# Project Summary

* Mostly good, could use some example solutions or partial solutions visualized

Kanoodle Jr. is a game where pieces of various colours and shapes need to be placed on a 5x5 grid. Initial puzzle configurations require certain squares to be coloured a certain way. This restricts where the pieces can be placed. This project aims to model any given initial configuration as a SAT theory, and a solution will correspond to the proper placement of all pieces in the game.

Here, you can see the board with an initial configuration and the set of pieces for the Kanoodle Jr. game.

# Propositions

* Complete.

There are three main propositions for the Kanoodle Jr. encoding:

* **PieceConfig(col, config)**: Piece of colour **col** is using configuration **config** (e.g., rotation). Most have 4 configurations, but some have 8 (flip it over).
* **PlacePiece(col, config, x, y)**: Piece of colour **col** and configuration **config** is at location (**x**,**y**). One location on the piece is considered the origin (0,0), and the locations on the rest of the piece are defined by its shape.
* **PlaceColour(col, x, y)**: The colour at location (**x**,**y**) is set to **col**. This is used to tie together the overlapping pieces, find the final solution, and restrict what colours are forced to be where as part of the initial puzzle.

# Constraints

* Definitely not complete. Each of the subsections should describe the constraint in both logic and English. No need to show every variant of them, though.

## Only one configuration for a colour

## Don’t allow pieces out of bounds

## At most one piece can be placed at a location

## Can only place a piece in one place

## If a piece is picked for a location, then the appropriate configuration is chosen

## Every location can take on at most one colour

## If a piece is placed, then it forces the colour of the cells that make up the piece

## Initial board configuration is satisfied

# Model Exploration

* Very weak. Suggestions for more:
  + Debugging that occurred throughout the project.
  + Screenshots of the different visualizations that were produced.
  + New ways of solving the problem that weren’t ultimately used in the final version (e.g., ways to call the solver to check if two pieces can be placed in a specific location).
  + If model counting was explored, then reporting how long it actually took to solver (versus just finding one solution)

*List all the ways that you have explored your model – not only the final version, but intermediate versions as well. See (C3) in the project description for ideas.*

## Visualizing the Model

Just printing the solution from Bauhaus was tough to see what was going on, so we used the **tabulate** library to illustrate where pieces go. For example:

Table

Description automatically generated with medium confidence

Brackets show the colour of the square, and capital shows the piece colour placed.

Ideas on how to improve this for the final:

* Move to shaded colour background for the pieces that are placed.
* ..

## Fixing the Overlap

At one point, we were facing an issue where pieces would be overlapping. An example:

A picture containing table

Description automatically generated

To fix this, we looked into adding constraints that prevented overlap. This ultimate lead to a new proposition (the third one discussed above) that says what colour a location should be.

# Jape Proofs

* Complete and full grade.

Because the model is too large to fit everything, we will prove sequents over a far reduced size of Kanoodle board: 2x2. Also, we will only consider pairs of pieces. Here we detail the proposition symbols and sequents proved in the Jape file.

## Proposition Symbols

* **Pij**: Piece 1 is placed at location (i,j)
* **Qij**: Piece 2 is placed at location (i,j)
* **Rijk**: Cell location (i,j) is coloured with piece k

## (0) Base Encoding Premises

All of the proofs will follow similar base premises.

### Piece Implies Colours

* P11→(R111∧R211)
* P12→(R121∧R221)
* Q11→(R112∧R212)
* Q12→(R122∧R222)

### Piece is Placed Somewhere

* P11∨P12
* ¬(P11∧P12)
* Q11∨Q12
* ¬(Q11∧Q12)

### Cell Can’t Have Two Colours

* ¬(R111∧R112)
* ¬(R121∧R122)
* ¬(R211∧R212)
* ¬(R221∧R222)

## (1) Coloured Wrong Cell Forces Placement

* {*base encoding*}, ¬R221 ⊢ P11

## (2) Coloured Cell Forces Placement

* {*base encoding*}, R222 ⊢ P11∧Q12

## (3) Vertical and Horizontal Don’t Fit

* {*base encoding, but Q is vertical*} ⊢ ⊥

# First-Order Extension

* Mostly all set. Suggestions to bring it to 100%
  + More explanation of a high-level summary of how the extensions works (e.g., Going to use objects for blablabla, and the constraints will be changed to reflect the newly introduced objects. Loops in the code covering all versions of a proposition will turn into constraints with a universal quantifier. Etc.”)
  + Predicates are good.
  + Constraints could use some more versions (but no need for all) from the ones above. Roughly 5 is a good place to hit.

## Predicates

These will mirror closely to the propositions defined above. We have objects in the first-order setting that correspond to colours, configurations, and coordinates. These “types” will also have a predicate defined for them.

* **Colour(col)**: object **col** is a colour
* **Config(c)**: object **c** is a configuration
* **Coord(c)**: object **c** is a coordinate
* **PieceConfig(col, config)**: Piece of colour **col** is using configuration **config** (e.g., rotation).
* **PlacePiece(col, config, x, y)**: Piece of colour **col** and configuration **config** is at location (**x**,**y**).
* **PlaceColour(col, x, y)**: The colour at location (**x**,**y**) is set to **col**.

## Constraints

### More complex predicates use the right types

* Arguments to the PieceConfig predicate are Colour and Config objects:  
   col. config. ( PieceConfig(col, config) (Colour(col) Config(config)) )
* Arguments to the PlacePiece and PlaceColour are similarly defined:  
   col. config. x.y. ( PieceConfig(col, config, x, y) (Colour(col) Config(config) Coord(x) Coord(y)) )  
   col. config. x.y. ( PieceConfig(col, x, y) (Colour(col) Coord(x) Coord(y)) )

### Every location can take on at most one colour

Variant 1:  
 x.y. ( (Coord(x) Coord(y)) ( col1.col2. (col1 = col2) (PlaceColour(col1, x, y) PlaceColour(col2, x, y))))

Interpretation:  
For all pairs of objects, if they are coordinates, then for every pair of colours, either they are the same or the PlaceColour doesn’t hold for both colours.

Variant 2:  
 x.y. ( (Coord(x) Coord(y)) ( col1.col2. ((col1 != col2) (PlaceColour(col1, x, y) PlaceColour(col2, x, y)))))

Variant 3:  
 x.y. col1.col2. ((col1 = col2) (PlaceColour(col1, x, y) PlaceColour(col2, x, y)))

There is no need to include the types of the objects x, y, col1, col2, because of the constraint above that defines the types for PlaceColour.