers to successfully apply ASP to their applications by providing some general programming guidelines relating to ASP modelling, based on the "guess and check" paradigm [34].

1. **Step (a): Determine Input and Output Atoms**: The set of input atoms provides the context required for the ASP program to give correct results. These are usually dynamic aspects of the game's runtime and change at each invocation of the ASP solver. Examples of these are the starting location of an agent or the list of objects that need to be placed. On the other hand, output atoms encode the results produced by the solver and which will be interpreted by the game runtime as artifacts or agent behavior. These include things like the direction at which an agent will move in the next timestep or the location an object should be placed at.

2. **Step (b): Generate "Random" Answer Sets**: During the development phase, the programmer can efficiently construct an ASP program

comprising of choice rules to generate partially random outcomes, considering the output atoms. Although the resulting artifacts or behavior may be flawed or unsuitable, this approach facilitates the identification and troubleshooting of potential technical problems. This stage also involves the creation of a visualizer or the integration of the solver with the game runtime to enhance the debugging process. A software architecture for embedding ASP into a game engine is proposed in [3].

3. **Step (c): Add Integrity Constraints/Optimization Directives**: Based on the current problem's domain, it is necessary to add integrity constraints and/or optimization directives. Constraints provide direct control over the produced answer sets for them to comply both with the game's ruleset as well as the designer's ideas. Among them, if needed, the solver can output the most optimal ones based on some variable using optimization rules.

C.5 Proof of Concept and Case Studies

To illustrate the aforementioned applicability heuristics of ASP in the context of game development, we devised a number of case studies[1].

C.5.1 Case study (a): Tile-Level Terrain Generation

With terrain generation, games often follow an approach where a noise function like Perlin noise [84] is used to determine the type of terrain to be placed at some specific (x, y) location. This approach has seen major adoption in the game industry as it is used in major titles like *Minecraft* [77]. However, this approach although performant, does not allow for high levels of controllability. For example, a designer cannot specify that they want a certain number of mountains to be placed, or a river that flows through a specific area. Using ASP, we can generate natural-looking terrain using a highly expressive language. The ASP applicability heuristics of **brevity** and **emergent complexity** are present in this example. In our ASP program, the placement of tiles inside the grid can be encoded in a single-choice rule. Then, through integrity constraints, we add rules that propagate through the entire grid, creating interesting patterns.

The application of our proposed methodology works as follows:

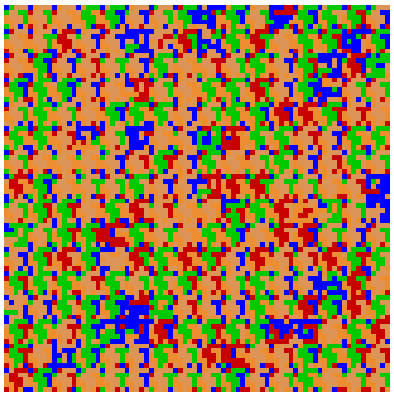
1. Determine output atoms, which are the type of tile in each of the grid's locations (an atom of the form $tile(x, y, type)$). Input atoms consist of

[1]Source code from our examples as well as the projects developed in the user study can be found in https://anonymous.4open.science/r/asp-games-anon/README.md.

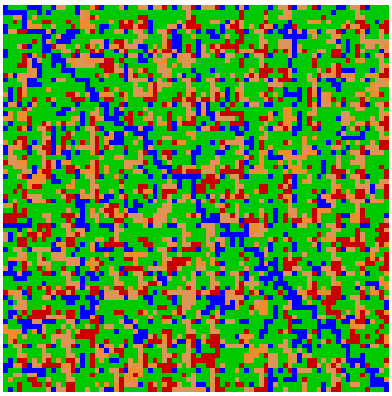
predicates that will control certain aspects of the generation. For example, the programmer could input a single fact of the form $tile(1, 1, water)$ which will force the generator to place the specific type of tile on that location.

2. Add a single choice rule that places a tile of random type at every grid space. At this point, we also develop a script that takes the output of the solver and translates the output atoms into colored pixels.

3. The generator's output is controlled using integrity constraints. In our example, we added constraints where no "water" and "lava" tiles can touch while there must also be a river flowing along the diagonal.



(a) A generated map with no constraints encoded.



(b) A generated map with a river flowing through it and no water and lava tiles touching (inside the same sub-grid).

Figure C.3: Examples of maps generated using our terrain generator.

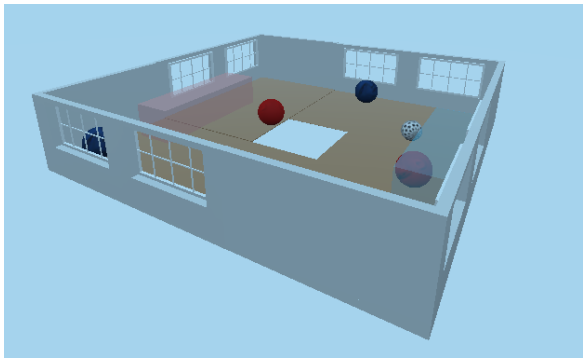
C.5.2 Case study (b): Football (Soccer) Game

We developed a game scenario that showcases how ASP can model two teams of adversarial agents and how the result is both interesting and appropriate. Our method avoids the need to implement path-finding algorithms like A* [94] or the application of reinforcement learning techniques [90] which can be hard to develop and debug during game prototyping. The characteristics of **relatively small solution space** and **emergent complexity** are present in this case study. The solution space can be controlled by reducing the number of time-steps the agent will "see" in the future. Emergent complex behavior comes from the fact that we will not explicitly ask the agent to kick the ball to score a goal. Rather, we explain through logical rules how the ball's location changes when kicked and allow the ASP solver to derive a winning strategy based on that information.

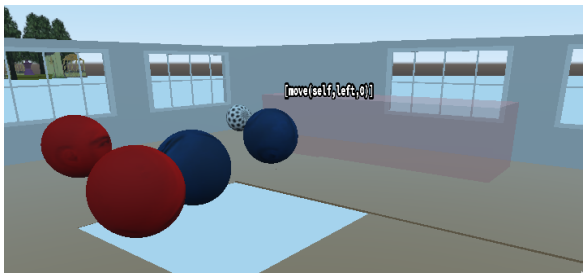
1. In this case study, the input atoms are the location of critical game world objects like the location of the ball, the two goals, and the other players. Output atoms will be the agent's decisions of where to move, whether they will kick the ball and towards which direction.

2. We add choice rules for the agent's possible actions. At this point, we integrate the solver inside the *Godot* [47] game engine, translating the output atoms into actions inside the game world.

3. We finally add integrity constraints, making the agents avoid colliding with each other as well as an optimization directive that tries to minimize the ball's distance to the enemy's goal. The sequence of actions an agent will take will be the ones that bring the ball closer to the goal.



(a) A top-down screenshot of the football playing field. It consists of two goals and four agents, two on each team. The ball colliding with one of the semi-transparent rectangles will result in a point being scored for the team opposite the goal.



(b) The football players. The floating text above the agent indicates the agent's latest "thought", which is the action it will perform next. In this image the predicate $move(self, left, 0)$ is part of the optimal answer set produced by the ASP solver, meaning that in the next timestep, the agent will move to the left.

Figure C.4: The football game implementation.

C.6 Empirical Study

C.6.1 Research Questions

The main research questions of the empirical study were to investigate: **RQ1**) Whether the aforementioned applicability heuristics can be confirmed by third-party game designers within the context of their own game designs; **RQ2**) whether the suggested ASP methodological approach supports creativity in game design and **RQ3**) whether end-users found difficulties in applying the suggested workflow.

C.6.2 Participants

We recruited a total of 8 participants (2 female and 6 male), all of whom were undergraduate *Electrical and Computer Engineering* students. All of them had experience programming with imperative languages, with 3 having experience with logical languages (either *Prolog* or *Clingo*). All but two of the participants had prior experience with game development, in the context of personal projects. Participants were informed that no personal data was collected aside from their answers to the interview part of the study. On average, each of the participants took part in the study procedure for a duration of 0.5 to 3 hours, resulting in a total study length of approximately twenty-four hours.

C.6.3 Study Procedure

The study conducted was a one-on-one user study, where each participant worked individually with the researcher. The study utilized all collected data in an anonymous manner, and participants had the freedom to withdraw from the study at any time of their choosing.

- **Phase A - Introduction to ASP.** The study began with a brief overview of Answer Set Programming (ASP) technology and the *Clingo* language's syntax and semantics.
- **Phase B - Implementation of Game Mechanic/Generation of Content.** Participants were then asked to think of a game mechanic or content generator that they would like to implement using ASP. We encouraged participants to be creative and come up with unique or challenging ideas. After participants had an idea in mind, the researcher assisted them in creating the logic program for their game mechanic using ASP. We avoided instructing the participants during the modeling process, where they would come up with the logical rules for their program and limited our interference to helping with issues concerning *Clingo*'s syntax. During the creation process, the researcher was available to answer questions and provide guidance as needed. After each participant completed their game mechanic, the researcher evaluated it with them to discuss the strengths and weaknesses of the approach taken and provided feedback on how it could be improved.
- **Phase C - Discussion.** Finally, we conducted a semi-structured interview to receive qualitative feedback and elicit the participant's likeability and comments concerning the proposed workflow.

The study provided valuable insights into how well participants were able

to understand and apply ASP to create game mechanics, as well as identify the strengths and limitations of this approach.

C.7 Analysis of Results

C.7.1 RQ1: Applicability of suggested ASP Heuristics in Game Development Scenarios

Given the number of participants, our assessment is based rather on a qualitative than a quantitative research approach. The participants gave concrete results on the strengths and weaknesses of the proposed workflow. Moreover, during the study, a range of applications were created. Most gravitated towards creating generation programs rather than agent behavior mechanics.

Application	Description	Design Characteristic	Duration	Iterations
Wind Direction Simulator	Simulate wind direction in a grid.	**B, E**	2.5h	3
Loot Generator	Generates reward combinations.	**B, S**	1h	2
Conversational Agent	Simulate a conversation.	**S, E**	2h	4
Space Traversing Agent	Agent that can navigate a 2D space.	**B**	1h	2
Level Generator	Generates game levels	**B, E**	2h	5
Ball Sort Solver	Solves/Generates instances of puzzle game.	**B, S**	1h	2
Futoshiki Solver	Solves/Generates instances of puzzle game	**B, S**	0.5h	2
Level Generator	Generate levels with controllable difficulty.	**B, E**	3h	6

Table C.1: The applications created by the participants during the study and the design characteristics which they satisfy for ASP suitability. These are brevity (**B**), relatively small solution space (**S**), and emergent complexity (**E**). Duration measures the total development time. A single iteration is defined as the process of inspecting the program's produced answer sets and making one or more significant modifications.

Participants demonstrated competence in applying the ASP applicability heuristics throughout the study. Their creations showcased an understanding of at least one of brevity, consideration of relatively small solution spaces, and the ability to capture emergent complexity. These observations affirm the practicality and effectiveness of the ASP applicability heuristics in guiding participants towards successful implementation and utilization of ASP in the game development process.

C.7.2 RQ2: Does the suggested ASP methodological approach supports creativity in game design?

We asked participants whether the proposed workflow provide design inspiration for developing games. A participant with extensive game development experience stated that the workflow provides the opportunity to conceive entirely novel game mechanics that would otherwise be

overlooked due to the challenges associated with traditional programming. This participant emphasized that Answer Set Programming (ASP) allows for the creation of game mechanics that might otherwise be deemed too complex to develop.

The majority of participants identified the primary value of ASP as its ability to address problems succinctly. They expressed that ASP enables concise problem-solving, streamlining the development process. Moreover, one participant specifically noted that ASP offers advantages for non-programmers, as it facilitates the expression of logical constraints by individuals without extensive programming experience.

Most participants after being introduced to the ASP paradigm were able to intuitively recognize scenarios where it could be applicable. Of the design heuristics we proposed, the one of *relatively small solution space* was the one that was hardest to apply and what participants felt as most limiting to their design.

While some participants mentioned a steep learning curve, overall, the ASP approach was seen as a powerful and creative method for game design. A designer/programmer pair (or a single designer also capable of programming in the paradigm) could quickly go from idea to working prototype, with time between iterations being minimized.

-- *Participant 1*: “It gives you the ability to create entirely new game mechanics that you wouldn't bother developing otherwise because of the programming difficulty.”

-- *Participant 5*: “[A game developer] might say something like "Oh, this can be easily encoded using rules". Now, it's easier to think of a game mechanic and come up with constraints to create it.”

During the study, participants iteratively added constraints. This resulted in a refinement process where more complex rules and constraints were added one at a time. The produced artifacts were examined to validate the correctness of their modeling and participants often came up with new rules and constraints after noticing some undesirable patterns or behavior in the output. This empirically confirms the validity of the conceptual loop in fig. C.1, where the results from the designer's modeling can fuel their creative process.

We asked participants whether they would use ASP again to create a game mechanic or content generator. The majority of participants expressed a positive response when asked about their potential usage of Answer Set Programming (ASP). Specifically, they emphasized its suitability for decision-making applications. Participants widely agreed that ASP could prove highly effective for programming game logic. However, they acknowledged that mastering ASP requires substantial practice to utilize comfortably.

-- *Participant 1*: "I believe it would take me a long time to learn the language. The syntax is strange, but I can create a mental model of how it works."

-- *Participant 5*: "I would use it again in simple scenarios."

-- *Participant 7*: "I would use Clingo again if what I wanted to build involved logic and decision making."

C.7.3 RQ3: Difficulties in the suggested ASP workflow application

We asked participants what they found challenging about applying the proposed workflow. One common challenge encountered by participants was the unfamiliar syntax used in *Clingo*. They found it to be unconventional, leading to difficulties in writing and reading programs. Moreover, the absence of a debugger posed a significant obstacle in identifying errors, particularly during later iterations when programs became more complex with multiple rules and integrity constraints. Slow solving times and limited scalability were also identified as limitations.

We reached a firm conclusion regarding the significance of investing time in the development of a comprehensive mapping between the Answer Sets generated by the ASP solver and their in game representation. It became evident that relying solely on inspecting the raw output of the default solver is prone to errors and can lead to a mentally demanding experience for developers.

-- *Participant 7*: "It would be nice to have a graphical interface that shows how the solver arrives at solutions."

-- *Participant 8*: "[The workflow] could be improved if Clingo had better syntax. Maybe an abstraction layer built on top of it."

C.8 Conclusions and Future Work

The aim of our research was to explore the applicability heuristics of Answer Set Programming (ASP) in the context of game development. We focused on identifying potential use cases where ASP could be effectively utilized and conducted an empirical study to evaluate its practical effectiveness. To achieve our research objectives, we first identified potential use cases where ASP could be applied. These could include tasks such as character behavior modeling, game level design, game rule specification, and puzzle generation, among others. By analyzing these potential use cases, we aimed to determine

the strengths and limitations of ASP and its suitability for each specific application. Furthermore, we conducted an empirical study to assess the practical feasibility and effectiveness of using ASP in game development. This study involved designing experiments or scenarios where ASP-based solutions were implemented and evaluated. We collected data, analyzed the results, and derived insights regarding the performance, efficiency, and overall applicability of ASP in the given game development contexts. The proposed workflow stems from the need for robust high level programming interfaces to assist with the development of complex applications like games. We have proposed a methodology for recognizing parts where game logic can be elegantly expressed using answer-set programming.

A research project that could overcome *ASP*'s narrow adaptation is the creation of language that preservers the *AnsProlog* language's semantics while providing a more developer-friendly syntax and structure. Future work should involve the application of ASP in larger-scale game projects, exploring how the proposed workflow can fit into long-running game projects.

Limitations. In this, we refer to the limitations of our research. Certainly, a limitation of our own study is that the participants' profile was limited to students rather than experienced game developers. Additionally, the number of participants was relatively small, resulting in our reliance on qualitative analysis for the research findings.

List of Figures

List of Tables

Πανεπιστήμιο Πατρών, Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών
Ευάγγελος Λάμπρου του Νικολάου