Proof Pearl: Formalizing Spreads and Packings of the Smallest Projective Space PG(3,2) Using the Coq Proof Assistant

Nicolas Magaud
Lab. ICube UMR 7357 CNRS Université de Strasbourg, France

Abstract

We formally implement the smallest three-dimensional projective space PG(3,2) in the Coq proof assistant. This projective space features 15 points and 35 lines, related by an incidence relation. We define points and lines as two plain datatypes (one with 15 constructors for points, and one with 35 constructors for lines) and the incidence relation as a boolean function, instead of using the well-known coordinate-based approach relying on $\mathrm{GF}(2)^4$. We prove that this implementation actually verifies all the usual properties of three-dimensional projective spaces. We then use an oracle to compute some characteristic subsets of objects of PG(3,2), namely spreads and packings. We formally verify that these computed objects exactly correspond to the spreads and packings of PG(3,2). For spreads, this means identifying 56 specific sets of 5 lines among 360 360 ($= 15 \times 14 \times 13 \times 12 \times 11$) possible ones. We then classify them, showing that the 56 spreads of PG(3,2) are all isomorphic whereas the 240 packings of PG(3,2) can be classified into two distinct classes of 120 elements. Proving these results requires partially automating the generation of some large specification files as well as some even larger proof scripts. Overall, this work can be viewed as an example of a large-scale combination of interactive and automated specifications and proofs. It is also a first step towards formalizing projective spaces of higher dimension, e.g. PG(4,2), or larger order, e.g. PG(3,3).

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification

Keywords and phrases Coq, projective geometry, finite models, spreads, packings, PG(3, 2)

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.25

Supplementary Material *Software (Source Code)*: `https://github.com/magaud/PG3q`
archived at `swh:1:dir:f4ea06977d8a030690dddc924ccaa0b7590831ff`

Acknowledgements We would like to thank Pascal Schreck for introducing us to the domain of (finite) projective geometry and the anonymous reviewers for their careful reading and suggestions.

1 Introduction

Projective incidence geometry [9, 6] is one of the simplest description of geometry, where only points and lines as well as their incidence properties are considered. In addition, in such a setting, we assume that two coplanar lines always meet. There exist some finite and infinite models of projective incidence geometry. Finite projective spaces are usually built from finite (Galois) fields of cardinality n denoted $GF(n)$ via a homogeneous coordinate system. Finite projective spaces arising from $GF(n)$ are denoted by $PG(d, n)$ where d is the dimension of the space and n the order of the underlying field. Several finite models are related to interesting mathematical puzzles and sometimes have practical and enjoyable applications. This is the case for the finite projective plane PG(2,7), which was used to design the card game Dobble[1]. In this game, players must identify a symbol which appears on both their card and their opponent's card. As the card desk (almost exactly) implements

[1] `https://en.wikipedia.org/wiki/Dobble`

Table 1 Numbers of points, lines and points per line depending on the dimension and the order of projective planes and spaces.

	# points	# lines	# points per line
$PG(2,2)$	7	7	3
$PG(2,3)$	13	13	4
$PG(2,5)$	31	31	6
$PG(2,n)$	n^2+n+1	n^2+n+1	$n+1$
$PG(3,2)$	15	35	3
$PG(3,3)$	40	130	4
$PG(3,4)$	85	357	5
$PG(3,n)$	$(n^2+1)(n+1)$	$(n^2+n+1)(n^2+1)$	$n+1$

the projective plane PG(2,7), given two cards (=lines), there always exists a symbol (=point) which belongs to both cards. In a three-dimensional setting, the smallest projective space PG(3,2) can be used to find some solutions to an old combinatorial problem: *Kirkman's schoolgirl problem* [7], which is stated as follows: *Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily, so that no two shall walk twice abreast.* As noted by Hirschfeld in [14, page 75], some solutions to this problem correspond to some packings of PG(3,2), which are one of the substructures of PG(3,2) that we study in this article.

Finite projective spaces have been studied extensively from a mathematical point of view (see e.g [14]). Recently [4], we started studying small finite projective planes/spaces from a computer science perspective. We formalized usual projective planes such as PG(2,2), PG(2,3) or PG(2,5) as well as the smallest projective space PG(3,2) using the Coq proof assistant [8, 2]. We especially focused on proving that the synthetic axioms for projective geometry hold in these models. In this paper, we follow up on experiments carried out recently [16] and we formally describe some of the characteristic subsets of PG(3,2), namely spreads of lines and packings of spreads as well as their properties.

In a three-dimensional setting, the number of points and lines increase rapidly with the order, as shown in Table 1. Thus we need to design extremely efficient proof techniques for PG(3,2) if we want our approach to be scalable to projective spaces of higher dimension or larger order. The whole Coq formalization is available online and can be retrieved at: `https://github.com/magaud/PG3q`[2]. Pointers to specific parts of the development are given throughout the document. Visual representations of the smallest projective space PG(3,2) can be retrieved from `https://demonstrations.wolfram.com/15PointProjectiveSpace/`. For illustration purposes, we reproduce a figure taken from wikipedia[3], which presents PG(3,2) as a tetrahedron (see Fig. 1).

This paper is organized as follows. In Sect. 2, we show how to formally describe PG(3,2) in Coq using plain inductive types. In Sect. 3, we define the notions of collineations, spreads and packings in the setting of PG(3,2). In Sect. 4, we characterize all the spreads of PG(3,2) and show that they are all isomorphic. In Sect. 5, we characterize all the packings of PG(3,2) and then classify them into two distinct classes. In Sect. 6, we present some proof engineering techniques and suggest some additional optimizations to make the proof development smaller and easier to compile. Finally, in Sect. 7, we draw some conclusions and outline how this work can be extended to projective spaces of higher dimension or larger order.

[2] Be aware that compiling all the .v files of this development requires about 13 hours on a standard PC.
[3] `https://en.wikipedia.org/wiki/PG(3,2)`

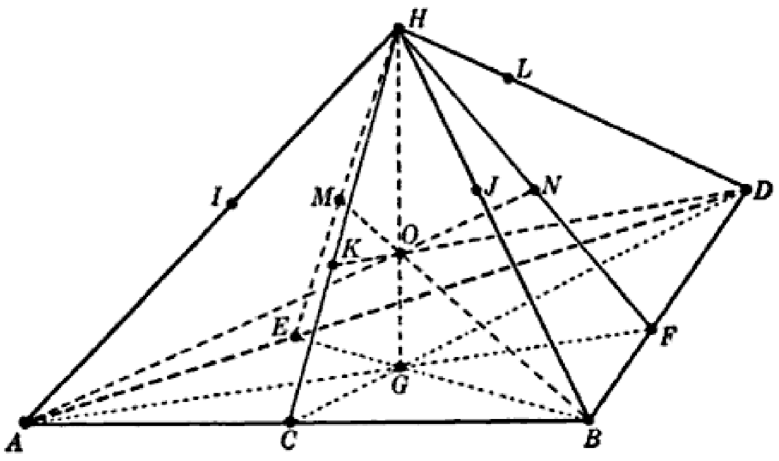

Figure 1 The smallest projective space PG(3,2), represented as a tetrahedron.

2 Formal Description of the Projective Space PG(3,2) in Coq

We first present an abstract interface (a Coq module) to describe what a projective space is. We also propose an implementation of PG(3,2), relying on plain inductive datatypes for points and lines. We then show that all axioms of the projective space are verified by this implementation.

2.1 Specification of Projective Spaces

A three-dimensional projective space is parameterized by two types `Point` and `Line` as well as an incidence relation `incid_lp` (see Table 2 for the actual specification in Coq). The two types are equipped with an equality. Axiom `a1_exists` expresses that given two distinct points, one can always define a line going through these points. Axiom `uniqueness` states that given 2 points A and B and 2 lines l and m, if A and B are both incident to both l and m, then either $A = B$ or $l = m$. Axiom `a2`, also known as Pasch axiom, states that two coplanar lines always intersect. Axiom `a3_1` expresses that given a line, there are always three distinct points on it. Axiom `a3_2` expresses that there exist two lines which are not coplanar, thus making the dimension $n > 2$. Finally axiom `a3_3` states that, given 3 lines l_1, l_2 and l_3, there always exists a fourth line m which intersects these 3 lines. This last axiom bounds the dimension so that $n \leq 3$.

2.2 Points, Lines and the Incidence Relation

We choose to use two simple inductive types to represent points and lines of PG(3,2). Points are represented by an inductive datatype of 15 constructors without arguments. Lines are represented in the same way using 35 constructors.

```
Inductive Point :=
| P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9
| P10 | P11 | P12 | P13 | P14.
```

```
Inductive Line :=
| L0 | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 | L10 | L11 | L12
| L13 | L14 | L15 | L16 | L17 | L18 | L19 | L20 | L21 | L22 | L23
| L24 | L25 | L26 | L27 | L28 | L29 | L30 | L31 | L32 | L33 | L34.
```

As there are three points per line, the incidence relation `incid_lp`[4] can be represented in a compact way using the `match ... with` construct of Coq specification language.

```
Definition incid_lp (p:Point) (l:Line) : bool :=
match l with
| L0 => match p with P0 | P1 | P2 => true | _ => false end
| L1 => match p with P0 | P3 | P4 => true | _ => false end
| L2 => match p with P0 | P5 | P6 => true | _ => false end
| L3 => match p with P0 | P7 | P8 => true | _ => false end
| L4 => match p with P0 | P10 | P9 => true | _ => false end
| [...]
end.
```

In order to avoid writing too many specifications and proof scripts manually in Coq, we choose to build an external specification and proofs generator (a simple C program) which takes as input the number of points, the number of lines as well as the incidence relation as a plain file (`pg32.txt`[5]) which contains for each line of the projective space, the list of the points which are incident to it. Given these three elements, the system automatically builds the inductive datatypes for points and lines as well as the incidence relation. It also defines an artificial order on points and lines based on the index of the corresponding points and lines, i.e. `P0` $<$ `P1` $<$ `P2` $< \ldots <$ `P14`. The specification generator also builds some auxiliary functions, which will be useful to prove existential statements of the form $\forall l1\ l2 :$ `Line`, $\exists\ P :$ `Point`, $\ldots$.

Using plain inductive datatypes may seem naive. An alternative approach to specify points and lines of PG(3,2) could be to use finite types $'I_n$ of ssreflect and the mathematical components library [12, 17]. However the main drawback is that ssreflect is designed for formal reasoning rather than computing. Thus checking the incidence between a point and a line is a highly expensive operation, which prevents us from carrying out proofs efficently. Using plain inductive types is much more efficient both to check incidence properties and to perform case analysis. The only drawback is that inductive datatypes and functions are huge to write, but this is not that important as we manage to generate these specifications automatically. Overall, our choice is to use the main features of ssreflect, especially the small-scale reflection pattern, but with our own datatypes.

2.3 Formal Proofs

Once the projective space PG(3,2) is described, we check whether all the axioms for projective space geometry hold for this model. This requires proving all axioms of the module defined in `https://github.com/magaud/PG3q/blob/master/generic/pg3x_spec.v` and presented in Figure 2. This is pretty straightforward and we try and make these proofs as generic and efficient as possible. We especially focus on writing general-purpose Ltac tactics, which can be easily reused for other models of projective space such as PG(3,3).

We also rely on our specification generator to enhance producing witnesses for existential quantification. We use a form of skolemisation to write functions which compute the existential variable from the other arguments. For instance, to achieve the proof of lemma `a3_3`, we automatically build a (large) Coq function `f_a3_3` which, given three lines l_1, l_2 and l_3 computes a line l_4 as well as its three intersection points with lines l_1, l_2 and l_3.

```
f_a3_3 : Line -> Line -> Line -> Line * (Point * Point * Point)
```

[4] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_inductive.v`
[5] `https://github.com/magaud/PG3q/blob/master/pg32/pg32.txt`

Table 2 Projective spaces of dimension 3: definitions and properties (pg3x_spec.v).

```
Parameter Point, Line : Type.

Parameter eqP : Point -> Point -> bool.
Parameter eqL : Line -> Line -> bool.

Parameter incid_lp : Point -> Line -> bool.

Definition Intersect_In (l1 l2 :Line) (P:Point) :=
  incid_lp P l1 && incid_lp P l2.

Definition dist_3p  (A B C :Point) : bool :=
  (negb (eqP A B)) && (negb (eqP A C)) && (negb (eqP B C)).

Definition dist_4p  (A B C D:Point) : bool :=
      (negb (eqP A B)) && (negb (eqP A C)) && (negb (eqP A D))
      && (negb (eqP B C)) && (negb (eqP B D)) && (negb (eqP C D)).

Definition dist_3l (A B C :Line) : bool :=
     (negb (eqL A B)) && (negb (eqL A C)) && (negb (eqL B C)).

Axiom a1_exists : forall A B : Point,
     {l : Line| incid_lp A l && incid_lp B l}.

Axiom uniqueness : forall (A B :Point)(l1 l2:Line),
     incid_lp A l1 -> incid_lp B l1  ->
     incid_lp A l2 -> incid_lp B l2 -> A = B \/ l1 = l2.

Axiom a3_1 : forall l:Line,
    {A:Point & {B:Point & {C:Point | (dist_3p A B C) &&
      (incid_lp A l && incid_lp B l && incid_lp C l)}}}.

Axiom a2 : forall A B C D:Point, forall lAB lCD lAC lBD :Line,
      dist_4p A B C D ->
      incid_lp A lAB && incid_lp B lAB ->
      incid_lp C lCD && incid_lp D lCD ->
      incid_lp A lAC && incid_lp C lAC ->
      incid_lp B lBD && incid_lp D lBD ->
      (exists I:Point, incid_lp I lAB && incid_lp I lCD) ->
      exists J:Point, incid_lp J lAC && incid_lp J lBD.

Axiom a3_2 : exists l1:Line, exists l2:Line,
       forall p:Point, ~(incid_lp p l1 && incid_lp p l2).

Axiom a3_3 : forall l1 l2 l3:Line,
  dist_3l l1 l2 l3 ->
  exists l4 :Line, exists J1:Point,exists J2:Point,exists J3:Point,
  Intersect_In l1 l4 J1 &&
  Intersect_In l2 l4 J2 &&
  Intersect_In l3 l4 J3.
```

As a consequence, proving the statement a3_3 boils down to feeding Coq with the correct existential variables for the line and the three intersection points, obtained by applying the function f_a3_3 instead of trying all possible lines and points (there are 35 possible lines and 15 possible points) for each of the $35^3 = 42\,875$ possible cases for parameters l_1, l_2 and l_3 of lemma a3_3.

Once that we checked that our implementation of PG(3,2) verifies all the axioms of projective space, we shall study some specific subsets of points and lines, namley spreads and packings.

3 Collineations, Spreads and Packings of PG(3,2)

Spreads are sets of lines of a projective space which can be defined when the number of points per line divides the number of points. This is the case for all PG(n,q) whose dimension n is odd. PG(3,2) features 15 points and has 3 points per line. Thus spreads exist in PG(3,2).

3.1 Collineations

A collineation is a couple of two functions f_p : Point $\rightarrow$ Point and f_l : Line $\rightarrow$ Line where both f_p and f_l are bijections and respect the incidence relation.

```
Definition inj {A:Set} {B:Set} (f:A->B) : Prop :=
  forall x y:A, f x = f y -> x = y.
Definition surj {A:Set} {B:Set} (f:A->B) : Prop :=
  forall y:B, exists x:A, y=f(x).

Definition bij {A:Set} {B:Set} (f:A->B) : Prop := (inj f) /\ (surj f).

Definition is_collineation fp fl :=
  ((bij fp) /\ ((bij fl) /\
  (forall x l, incid_lp x l -> incid_lp (fp x) (fl l)))).
```

Collineations, which are automorphisms of PG(3,2) which respect the incidence relation, shall be useful to establish that two given sets of lines are isomorphic, thus allowing us to classify spreads and packings into equivalence classes.

3.2 Spreads

3.2.1 Definition and Properties

A spread of PG(3,q) is a set of $q^2 + 1$ lines which are pairwise disjoint and thus partition the set of points. In PG(3,2), it corresponds to some sets of 5 lines. As recalled in [3, 7, 15], it is well known that there is only one spread (up to isomorphism) in PG(3,2).

3.2.2 Generating all Spreads of PG(3,2)

Using our external specification and proofs generating program, we automatically compute all sets of lines of PG(3,2) which are disjoint and cover all the points. As lines contain exactly 3 points, they need to be sets of exactly 5 lines so that all the points of PG(3,2) are accounted for. We generate 56 distinct spreads (modulo permutations of the order of the lines involved). These spreads are defined in Coq as a list of 56 sets of 5 lines, as follows:

```
Definition S0 := [ L0; L19; L24; L28; L33 ].
Definition S1 := [ L0; L19; L26; L29; L32 ].
[...]
Definition spreads := [ S0 ; S1 ; S2 ; ... ; S54; S55 ].
```

We also generate automatically the collineations[6] which allows to go from the spread S_i to the spread $S_{((i+1) \bmod 56)}$ of the list `spreads`[7] as shown in the following example for the spreads `S0` and `S1`.

```
Definition fp0_1 (p:Point) := match p with P0 => P0 | P1 => P1 | P2 =>
P2 | P3 => P3 | P4 => P4 | P5 => P5 | P6 => P6 | P7 => P11 | P8 => P12
| P9 => P13 | P10 => P14 | P11 => P7 | P12 => P8 | P13 => P9 | P14 =>
P10 end.

Definition fl0_1 (l:Line) := match l with L0 => L0 | L1 => L1 | L2 =>
L2 | L3 => L5 | L4 => L6 | L5 => L3 | L6 => L4 | L7 => L7 | L8 => L9 |
L9 => L8 | L10 => L11 | L11 => L10 | L12 => L12 | L13 => L14 | L14 =>
L13 | L15 => L15 | L16 => L18 | L17 => L17 | L18 => L16 | L19 => L19 |
L20 => L20 | L21 => L21 | L22 => L22 | L23 => L25 | L24 => L26 | L25
=> L23 | L26 => L24 | L27 => L30 | L28 => L29 | L29 => L28 | L30 =>
L27 | L31 => L34 | L32 => L33 | L33 => L32 | L34 => L31 end.
```

3.3 Packings

Once that we have built spreads as (disjoint) sets of lines covering all the points, we can define packings as sets of spreads covering all the lines of PG(3,2).

3.4 Definition and Properties

A packing of PG(3,q) is a set of $q^2 + q + 1$ spreads which are pairwise disjoint and thus partition the set of lines. In PG(3,2), it corresponds to some sets of 7 spreads. There are 240 packings, each of them being a list of 7 spreads. As recalled in [3, 7, 15], there are (up to isomorphism) exactly two distinct classes of packings in PG(3,2).

3.5 Generating all Packings of PG(3,2)

We generate all sets of spreads which are disjoint and cover all the lines, and which thus are packings. As before, these sets of spreads must have 7 elements, as the number of spreads multiplied by the number of lines in each spread must be equal to the number of lines (35) of PG(3,2). As expected (see Theorem 17.5.6 in [14]), we find 240 labelled packings.

```
Definition PA0 := [ S0; S9; S19; S24; S36; S46; S53 ].
Definition PA1 := [ S0; S9; S19; S28; S38; S40; S53 ].
Definition PA2 := [ S0; S9; S20; S27; S36; S46; S49 ].
[...]
Definition packings := [ PA0 ; PA1 ; PA2 ; ... ; PA238 ; PA239 ].
```

These packings belong to two distinct classes `class0` and `class1`[8].

[6] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_collineations.v`
[7] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_packings.v`
[8] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_packings.v`

```
Definition class0 := [PA0; PA3; PA5; ... PA237; PA239 ].
Definition class1 :=  complement class0 packings.
```

As for spreads, we automatically generate the collineations[9] which allow to go from one packing of the list `class0` (resp. `class1`) to the next packing of `class0` (resp. `class1`).

Now that all externally-computed spreads and packings are defined in Coq, we shall formally verify that they actually are spreads and packings of PG(3,2). We shall also check that all the spreads are isomorphic and that the 240 packings can be classified into two distinct classes of 120 elements.

4 Properties of the Spreads of PG(3,2)

4.1 Characterizing all Spreads of PG(3,2)

Spreads can be specified using the following definitions: the boolean function `is_partition` computes whether the lists of points p, q, r, s and t partition the set of points and `is_spread5`[10] used in conjunction with the function `all_points_of_line` computes whether the lines $l1, l2, l3, l4$ and $l5$ actually constitutes a spread. The boolean function `forall_Point` is a finite universal quantification: this means that `forall_Point (fun t => X t)` stands for `X P0` && `X P1` && `X P2 ...` && `X P14`.

```
  Definition is_partition (p q r s t: list Point) :bool :=
  (forall_Point
  (fun x => inb x p || inb x q || inb x r || inb x s || inb x t))
  &&
  (forall_Point
  (fun x => negb (inb x p && inb x q && inb x r &&
                             inb x s && inb x t))).

Definition is_spread5 (l1 l2 l3 l4 l5:Line) : bool :=
  disj_5l l1 l2 l3 l4 l5  &&
  is_partition (all_points_of_line l1) (all_points_of_line l2)
               (all_points_of_line l3) (all_points_of_line l4)
               (all_points_of_line l5).
```

Once these definitions are set, we prove that the spreads of PG(3,2) are exactly the ones automatically generated by our external program. On the one hand, we easily check that all the computed spreads belonging to the list `spreads` actually verify the property `is_spread5` of being a spread. On the other hand, we prove that all sets of 5 lines verifying the property `is_spread5` belong to the proposed list `spreads`. Due to the size of the proofs and in order to make them accepted by the Coq proof assistant, we need to decompose this part of the proof into 35 specific cases. Each of them corresponds to one of the cases $l1 = \texttt{L0}, l1 = \texttt{L1}, \ldots, l1 = \texttt{L34}$. Eventually, using all these auxiliary lemmas, we prove the following property:

```
Lemma is_spread_descr : forall l1 l2 l3 l4 l5,
    leL l1 l2 && leL l2 l3 && leL l3 l4 && leL l4 l5 ->
    (is_spread5 l1 l2 l3 l4 l5) <-> In [l1;l2;l3;l4;l5] spreads.
```

In the previous statement, `leL` is a total order on the datatype `Line`, which expresses that $\texttt{L0} \leq \texttt{L1} \leq \ldots \texttt{L34}$ and allows to only consider ordered spreads of lines.

[9] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings_collineations.v
[10] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads.v

4.2 Classifying all Spreads of PG(3,2)

The next step consists in proving that all 56 spreads of PG(3,2) are isomorphic. It can be expressed by stating that there exists a collineation, i.e. an automorphism of PG(3,2) which respects incidence, between any two spreads of PG(3,2).

```
Definition are_isomorphic (s1:list Line) (s2:list Line) : Prop :=
  exists fp, exists fl, ((is_collineation fp fl) /\ (map fl s1 = s2)).
```

We show that the property `are_isomorphic`[11] is reflexive and transitive. Thanks to these results, we show that proving the equivalence can be achieved by simply proving that there exists a collineation (we actually build it) from the n-th element of the list to the ($n+1$ `mod` 56)-th element of the list `spreads`. Using this transitivity property and the collineations computed by our external program, we fairly easily prove the following statement:

```
Lemma all_isomorphic_lemma : forall t1 t2 : list Line,
  In t1 spreads -> In t2 spreads -> are_isomorphic t1 t2.
```

Overall, in this section, we formally proved in Coq that there are 56 labelled spreads in PG(3,2) and that there are all isomorphic.

5 Properties of Packings of PG(3,2)

In the following, we shall prove that there are 240 labelled packings in PG(3,2) and that they can be classified into two distinct classes.

5.1 Characterizing all Packings of PG(3,2)

A packing is defined using the predicate `is_packing7`[12] as a set of 7 spreads (each being a list of lines) which are disjoint and form a partition of the set of lines.

```
Definition is_partition7 (p q r s t u v: list Line) : bool :=
   (forall_Line
     (fun x => inbL x p || inbL x q || inbL x r || inbL x s ||
               inbL x t || inbL x u || inbL x v))
&& (forall_Line
      (fun x => negb (inbL x p && inbL x q && inbL x r &&
                      inbL x s && inbL x t && inbL x u && inbL x v))).
```

```
Definition is_packing7 (s1 s2 s3 s4 s5 s6 s7:list Line) : bool :=
  disj_7s s1 s2 s3 s4 s5 s6 s7 &&
  is_partition7 s1 s2 s3 s4 s5 s6 s7.
```

Checking that all computed elements of the list `packings` are actual packings is straightforward. Proving that all packings of PG(3,2) are in the list `packings` is a lot more challenging, especially because of the number of cases to deal with. In order to make it tractable in Coq, we prove several (56) lemmas of the form `statement_packings`[13] s for some s.

[11] https://github.com/magaud/PG3q/blob/master/pg32/pg32_spreads_collineations.v
[12] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings.v
[13] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings.v

```
Definition statement_packings s :=
  forall s2 s3 s4 s5 s6 s7 : list Line,
  In s2 spreads -> In s3 spreads -> In s4 spreads ->
  In s5 spreads -> In s6 spreads -> In s7 spreads ->
  ltS s s2 -> ltS s2 s3 -> ltS s3 s4 ->
  ltS s4 s5 -> ltS s5 s6 -> ltS s6 s7 ->
  is_packing7 s s2 s3 s4 s5 s6 s7 ->
  In [s;s2;s3;s4;s5;s6;s7] packings.
```

In each of them, we fix the first spread (e.g. s=`S0`) and then verify that all packings containing s as the first spread actually belong to the list `packings`.

```
  Lemma aux_S0 : statement_packings S0.
  [...]
  Lemma aux_S55 : statement_packings S55.
```

Finally, we agregate all 56 lemmas to obtain the following property:

```
Lemma is_packing_descr : forall s1 s2 s3 s4 s5 s6 s7 : list Line,
  ltS s1 s2 && ltS s2 s3 && ltS s3 s4 &&
  ltS s4 s5 && ltS s5 s6 && ltS s6 s7 ->
  In s1 spreads -> In s2 spreads -> In s3 spreads -> In s4 spreads ->
  In s5 spreads -> In s6 spreads -> In s7 spreads ->
  (is_packing7 s1 s2 s3 s4 s5 s6 s7) <->
     In [s1;s2;s3;s4;s5;s6;s7] packings.
```

In the above statements, `ltS` is an order on spreads, which implements the lexicographic order on spreads using the order on lines `ltL` as its basic order.

5.2 Classifying all Packings of PG(3,2)

In order to classify the packings of PG(3,2), we shall first prove that there are at most two distinct classes of packings in PG(3,2). This is achieved using the collineations relating packings provided in Sect. 3. Finally, considering two packings (one in each of the conjectured classes), we show that no collineation can transform the first one into the second one.

5.2.1 There are at most 2 Classes of Packings in PG(3,2)

When considering packings, the relation `are_isomorphic`[14] is a bit more complex as collineations may transform a packing into a packing whose spreads are not sorted in increasing order any more. Therefore we enforce that the images of the spreads computed using f_l must be sorted with respect to the relation `ltL`.

```
Definition are_isomorphic
  (p1:list (list Line)) (p2:list (list Line)) : Prop :=
    exists fp, exists fl,
      is_collineation fp fl /\
      forall s:(list Line), In s spreads -> In s p1 -> In
      (sort (map fl s)) spreads /\ In (sort (map fl s)) p2.
```

[14] https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings_collineations.v

Once again, we prove that the property `are_isomorphic` is reflexive and transitive. This allows to prove that all elements of a class are isomorphic by performing a circular permutation, simply proving that there exists a collineation (which was built explicitly by our external specification and proofs generating program) from the n-th element of the list `class0` (resp. `class1`) to the $(n+1 \texttt{ mod } 120)$-th element of the list `class0` (resp. `class1`).

```
Lemma all_isomorphic_lemma0 :  forall t1 t2 : (list (list Line)),
  In t1 class0 -> In t2 class0 -> are_isomorphic t1 t2.

Lemma all_isomorphic_lemma1 :  forall t1 t2 : (list (list Line)),
  In t1 class1 -> In t2 class1 -> are_isomorphic t1 t2.
```

At this stage, we only proved that there are at most two classes of packings in PG(3,2). The last step of the proof consists in proving that the two classes `class0` and `class1` are distinct. To do that, we choose two packings, e.g. `PA0` and `PA1`, one in each of the supposed classes. We then generate all collineations of PG(3,2) and verify that none of these collineations allows to go from the packing `PA0` to the packing `PA1`.

5.2.2 Characterizing all collineations of PG(3,2)

So far we defined a collineation as a pair of two bijective functions f_p and f_l and a property that these functions respect the incidence relation. We shall see that a collineation (f_p, f_l) can be exactly characterized by simply defining the images of the four following points `P0`, `P1`, `P3` and `P7`. This relies on the property that there are only three points by line and that collineations are bijections which respect the incidence relation.

Let us start by choosing an image for the first point `P0`. Let us then choose an image for the second point `P1`. Then the image of the point `P2`, which is on line `L0`=(`P0P1`) is imposed. It is the third point of the line generated by f_p `P0` and f_p `P1`. Let us choose the image of the third point `P3`, which lies outside line `L0`. The images of points `P4`, `P5`, `P6` (see Fig. 2 for an visual interpretation of the process) are imposed by the rules of the projective spaces and the collineation properties. We can then choose a fourth point `P7`. This point is outside the plane generated by `P0`, `P1` and `P3`. Once these four images f_p `P0`, f_p `P1`, f_p `P3` and f_p `P7` are chosen, the images of all remaining points `P8`, `P9`, `P10`, `P11`, `P12`, `P13`, `P14` are imposed as being third points of some lines defined by the combination of images of the four initial points `P0`, `P1`, `P3` and `P7`. In addition, the images of all lines are fully determined as well. Indeed, the image of the line going through points A and B is the line going through points f_p A and f_p B.

From a combinatorial point of view, we can choose the image of `P0` by f_p among 15 points. The image of `P1` by f_p can be chosen among 14 points (all points except `P0`). The image of `P3` by f_p can be chosen among 12 points (all points except those on line `L0`, which contains points `P0`, `P1` and `P2`). Finally, the image of the fourth point `P7` by f_p can only be chosen outside of the images of points `P0`, `P1`, `P2`, `P3`, `P4`, `P5`, and `P6`, thus leaving only 8 options available. Once the images of these four points are chosen, the collineation is fully characterized because both f_p and f_l must be bijective and that they must respect the incidence relation. This means that there are $15 \times 14 \times 12 \times 8 = 20\,160$ different ways to define a collineation of PG(3,2).

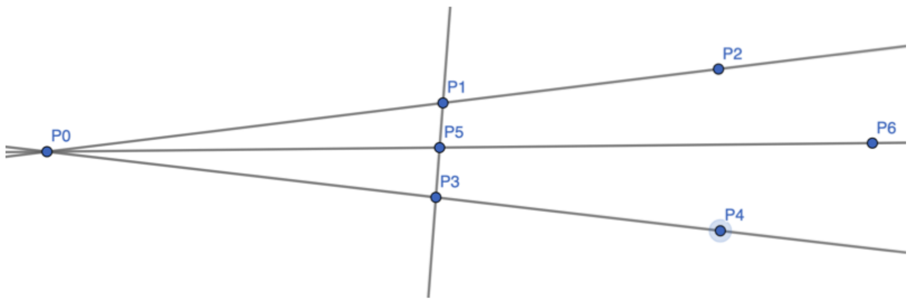

Figure 2 Describing a collineation of PG(3,2) can be achieved by simply providing the images of points `P0`, `P1`, `P3` and `P7`.

Our external specification and proofs generating program takes care of computing all possible collineations of PG(3,2). It generates some very large files: `pg32_automorphisms.v`[15] (where the 20 160 collineations are enumerated), `pg32_automorphisms_inv.v` as well as files `pg32_collineationsX.v` and `pg32_decompX.v`, with X ranging from 0 to 14. The list of all collineations is splitted into smaller lists of size 96 in order to be able to handle the proofs in Coq. Each subset of 96 collineations corresponds to specific collineations whose images of points `P0` and `P1` are the same.

```
Definition all_c0 := [
  (fp_0, fl_0); (fp_1, fl_1); ... ; (fp_94, fl_94); (fp_95, fl_95)].
[...]
Definition all_collineations :=
  all_c0 ++ all_c1 ++ all_c2 ++ ... ++all_c208 ++ all_c209.
```

On the one hand, for each of these subsets, we can check that the given collineations actually verify the property `is_collineation`. On the other hand, we verify that all collineations which verify the following conditions: f_p `P0` = `PX` and f_p `P1` = `PY` actually belong to the corresponding subsets of collineations, namely `all_cZ` where $Z = 14 \times X + Y$. As an example, all collineations which respect the conditions f_p `P0` = `P8` and f_p `P1` = `P2` belong to the subset of collineations `all_c114`.

```
Lemma is_collineations_descr_B_P8_P2 :
  forall fp fl, is_collineation fp fl -> fp P0 = P8 -> fp P1 = P2 ->
  In (fp,fl) all_c114.
```

To prove that a specific collineation (`fp`,`fl`), characterized by `fp P0` = `P8`, `fp P1` = `P2`, ..., actually corresponds to an element of the list `all_c114`, we rely on extensionality and thus add the two following safe axioms to our development:

```
Lemma fp_ext: forall (fp:Point->Point) (fp':Point->Point),
  (forall (p:Point), fp p = fp' p) -> fp = fp'.
Admitted.
```

```
Lemma fl_ext : forall (fl:Line->Line) (fl':Line->Line),
  (forall (l:Line), fl l = fl' l) -> fl = fl'.
Admitted.
```

[15] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_automorphisms.v`

Splitting the main statement characterizing all collineations into 210 smaller statements allows to handle the proofs in Coq. Thankfully, all these 210 statements are almost automatically generated. Only some minor parts, which we plan to fully automate in a near future, require to be fixed by hand (e.g. unfolding some specific constants or making the naming of lemmas coherent to prove the general statement). The last step consists in agregating all these lemmas to obtain the following statement, which explicitly characterize all collineations of PG(3,2).

```
Lemma is_collineations_descr : forall fp fl,
  is_collineation fp fl <-> In (fp,fl) all_collineations.
```

5.2.3 There are exactly 2 Distinct Classes of Packings in PG(3,2)

Now that we have a list of all collineations of PG(3,2) at our disposal, we can traverse it to verify that none of these collineations allow to transform a packing of the class `class0`, say `PA0` into a packing not in `class0`, say `PA1`. The proof simply consists in assuming, for each collineation that they allow to transform the packing `PA0` into the packing `PA1` and exhibit a contradiction. As we must check all collineations, the Coq file has more that 20 160 lines.

```
Lemma not_iso : ~are_isomorphic PA0 PA1.
```

The statement `not_iso`[16] shows that the two classes of packings `class0` and `class1` are distinct. We can conclude that the 240 packings of PG(3,2) belong to two distinct classes, each of these classes containing exactly 120 elements.

6 Discussion

The Coq development is quite large. It contains more than 50 files. Thankfully, most for them are automaticaly generated. It consists in more than 317 345 lines of specifications and proofs, among them more than 290 000 are proof steps. Some files have about 20 000 lines, which makes them difficult (or at least very slow) to handle in an editor for Coq. Compiling the whole development requires about 13 hours (584 minutes on a Intel (R) Core(TM) i5-4460 CPU @ 3.20GHz with 32GB of memory). Therefore it is important that all proofs are as concise as possible and the development must be well structured as changes in the structure may result in several hours of compilation before being able to resume interactive theorem proving. In the following, we present some proof engineering techniques which proved very useful in our development. We also propose some possible improvements to our work.

6.1 Proof Engineering

6.1.1 Using bool instead of Prop

As we work with finite types, equality and the other relations that we use are decidable. We can directly implement such relations as operations producing elements of the boolean datatype `bool`. This is more convenient than defining them as operations producing elements of type `Prop` together with a decidability property: $\forall\ x\ y, \{x = y\} + \{\neg x = y\}$. This practical approach is inspired by the ssreflect [11] and the mathematical components [17] libraries. In this setting, logical reasoning (eliminating conjunctions or disjunctions) is a bit more technical. However this makes most proofs much easier to complete by simply computing a

[16] `https://github.com/magaud/PG3q/blob/master/pg32/pg32_packings_two_distinct_classes.v`

boolean value and checking that it is equal to `true`. We could push this technique further and replace all remaining occurences of Leibnitz equality (e.g. in the axiom `uniqueness`) with the two bool predicates `eqP` and `eqL`, respectively implementing equality on `Point` and `Line`.

6.1.2 Optimizing proofs

We design some optimization techniques for generating and checking proof terms. We focus on the current goal, applying some sort of locality principle which means that we try to prove a (sub-)goal the very first time we face it. This means sequences of tactics such as

```
intros a; case a; intros H;
  try (exact (degen_bool )_ H).
solve_goal.
```

must be replaced by more efficient sequences like

```
intros a; case a; intros H;
  solve [(exact (degen_bool )_ H |solve_goal].
```

In this simplified example, we try to apply the tactic `(exact (degen_bool )_ H)` for a subgoal and then we switch to the next subgoal. Eventually we solve the remaining subgoals using the `solve_goal` tactic. The idea here is to solve the goal the first time we encounter it. It is achieved by having several possibilities of tactic applications to solve the goal (this corresponds to the `solve [t1|t2|t3]` syntax). The order of the tactics `t1`, `t2` and `t3` can be highly significant as well: we should always call the tactic which is the most successful one on such subgoals first.

As we face a huge number of cases, we need to design extremely efficient prototype tactics on some specific subgoals and apply them automatically to all the subgoals at stake. Fine tuning the tactics rapidly is the key to making the proofs faster to complete.

Finally, Coq provides some sort of task parallelism in the form of the `par` tactical. It is very useful to deal with all the sub-goals of a proof, once we figure out how to prove the first one. The generic tactic proving the first goal, say `mytactic` can be easily applied to all sub-goals in parallel (in some cases, we have 35x35=1225 or more goals to deal with) by simply writing `par:mytactic`.

6.1.3 Without Loss of Generality

Most proofs are highly branching. For instance, performing case analysis on all three lines to prove the lemma `a3_3` leads to $35^3 = 42875$ cases. In order to make the proof more tractable, we propose a new tactic named `wlog`[17], which implements the *without loss of generality* principle, as it is described in [13]. This allows to reduce the number of cases to solve explicitly. To use it, we build a virtual order on the points and lines, simply mapping point `Pi` (resp. line `Li`) to the value i of its index and then extend statements of the form $\forall l1, l2 : \texttt{Line}, \ldots$ to $\forall l1, l2 : \texttt{Line}, l1 \leq l2 \rightarrow \ldots$

Surprisingly, using the without loss of generality tactic forces us to generalize our statement for Pasch axiom to accommodate all cases, depending on the order in which we consider points A, B, C, and D, as shown in Fig. 3. The usual conclusion of Pasch axiom:

[17] `https://github.com/magaud/PG3q/blob/master/generic/wlog.v`

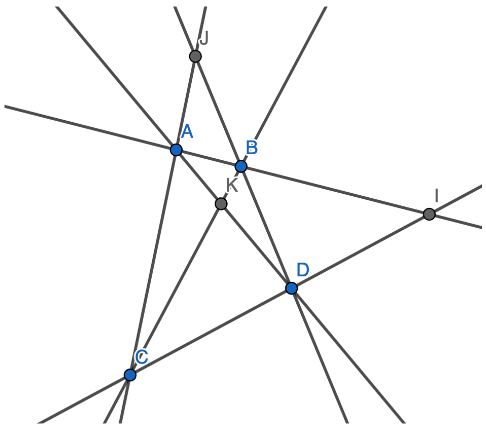

Figure 3 An illustration of the new form of Pasch axiom used to deal with symmetries.

```
(exists I:Point, incid_lp I lAB && incid_lp I lCD) ->
        exists J:Point, incid_lp J lAC &&  incid_lp J lBD.
```

is transformed into a conjunction of two existential properties:

```
(exists I:Point, incid_lp I lAB && incid_lp I lCD) ->
        (exists J:Point,  (incid_lp J lAC && incid_lp J lBD)) /\
        (exists K:Point, (incid_lp K lAD && incid_lp K lBC)).
```

The principles behind the tactic `wlog` were also extremely useful when dealing with spreads and packings, especially when checking inside Coq which sets of lines are actual spreads and which sets of spreads are actual packings.

6.2 Improvements

While carrying out such a proof development, one of the main difficulties is to decide what a *small* Coq proof is. Our first experiments crashed because we assumed Coq will handle very large specifications and proofs easily. Instead we needed to scale down our proofs and decompose them a lot to make sure they can be compiled. The current decomposition is probably too strong, but it has the advantage of being tractable by Coq.

Most definitions and properties used in this development are first order. So it would be interesting to implement the same formal description in a first-order prover such as Z3 [10]. It can also be of interest to use first-order tools such as [1] provided in Coq.

From a specification point of view, as collineations can be simply characterized by the images of only four points, we shall study how to remove the bijection on lines from the definition of the collineation and reconstruct it from the bijection on points. This would make the proof development much smaller and reduces the number of objects we are handling simultaneously. In addition, some combinatorial arguments could be used to simplify the specification of spreads `is_spread5` and packings `is_packing7`, e.g. removing the disjointness condition for the input lists. Finally, most proofs are very similar to one another. In the near future, we shall study how symmetry arguments could help reduce the number of cases to handle. We shall also investigate how to carry out circular permutations of the set of points so that some proofs can be factorized by simply specifying the first point or line at stake and then rotating the statement to obtain the other cases.

7 Conclusion and Future Work

In this work, we show how to formalize in Coq the spreads and packings of PG(3,2). Using an external specifications and proofs generating program, we build automatically all the spreads and packings, as well as all the collineations of PG(3,2). We then easily verify that these generated sets of lines (resp. spreads) are actual spreads (resp. packings). We also successfully prove that they are the only ones. In addition, we classify the spreads and packings, showing that there is only one class for the 56 spreads and that the 240 packings are splitted into two classes of 120 elements. Showing that these two classes are distinct required generating and characterizing in Coq all the 20 160 collineations of PG(3,2).

All the proofs carried out in this work are very large. A single case analysis on a point generates 15 cases, and a single case analysis on a line generates 35 cases. In order to let Coq deal correctly with all these proof scripts, we had to decompose our statements into several smaller lemmas, which could each be independently handled by Coq. During this study, we faced case analysis with a huge number of cases as well as debugging proof script with thousands of sub-goals. We propose some proof engineering techniques to make Coq process the files more easily e.g by directly providing witnesses or by pruning the proof tree by using a *without loss of generality* principle.

So far, we only address properties and transformations which remain in the same (projective) space. We are currently working on generating specifications of projective spaces automatically in order to easily have a formal description of two different projective spaces and thus to be able to formally describe constructions as the Bruck-Bose construction which allows to build translation planes from projective planes [5]. In parallel, we plan to formalize the spreads and packings of PG(3,3) and their properties, as presented in [3]. Handling the projective space PG(3,3) which features 40 points and 130 lines (instead of 15 points and 35 lines in the present case study) would require more decisive specification and proofs techniques. Indeed, PG(3,3) has 8 424 distinct spreads and 12 130 560 collineations, compared to the mere 56 spreads and the 20 160 collineations of PG(3,2). One of the main issues to tackle will be to avoid the exhaustive enumeration of all objects at stake. This could be achieved by using algorithms such as McKay algorithm [18] or Faradžev-Read algorithm [19].

References

1 Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. Verifying SAT and SMT in Coq for a Fully Automated Decision Procedure. In *International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT'11)*, 2011.

2 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin/Heidelberg, May 2004. 469 pages.

3 Anton Betten. The packings of pg(3, 3). *Designs, Codes and Cryptography*, 79(3):583–595, 2016. `doi:10.1007/s10623-015-0074-6`.

4 David Braun, Nicolas Magaud, and Pascal Schreck. Formalizing Some "Small" Finite Models of Projective Geometry in Coq. In Jacques Fleuriot, Dongming Wang, and Jacques Calmet, editors, *Proceedings of Artificial Intelligence and Symbolic Computation 2018 (AISC'2018)*, number 11110 in LNAI, pages 54–69, September 2018. URL: `https://hal.inria.fr/hal-01835493`.

5 Richard Hubert Bruck and Raj Chandra Bose. The construction of translation planes from projective spaces. *Journal of Algebra*, 1(1):85–102, 1964. `doi:10.1016/0021-8693(64)90010-9`.

6 Francis Buekenhout, editor. *Handbook of Incidence Geometry.* North Holland, 1995.

7 Frank Nelson Cole. Kirkman parades. *Bull. Amer. Math. Soc.*, 28(9):435–437, December 1922. URL: `https://projecteuclid.org:443/euclid.bams/1183485271`.

8 Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.13.2*. INRIA, 2021. URL: `http://coq.inria.fr`.

9 Harold Scott Macdonald Coxeter. *Projective Geometry*. Springer Science & Business Media, 2003.

10 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. `doi:10.1007/978-3-540-78800-3_24`.

11 Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008. URL: `http://hal.inria.fr/inria-00258384/`.

12 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the coq system. Research Report RR-6455, Inria, Saclay Ile de France, 2015. URL: `https://hal.inria.fr/inria-00258384`.

13 John Harrison. Without loss of generality. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 43–59. Springer, 2009. `doi:10.1007/978-3-642-03359-9_3`.

14 James William Peter Hirschfeld. *Finite projective spaces of three dimensions*. Oxford mathematical monographs. Clarendon Press ; New York : Oxford University Press, Oxford, 1985.

15 R.H. Jeurissen. Special sets of lines in PG(3, 2). *Linear Algebra and its Applications*, 226-228:617–638, 1995. Honoring J.J. Seidel. `doi:10.1016/0024-3795(95)00200-B`.

16 Nicolas Magaud. Spreads and packings of pg(3,2), formally! *Electronic Proceedings in Theoretical Computer Science*, 352:107–115, December 2021. `doi:10.4204/eptcs.352.12`.

17 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021. `doi:10.5281/zenodo.4457887`.

18 Brendan D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2):306–324, 1998. `doi:10.1006/jagm.1997.0898`.

19 Ronald C. Read. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. In B. Alspach, P. Hell, and D.J. Miller, editors, *Algorithmic Aspects of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 107–120. Elsevier, 1978. `doi:10.1016/S0167-5060(08)70325-X`.