Logo

Description automatically generated

**Generating BRIO™ layouts**

Alexandre Kings

Supervisor: Tom Spink

Contents

[2 Introduction 3](#_Toc130205287)

[3 Objectives 4](#_Toc130205288)

[4 Characterization of the BRIO™ pieces 5](#_Toc130205289)

[4.1 Required characteristics 5](#_Toc130205290)

[4.2 Capturing the shapes of pieces 5](#_Toc130205291)

[4.3 Piece splitting into parts 9](#_Toc130205292)

[4.4 Bounding circles 9](#_Toc130205293)

[4.5 Extension idea 10](#_Toc130205294)

[4.6 Summary of piece characteristics 10](#_Toc130205295)

[4.7 Pieces determined 11](#_Toc130205296)

[5 User interface 13](#_Toc130205297)

[5.1 Tools 13](#_Toc130205298)

[5.2 Representation of generated circuits 13](#_Toc130205299)

[5.3 Preventing piece overlaps 15](#_Toc130205300)

[5.4 Addition of zoom and drag on canvas 15](#_Toc130205301)

[5.5 Piece picker 16](#_Toc130205302)

[6 Track Generation 18](#_Toc130205303)

[6.1 Tools 18](#_Toc130205304)

[6.2 Algorithm description 18](#_Toc130205305)

[6.3 Piece positioning 19](#_Toc130205306)

[6.4 Collisions 20](#_Toc130205307)

[6.5 Ascending tracks 22](#_Toc130205308)

[6.6 Multi-level tracks 22](#_Toc130205309)

[6.7 Optimisations (discuss heuristic here). 22](#_Toc130205310)

[6.8 Possible improvements 22](#_Toc130205311)

[7 Critical appraisal 24](#_Toc130205312)

[8 Conclusion 25](#_Toc130205313)

[9 Running the programme 26](#_Toc130205314)

[10 References 27](#_Toc130205315)

# Introduction

BRIO™ is a Swedish brand which has been making and selling wooden railway pieces since 1884 [1]. Pieces are link together using pin and hole connectors, and circuits can be created for small wooden trains to move on. These toys are particularly popular amongst families with children, who can end up gathering rather large collections of railway pieces. The aim of this project is to create a tool that makes it easier for people to construct unique BRIO™ layouts using their collection of pieces. The application developed includes a web interface for users to specify their pieces, and allows them to automatically generate closed circuits, that they can visualise on a simple display. Procedural generation is used to obtain tracks that can be built from a certain collection of pieces.

Procedural generation is commonly used in the field of video games and has two key motivations. In early video games, when computational memory was very limited, procedural generation was used as a way of saving large amounts of space. In 1982, *River Raid* used procedural generation to create interesting terrains, which could not have been stored otherwise on a basic 4 kilobyte Atari 2600 cartridge [2]. More recently, other games like the very famous *Minecraft* [2], and *No Man’s Sky* [3], use procedural generation from a single seed number to create huge 3D universes.

The second key usage of procedural generation is to facilitate tedious, repetitive tasks and provide variety. This is the motivation for this program, where procedural generation is used to facilitate the construction of interesting railway layouts. One of the core ideas of this project is the necessity to generate closed-loop circuits. The closest kind of procedural generation to this is found in racing games, where the tracks are generated as close loops to mimic real-world racing tracks. Some techniques used for generating such racing tracks include the positioning of sets of control points in a closed loop, with Bezier curves employed to connect these points together [4]. Another technique for generating racing tracks, presented in [5], involves the use of a human-like AI agent that races through trial circuits in order to evaluate and improve them.

However, the requirement for close loop circuits is only part of the problem. The set of input pieces specified by the user, and the discrete nature of BRIO™ pieces, each having a specific length, curvature, ascent and number of connection points, are important restrictions that need to be taken into account. It is the inclusion of these constraints that make this a one-of-a-kind procedural generation problem. This is, as far as the author knows, the first automatic BRIO™ track generator ever made.

# Objectives

A set of objectives was initially determined. These objectives are ranked in three categories: *primary objectives* are the most important ones, then come *secondary objectives*, and finally *additional objectives*.

**Primary objectives**

One of the most important goals is the creation of an interface for users to select pieces and to visualise generated tracks on a 2D display. The option to display generated layouts in 3D was considered at the start of the project. This would make it useful when it comes to visualising multi-level circuits, allowing users to clearly see which pieces are placed above others. However, after a discussion with Tom, it was decided that a simple 2D display would be sufficient to show the layouts built, and the emphasis was put on the generation of interesting track layouts rather than on the display of generated tracks.

Another one of these primary objectives is the generation of non-trivial tracks that contain at least one closed loop. Tracks that contain no closed loops are not considered to be particularly interesting, and such tracks were completely omitted.

The last of these objectives is one regarding performance: “*The tracks should be generated in a reasonable time*”. Efficiency of the track-generating algorithm was a primary concern throughout the development of the program and a lot of decisions were made in the hopes to get the program to run faster for large and more complex sets of input pieces.

**Secondary objectives**

These objectives are valuable additions to the programme, making it more interesting and valuable. The secondary objectives include the addition of pieces containing more than one connector, allowing for the creation of multi-loop circuits. Ascending pieces should also be added, allowing for multi-level circuits. The addition of single-connector pieces was also mentioned.

Another one of the secondary objectives was to make it possible for users to specify the dimensions of their room, and only allow the generation of tracks that fit in the room specified.

The last one of the secondary objectives specifies that the connection between any two pieces is not perfect, and circuits which have ends that are “close enough” to each other should be counted as valid circuits. This proved to be an absolute necessity for closing any track, as the vast majority of circuits that can be generated using this tool have ends that do not perfectly meet.

**Additional objectives**

These objectives were interesting ideas that could be included in the project. The first additional objective mentions the addition of more sophisticated pieces to pick from. The second one stipulates that support blocks to hold tracks that are not touching the ground should be automatically positioned. Another additional objective says that the number of loops contained in a track should be specifiable by the user. Finally, the use of WASM and multi-threading are mentioned as ways to make the generation faster.

# Characterization of the BRIO™ pieces

A key aspect of this project was to determine the characteristics of a collection of BRIO™ pieces in such a way that a computer program could understand. The description of pieces is of major importance, as it is this collection of pieces that is used in the rest of the program, both for the generation of circuits and for the display of these circuits. For this reason, this project started with finding a good data structure to represent pieces, and then determining the specific characteristics of an initial set of BRIO™ pieces. This section discusses the way in which a library of BRIO™ pieces was put together.

## Required characteristics

The set of characteristics for each piece had to be carefully determined, to provide a program with enough information to generate layouts using these pieces. More specifically, the track-generating algorithm should be able to determine the following things:

- position and orientation of pieces,

- ability for pieces to connect with each other,

- collisions between pieces.

Also, the data structure representing pieces must be flexible enough to allow for the characterisation of more complex pieces, like switches and ascending pieces. From these requirements, some important attributes appeared.

Each piece is given a list of connectors, each determined by a position, a direction and a type (pin or hole). This allows to connect pieces together, by rotating and translating the pieces so that the two connectors align. The list of connectors can be of any size, allowing for the construction of pieces with one, two, three or more connectors.

## Capturing the shapes of pieces

To allow for the computation of collision between pieces, the general shape of each piece had to be captured. The geometry of pieces had to be modelled in a way simple enough to allow fast and efficient collision calculations. The first option considered was outline each piece by a single polygon. However, this idea was not kept because the Separating Axis Theorem (SAT) typically used to determine collisions only works for convex shapes, and curved pieces and switches are not convex but concave. Instead of a single concave polygon, a set of multiple cleverly picked convex ones could have been used for these pieces. It however seemed complicated to find the right set of polygons to use for some pieces, and having a set of complex polygons for each piece would likely negatively impact the performance of collision-detection calculations. It is especially important that collisions calculations are fast, considering that, to place one railway piece, the collision with each of the previously placed pieces must be calculated first.

Another option considered was to approximate piece’s shapes by giving them a set of circles. Collision between two circles is very efficiently and very easily determined, using Pythagoras’ Theorem, and comparing the square of the sum of their radii to the square of their distance. However, this idea was discarded quickly because using circles to approximate rectangular shapes would be unnatural, and a rather large number of circles would need to be used to approximate each shape. Performance would have ended up being negatively impacted from this.

The option chosen was to give each piece a set of rectangles, or Oriented Bounding Boxes (OBBs). Curves in BRIO™ pieces are 45-degree arcs of different radii [6]. These are relatively small curves and are well approximated using a small set of rectangles.

To help with the determination of OBBs for a set of BRIO™ pieces, it was decided to build a simple graphical tool using the JavaScript Canvas API. This choice of making the tool web-based was made so that it could potentially be included, later, as part of the track-generating webpage, allowing users to create their own railway pieces and generate tracks out of these pieces.

A first version of this tool was made, with the idea that the connectors and OBBs for each piece could be manually positioned, following the shape provided by an image of the piece displayed on the background. An example of the determination of a piece using this tool is shown in Figure 1 (the ability to determine connectors had not been added to the tool yet).



Figure 1: Tool initially developed for the determination of piece characteristics. Example of rectangles (in red) manually positioned, with an image of the BRIO™ A track in the background. Rectangles are determined by 5 numbers: x and y positions of their centre, width, height, and their orientation angle around their centre.

It was however decided that this way of manually determining the OBBs and connectors for each piece would not be satisfactory, for a few reasons. Firstly, while the OBBs could be quite easily positioned for simple rectangular pieces, it was difficult to position them properly for curved pieces. Secondly, imperfections in the determination of the orientation of connectors for curved pieces would likely result in strange-looking tracks getting generated. The last, and arguably the most important of these reasons, was the scale used for the pieces. The BRIO™ Wooden Railway Guide website provides 2D models of pieces that can be loaded on SketchUp, a free modelling program [7]. Images of the pieces were exported as PNG files from SketchUp, using these resources. This way in which images of the pieces were obtained did not conserve scale, and there did not appear to be a trivial way to normalise the scales used for the different pieces. For these reasons, this version of the tool was soon abandoned.

The idea for the next version of this tool was the following: instead of using images of pieces as a basis to determining their characteristics, pieces and their characteristic would be determined in a solely geometrical way, using their known dimensions. Not only does a purely geometrical approach lead to perfectly aligned pieces, it also naturally addresses the scale problem. The dimensions of pieces were found on the BRIO™ Wooden Railway Guide [6]. Each JavaScript Canvas unit is made to correspond to a millimetre of a real piece. Using the fact that all typical BRIO™ pieces are 40mm wide, a function was made to create an OBB of fixed width between two points.

Rectangular pieces are then easily determined by two points only, one on each of the piece’s connectors. From these points, a single OBB is automatically generated, surrounding the bulk of the piece. The two connectors are oriented along the single axis of the piece. The tool also allows to determine the type of each connector, pin or hole. Figure 2 shows an example of determination of the characteristics of a rectangular piece using the tool described.

A picture containing box and whisker chart

Description automatically generated

Figure 2: Determination of the characteristics of the A piece, using version 2 of the tool. HTML inputs, with minimal CSS styling, are used to position the different components of each piece.

Curved pieces are determined with the same tool, using a set of up to four control points which are used to construct a Bezier curve. This decision of using Bezier curves to determine the characteristics of pieces was made to leave the maximum degree of freedom for constructing complex pieces, if required later in the project. The OBBs for the piece are automatically generated and positioned along the defined Bezier curve. An example of the determination of the characteristics of a curved piece with different numbers of OBBs is shown in Figure 3.



Figure 3: Determination of the E piece using the Bezier curve plotting tool, with 2, 6 and 20 OBBs. The green points represent the four Bezier control points, and the arrows show to position and direction of connectors.

The use of Bezier curves makes it easy to determine simple curved pieces using the arc circle approximation formulae. For an arc of angle starting at point and ending at point , the coordinates of the two Bezier control points are given by:

)

)

Using , an approximation of an arc circle with an error of in the radius is obtained, which is perfectly acceptable in this case [8]. Figure 4 shows an example of a Bezier curve obtained using these formulae.



Figure 4: Approximation of an arc circle using a Bezier curve [8].

The larger the number of rectangles, the better defined the piece (see Figure 3), but also the longer it will take the program to compute collisions between pieces. It was decided that two OBBs would be sufficient to determine the shape of curved pieces, with an acceptable degree of precision. This decision was made to keep the program as performant as possible.

## Piece splitting into parts

To account for switches, it was decided to give each piece a set of separate parts, with parts composed of a certain number of OBBs. Switches are then built using two parts, with one of each part’s ends coinciding at the position of the first connector. Figure 5 shows the determination of the characteristics of a simple curved switch.



Figure 5: Determination of the characteristics of the L piece using the Bezier curve plotting tool. Bezier control points are shown in green, and the arrows show the position and direction of connectors.

This separation of pieces into a set of parts is especially convenient, allowing for the description not only of switches, but also of ascending pieces. BRIO™ pieces can only ascend to multiples of a certain discrete amount. This leads to the use of an integer to represent the level at which each part of a piece is at. This way, a simple straight ascending piece is represented by dividing it into two equal rectangles, one placed at level 0, the other at level 1. Pieces’ connectors are given a level in the same way as parts are.

## Bounding circles

Later in the program, a bounding circle needed to be added around the bulk of each piece. This addition was made to make collision computation between pieces much more efficient – see Section 6.4 for more details on this. To add these bounding circles, the piece characterisation tool was extended, allowing to draw a circle on each piece by providing its centre and its radius. Bounding circles were then manually determined for each piece, making sure that each circle encloses all its piece’s OBBs. An example of a piece’s bounding circle is show in.

Engineering drawing, pie chart

Description automatically generated

Figure 6: E piece, with OBBs shown in blue, Bezier control points in green and red, and bounding circle in black, with its centre a black dot.

## Extension idea

To allow for the construction of special pieces like tunnels or bridges, the tool could be extended to make it possible to add OBBs of arbitrary dimensions. Such pieces are usually composed of a standard piece to which a decoration is added. For example, shows a tunnel piece, which is composed of a straight piece enclosed by a simple cuboid casing. This casing, being 10.7cm tall, is high enough so it would prevent the placement of pieces just above it – the difference in height between levels zero and one being 6.4cm [9], [6]. This piece could be simply characterised using two connectors, to which two parts are appended: one with a long rectangle at level zero, slightly larger than the ones used for standard straight pieces, and a second one, positioned at level one, with the same dimensions as the first.

A picture containing text, container, box

Description automatically generated

Figure 7: A tunnel piece [9].

## Summary of piece characteristics

Figure 8 shows the classes and attributes representing BRIO™ pieces.



Figure 8: UML diagram representing the way in which BRIO™ pieces are stored in the Piece.json file.

The Vertex class is simply a pair of two floats, which is a convenient representation for points and directions 2D space. Two piece-attributes present in the diagram have not yet been discussed. The *flipType* integer property tells what kind of algorithm is used to flip the piece around its axis – this is detailed in Section 6.3. The *maxLength* float given to each piece is the distance between the two connectors of that piece that are furthest away from each other. This property is essential to the heuristic presented in Section 6.7.

## Pieces determined

Using the piece characterisation tool, a set of twelve pieces were initially determined, to which the standard ascending piece and two switch pieces were later added. Currently, fifteen pieces are available for users to choose from (see Table 1). These pieces and their associated characteristics are stored in a single JSON file, accessible to different parts of the program.

Table 1: Currently available pieces.



# User interface

Once a library of pieces was starting to get completed, two important aspects of the project remained: making an algorithm to generated tracks, and creating an interface for users to pick sets of pieces and visualise automatically-generated tracks. The decision was taken to first implement a simple graphical interface to display BRIO™ pieces and tracks, so that it could later be used to debug the track-generating algorithm. This section details the design decisions made regarding the user interface.

## Tools

An early decision was made that this program should be web-based, for ease of access of anyone wishing to use it. This also allows the program to run on any kind of device, mobile phones and tablets included. The interface is rather simple, the purpose it being to allow users to pick a certain set of pieces, generate a circuit from these pieces, and visualise the generated track. From this, it was determined that no frontend framework was needed, and the choice was made of building a simple HTML and JavaScript website, with basic CSS styling.

The page is divided into two simple components: a set of inputs for users to select pieces and specify different generation options, and a display to visualise the generated tracks. Tracks are displayed on a HTML5 Canvas, using the JavaScript Canvas API. This choice was made for the familiarity of the author with this API, and it being perfectly suitable for a 2D application like this one.

## Representation of generated circuits

Tracks are obtained from the core generating algorithm in JSON format, specifying the position of the pieces’ OBBs and connectors. The option to display on screen images of real BRIO™ railway pieces at the right locations to visualise tracks was considered. For this to work however, a careful, time consuming work of texture mapping would have to be done for each one of the available pieces. For this reason, a different solution was opted for – the OBBs and connectors themselves were chosen to be displayed, instead of pictures of the real pieces. The initial display is shown in Figure 9.

A picture containing engineering drawing

Description automatically generated

Figure 9: Initial display of a generated track on a HTML5 Canvas. The Bezier curve of each piece was made visible at this stage, but later removed.

Pieces were later colour-coded by their ID, and a simple shape was drawn at the location of each pin connector. These improvements facilitate the visualisation of the placement of pieces and make it easier for users to reproduce circuits with their real BRIO™ pieces. Figure 10 shows an example of track displayed on screen after these improvements were made.

Shape, arrow

Description automatically generated

Figure 10: Example of a track displayed on screen, with the addition of connectors and a colour coding for pieces.

Once multi-level tracks made their appearance, another colour code was added to the display to allow users to visualise the level at which pieces were placed. Pieces are still coloured depending on their ID, but outlined in a colour specific to their level. An example of a multi-level track with colour-coded outlines is shown in Figure 11.

Chart, shape, arrow, sunburst chart

Description automatically generated

Figure 11: A three-level circuit, with pieces placed on level 1 outlined in black, level 2 in blue and level 3 in yellow.

## Preventing piece overlaps

The JavaScript Canvas API works so that any element added at a position of the canvas hides the elements previously at that position. From the core generating algorithm, the JavaScript frontend code obtains circuits as a list of piece objects, sorted in no particular order. The addition of ascending pieces leads to an issue in the display of circuits. The fact that some pieces were allowed to stand above others, and due to the random order in which the pieces were obtained, some level zero pieces were sometimes painted after higher level ones. This led to some strange piece overlaps in the track and made it difficult to understand which pieces truly lied above others (see Figure 12).

This problem was addressed by making use of a simple technique called the Painter’s algorithm (see Figure 13). Pieces are first separated into their component OBBs and connectors, and a single list is made to contain these elements. The list is then sorted in ascending order of levels. Finally, each component gets displayed on screen, in the order given by the list; this leads to the pieces appearing at the right level on canvas. To account for pieces’ connectors not getting displayed below their respective OBBs, the level of each connector is raised by 0.5 before the list is sorted, which leads to the correct drawing order for the entire track.

A picture containing chart

Description automatically generated

Figure 12: Example of bad display of a portion of multi-level track. Pieces outlined in black are at level zero, and a red outline is used for level one. The two pieces at level one, at the bottom-left of the image, should be painted above the ones at level zero.

Chart, radar chart

Description automatically generated

Figure 13: Same portion of track as the one displayed on Figure 11, with the addition of the Painter's algorithm, solving the issue of pieces overlapping each other in the wrong order.

## Addition of zoom and drag on canvas

The option for users to move on the canvas by a clicking and dragging action of the mouse, and the ability to zoom by scrolling the mouse wheel, to view circuits from closer or further away, was added relatively early in the project. This way of interacting with the canvas to visualise circuits was chosen for its simple and intuitive use. These actions were implemented using a very light JavaScript library [10].

It turns out that this implementation of drag and zoom on canvas works well when using a physical mouse. However, touchpads often prove too sensitive and make it hard to position the generated track at the right level of zoom. Also, the dragging and zooming effects do not seem to be working on mobile phones. For these reasons, given more time on the project, research on how to implement these features in a more satisfactory way would be needed.

## Piece picker

The tool to select sets of pieces was created as a vertical scrollable list, displaying the ID, description and an image of each piece, alongside a basic HTML input to select a number of the corresponding piece. Images of the pieces were obtained from [6]. This list is shown in Figure 14.

Table

Description automatically generated

Figure 14: Table for selecting pieces to generate a track from.

## Small improvements

Small improvements on the user interface were made towards the end of the project. These were added following Tom’s suggestions, after testing of the program to build a track with real BRIO™ pieces. Tom’s testing of the program is detailed in section … HERE EVALUATION.

One issue found by Tom was that when generating a track from a random seed, there was no way of knowing which seed actually was used by the program to generate the track. This could be frustrating for a user who obtained a nice track from a random seed and would like to recreate it later. For this reason, a message displaying the seed used for any track generated was added.

Another issue was that a track generated was lost on refreshing the webpage. Users had to manually re-specify their selection of pieces every time the page was refreshed – this could be annoying for users trying to build a real BRIO™ track following the one displayed on screen. To tackle this issue, the pieces used to generate the last track were made to be kept in local storage, allowing a user to very easily recreate the previous circuit after a page reload.

The current state of the user interface is shown in Figure 15.

Diagram

Description automatically generated with medium confidence

Figure 15: Current state of the user interface, with an example of circuit displayed.

*Part on assumptions:*

* *Vario system and out way to deal with it (all pieces are connected perfectly with each other apart from the first and last piece of a loop -> this is not accurate but, with sufficiently small validation conditions, should lead to buildable tracks) + appearance of an issue due to this assumption: pieces in between. And our way to deal with it too.*
* *Multi-level tracks: two things: first, the absence of support for tracks hanging in the air. Support was completely omitted, and this could lead to some tracks generated and unbuildable in real life. It is the user’s responsibility to come in and place supports at the right spaces and perhaps modify the circuit a little bit if needed. Also, option to build tracks at max lvl 1. This is given because in real life, due to having to use support for pieces placed above the ground level, higher pieces get exponentially harder to position. The option to build very high tracks is left to the user but it could be a good idea to stick with 0 and 1 level tracks.*
* *Generation time and user input: the program attempts to build the track regardless of the user input. This leads to trouble not for unbuildable single loop tracks, but rather for tracks with unbuildable 2+ loops. Could make sure there are enough pieces to close the 2nd+ loop in sanitisation?*
* *Way in which the users can interact with the canvas only works on laptop, and is easiest used with a physical mouse. Tracks can still be generated on mobile devices but hardly visualisable. (Addition of buttons to move the canvas?)*

*Part on how WASM was set up. Then memory issues with it, needed to allow it for expansion of memory. Flags used, etc.*

# Track Generation

As discussed in the introduction, the core track-generating algorithm is implemented in a C++ program that is separated from the user-interface code. The algorithm works by randomly picking pieces and connecting them to each other until a closed loop is found. This section details the method used for generating BRIO™ track layouts.

## Tools

Java and C++ were both initially considered for the implementation of the track-generating algorithm. Both are object-oriented languages, and both offer options for multithreading, which seemed important at the start of the project. However, potential hindrance on performance caused by Java’s garbage collector, and more importantly, C++’s potential for getting compiled to WebAssemlby made the scales tip in favour of C++.

Initially, the decision was made to run the track-generating algorithm on a backend server, and to setup communication between frontend and backend using a NodeJS API. This worked fine for a while. However, this setup was later changed in favour of a fully static website, with the track-generating C++ program compiled to WebAssembly and running directly on the client’s machine. This decision was taken as it seemed more adequate to make the expensive track-generating program run on the user’s machine instead of a server that could get overloaded.

Meson [11] is used as a build system to compile the C++ project, and make it easier to include the JsonCpp [12] dependency that is used to parse the *Pieces.json* file described in Section 4.7.

## Algorithm description

The generation of track layouts is implemented in a recursive method, inspired by the depth-first search algorithm. At the start, one piece is placed at the centre of an infinite 2D area. The algorithm then selects a second piece and connects it to the first one. At each recursion, a piece gets placed at the end of the track, until either the track is fully generated, or no more pieces can be placed. When this happens, the function recurses back to the previous step, picking a different piece to place.

Each individual BRIO™ piece picked by the user is represented as a C++ object. The classes used in the C++ backend are detailed in the UML diagram shown in figure … . The set of all pieces is kept in a single C++ vector, with a flag on each piece telling whether the piece is in use or not.

For a single loop circuit, the generating algorithm works as follows: (CHANGE FOR PSEUDO CODE)

1. The first piece is positioned in the centre of the area. Its first connector is kept in memory as the *Open Connector*, and its second connector is taken as the *Validation Connector*. The Open Connector is the connector to which the next piece will link itself, and the Validation Connector is the one that needs to be reached for the track to be considered as closed.
2. The available pieces are shuffled into a random order.
3. The first available piece (that hasn’t been checked already) is taken into consideration. If it has a connector that is of the opposite type to the Open Connector, the pieces are linked. Otherwise, back to step 3. Then, it is checked that the newly placed piece is not colliding with any other placed piece. If the piece collides, back to step 3.
4. The newly placed piece is marked as used, and its available connector now takes the place of the previous Open Connector. Then, back to step 3.
5. Every time a new piece is placed, the *Validation Conditions* are tested for that piece. Validation conditions are discussed in Section 6.4.
6. When all the validation conditions are met, the track generation stops and the circuit is presented to the user.
7. If none of the pieces succeed in step 3, back to step 2.

This algorithm is implemented in a recursive way. This choice was made because it naturally works well: every time a new piece is placed, the track generation function can be called recursively with a new set of available and placed pieces, and a new Open Connector. Also, due to the fact that a maximum of one recursive call is made per available piece, a stack overflow should not occur from this recursive function as a maximum of only *N* stack frames are ever required for a set of *N* initial pieces.

## Piece positioning

An important part of the track-generating algorithm consists in connecting two pieces together. The requirements for connecting two pieces together are as follows: one of the pieces must already be placed and have an open connector, that is, a connector that is not currently used in a connection with another piece. The other piece must not be placed, and it must have a connector of the opposite kind to the open connector.

The objects containing information on a piece’s position in space are its OBBs, connectors and bounding circle. Each OBB contains a set of four vertices, one per corner. Each connector contains one vertex for its position, and one 2D vector for the direction in which it is pointing. A 2D vector class – named Vec2D – is used for representing positions and directions.

To connect the two pieces, the second piece is rotated around its connector until the two pieces’ connectors align and are in opposite directions. The piece then gets translated to a position where the two connectors touch each other.

Vector translations and rotations are necessary to change the position of pieces. Translations are trivially done by adding or subtracting vectors together. To rotate a piece, each one of its OBBs and connector’s position vertices are rotated using the Vec2D rotation function. This translates the vector’s coordinates by the opposite of the rotation point’s coordinates. Then, the vector is multiplied by the following rotation matrix:

Finally, the vector gets translated by the rotation point’s coordinates, completing the rotation around that point.

Another important aspect of piece positioning is the ability to flip pieces. Most BRIO™ pieces can be positioned facing up or down. However, straight pieces are naturally symmetric with respect to flipping. Each piece is thus given a *flipType* attribute corresponding to the way in which it must be flipped. A *flipType* of 0 means that the piece does not need to be flipped – this corresponds to straight pieces. A *flipType* equal to 1 is used for simple curved pieces that can be flipped by a simple inversion of its two connectors. This also works for the straight ascending piece. A *flipType* of 2 is given to complex pieces like switches, which require each one of their vertices individually reflected around an axis.

The ability to flip a piece if implemented in a method of the Piece class. This is shown in the following pseudo-code:

* if (*flipType* == 0): return; // No flip required.
* if (*flipType* == 1) { // Exchanging the two connectors is sufficient.
  + exchangeConnectors(*connectors*[0], *connectors*[1]);
* }
* if (*flipType* == 2) { // A full reflection is needed
  + for (Connector **con** : *connectors*) {
    - **con**.*position*.reflect();
    - **con**.*direction*.reflect();
  + }
  + for (Part **part** : *parts*) {
    - for (Obb **rect** : **part**.*obbs*) {
      * for (Vec2D **v** : **rect**.*vertices*) : **v**.reflect();
    - }
  + }
* }

Where reflect is a function of the Vec2D class – a class representing a simple 2D vector – that reflects the coordinates of the vertex around the x axis. This is done simply by leaving the y-coordinate unchanged and changing the sign of the x-coordinate. Note that the axis of reflection is arbitrary as long as it is the same for all the vertices of the piece.

This use of a *flipType* property for pieces is very impactful performance-wise. Thanks to it, straight pieces are known by the program to not require any flipping, preventing certain track-generation branches from being searched twice for no reason. **MEASURE PERFORMANCE GAIN??**. However the performance gained by making simple curves flip by exchanging their connectors, rather than reflecting each of their vertices around an axis, has not been measured and could be quite small.

## Validation Conditions

Validation Conditions are the conditions for the circuit to be considered closed. These conditions are checked every time a new piece is placed on the board. At any point in the generation of a track, if these Validation Conditions are verified, the track generation is considered to be successful and the circuit is returned to the user.

Validation Conditions are verified between the connectors at each end of the track, and include the following tests:

* the two connectors need to be close enough to each other.
* the two connectors need to have their directions align with each other, within a certain margin of error.
* a minimum proportion of the initial number of available pieces need to be placed.

Validation conditions are implemented inside a validation function.

## Collisions

After connecting a piece to the end of the partly generated track, the algorithm checks for collisions between the newly placed piece and all the already placed pieces. This is essential for preventing pieces from overlapping each other.

Due to the way in which pieces are represented using OBBs, slight overlaps between connected curved pieces are inevitable – see Figure 14. For this reason, a piece’s direct neighbour is ignored when checking for collisions. This assumption works well for the fifteen pieces made available in this version of the programme – two pieces taken from this set can never directly collide with each other. Although it is possible to imagine two theoretical pieces with unusual shapes that would collide when connected to each other, after inspection on the BRIO™ website, it appears that any two pieces can always directly connect without colliding.

A close-up of a cell phone

Description automatically generated with low confidence

Figure 16: Connected pieces slightly overlapping. The overlap is shown as a red-striped triangle.

To check for collisions between two pieces, each of the pieces’ OBBs are checked against each other. The collision between two OBBs is calculated using the Separating Axis Theorem. This theorem states that if a line can be drawn between two polygons without intersecting with either one, then the two polygons do not collide.

It is then possible to compute the collision between any two convex polygons in the following way. Start by taking the projection of all the corners of each of the polygons, along one of the first polygon’s side. Then, take the maximum and minimum points of each polygon along that axis, and check if their maximums intersect with each other – if they don’t, the two polygons do not collide. And if the projections of their corners do overlap for all of the polygons’ sides, they collide. Figure 15 shows an example of collision computation for two rectangles using this method.

A picture containing text, accessory, umbrella

Description automatically generated

Figure 17: Collision verification for two rectangles. The maximum and minimum of the projections of the rectangles' corners overlap along axis A2, but do not overlap along axis A1. This means that the rectangles do not collide.

Collisions are computed at each piece placement attempt, and against all of the pieces currently in use. This leads to a lot of collision computations, meaning that the efficiency of the collision verification function is critical to the overall efficiency of the programme.

The collision verification method described above is applied for OBBs in a slightly more efficient way. Indeed, rectangles have a pair of parallel sides, and the overlap of corners along two parallel axes is the same – this saves from computing overlaps along half of the rectangles’ sides. Also, the maximum and minimum of the projections of a rectangle’s corners along one of its own sides are simply the corners of that side – which saves from checking the two other corners.

To make collision computations even faster, a bounding circle was assigned to each piece. A bounding circle is a circle that encompasses all the piece’s OBBs within its area. Bounding circles were determined manually in the piece characterisation tool detailed in Part 4.

A collision between two circles requires a comparison of the square of the sum of their radiuses to the square of their distance – which is easily determined using Pythagoras’ theorem. This necessitates only a few operations, making collision computations for circles very efficient.

The collisions between two pieces are finally verified using the following algorithm:

1. Check collision between the pieces’ bounding circles. If these do not collide, the pieces do not collide.
2. If the bounding circles collide, check collisions between each one of the pieces’ OBBs. If any of these collide, the pieces collide, otherwise they don’t.

An estimate of the performance gained thanks to the addition of bounding circles is calculated using the data displayed in Figure 16. This data was obtained by timing the generation of a circuit with a range of seeds. Due to the way in which seeds are implemented, the addition of bounding circles, while changing the collision computation time, does not affect the way in which the track is generated for a specific seed. This allows for easy comparison between the track generation with and without the addition of bounding circles.

From this data, one can see that the track generation using bounding circles to help with collision computations was more performant that the one without bounding circles, for every single seed tested. For these input pieces and for seeds ranging from 1 to 10, the addition of bounding circles decreased the circuit generation completion time by a factor of , which is a very substantial performance gain.



Figure 18: Comparison of circuit generation completion time with and without bounding circles, for the input pieces: 100E, 30A, 100E1, 2L, 2M (close validation conditions)

## Avoiding repetitive computations

Once a basic version of the algorithm was implemented, some simple circuits could be generated. However, the algorithm would find circuits only for a very limited number of input pieces. Before making progress on getting multi-level and multi-loop tracks to generate, some improvements needed to be made regarding performance.

Most sets of BRIO™ pieces contain multiples of the same pieces. For example, the “medium straight” or “large curve” pieces are very common and often picked multiple times. When a collision is detected, the next available piece is tried instead – however, when that piece has the same id as the previously tested one, it is useless to attempt its placement, as the two pieces have the same geometry. To avoid repeating the same work multiple times, a set is created and made to contain the ids of pieces that were unsuccessfully tested. Before attempting to place any piece, it is verified whether its id is present in that set: if it is, the piece is skipped.

To get an idea of the impact this has on performance, one can consider the following formulas. The number of arrangements of *N* pieces, ignoring collisions, is the factorial of *N*. However, taking into account that some pieces are present *M* times in the set reduces the number of possible arrangements by the factorial of *M*.

For example, the number of arrangements of a set of 20 pieces is . However, if half of these pieces are known to be of the E type, and the other half of the A type, the number of arrangements is drastically reduced: . This is nine orders of magnitude smaller than the previous number. This simple example shows that this improvement avoids enormous amounts of useless computations.

## Further improvements on performance

The improvements detailed in Section 6.6 accelerated the program and allowed larger sets of input pieces to be picked. However the time taken to generate a track with a certain set of input pieces was unpredictable and highly variable depending on the seed chosen. For the same set of pieces, one seed could successfully generate a track in a few tenth of a seconds, while the program would not complete in less than 15 minutes for another seed.

The reason for this is that sometimes a piece is placed in such a way that the generation is impossible to succeed. For instance, one piece could be positioned in front of the Validation Connector and block it. Or a piece could be placed too far from the Validation Connector, making it impossible for any of the following pieces to reach it.

While these issues could be addressed individually thanks to the use of heuristics, a different solution was found. Instead of starting a generation and waiting until it completes, the algorithm was modified to attempt generation of the pieces in a particular order only for a certain number of recursions. When this number is reached, the generation is stopped, all the pieces are taken off the board, and the generation restarts with a different initial order.

It turns out that interrupting a restarting the generation after a small number of recursions, and attempting many generations quickly renders the program quite efficient for most initial sets of pieces chosen.

Measurements on the efficiency gained thanks to this addition is detailed in Section … (EVALUATION HERE).

## Multi-level tracks

According to [7], there exists two kinds of ascending BRIO™ pieces: the *N* piece, which essentially is an ascending version of the *D* piece, and the *N1* piece, a version of the *N* piece with two pin-type connectors instead of a pin and a hole. Only the *N* piece was added to the set of available pieces. This addition makes it possible for users to build multi-level tracks.

Only a small number of modifications were required for multi-level track generation to work. The main modifications regarded the collision-computation, the piece-connection and the track-validation functions. These changes are the following:

* Collisions were made to be ignored when two OBBs are positioned on different levels.
* To connect a piece to the rest of the track, all of the piece’s components are first raised or lowered to the level of the track’s open connector. A piece can be raised to any level, however, it is important to check that none of its components are lower than the ground floor.
* The validation function, detailed in Section 6.4, is slightly modified: a verification checking that the two end connectors are located at the same height is added.

Also, the addition of ascending pieces brought a new kind of performance issue, concerning certain kinds of sets of input pieces. When an odd number of ascending pieces is picked, the ends of the track will always be positioned on different levels, making it impossible to create a closed loop. To counter this, any odd ascending piece in a set of pieces was made to be removed before track-generation starts.

## Multi-loop tracks

Once single-loop tracks could generate in decent amounts of time, and multi-level tracks were implemented, the focus moved to multi-loop circuits. A divide-and-conquer approach was used to tackle this problem.

The track-generating algorithm was modified substantially to include multi-loop tracks. An outline of the algorithm for generating multi-loop circuits is as follows:

* Function generateMultipleLoops(Piece[] selection) {
  + // Remove odd ascending piece and odd 3-connector piece
  + clean(selection)
  + int threeConPieces = countThreeConPieces(selection)
  + // Shuffle all the pieces together
  + shuffle(selection)
  + int numberLoops = 1 + threeConPieces / 2
  + // Locate indices of pieces available for the first loop
  + int indexStart = 1
  + int indexEnd = selection.length / numberLoops
  + // Place the first piece
  + selection[0].place()
  + for (int i = 0; i < numberLoops; i++) {
    - // Ensure that there are an even number of ascending and of 3-connector pieces in the set
    - sanitise(selection, indexStart, indexEnd)
    - // Generate the loop
    - generateLoop(selection, indexStart, indexEnd)
    - // Position the indices for the next loop
    - indexStart = indexEnd
    - indexEnd = indexEnd + selection.length / numberLoops
  + }
* }

Note that this is a simplified version of the actual algorithm. For instance, the way to deal with the unsuccessful generation of a loop is not specified here. If a certain loop fails to get generated after a certain number of attempts, the generation falls back to the previous loop. After too many failed attempts at generating the first loop, the program is stopped altogether.

The algorithm starts by determining the number of loops *N* that can be built for the given set of pieces – this is equal to the number of pairs of 3-connector pieces available, plus one for the initial loop. Then, the set of pieces is partitioned in *N* subsets of equal sizes. It is ensured that exactly one pair of 3-connector pieces are located in each subset, apart from the last one. Then, a loop is generated for each subset in order, with the two 3-connector pieces positioned in a loop providing Validation Conditions for the next loop.

However the pieces are dealt with in the vector “in place” (explain here).

* + Importance of 3 connector pieces
  + Divide and conquer method
  + Know how to generate single loop circuits. At this point, only 2 connectors are considered when placing 3 connector pieces
  + The idea is the following: generate a loop in the usual way, repeating generation attempts with different shufflings of the initial set of pieces until a loop is found.
  + 1)Sanitise. Keep only 2 3con pieces per sets of pieces.
  + 2)generate a loop with the first set of pieces, making sure the 3con pieces are positioned
  + 3) Attempt N generations for the next loop, taking the two open connectors from the previous loop as the new validation connectors
  + 4) If the N generations work, go to the next loop, etc.
  + If they do not work, go back to the previous loop, reshuffle the pieces and start again.

Add picture of multi loop track

## Optimisations (discuss heuristic here).

Without the help of optimisations, the generation of certain tracks can be very slow.

* Addition of a check for same pieces
* Addition of reset after a certain number of generation attempts
* Addition of bounding circles (already discussed above)
* A heuristic for stopping the generation when a piece is placed too far away

## Possible improvements

One of the main drawbacks of the algorithm currently used to generate BRIO™ tracks is the computation of collisions. While efforts were made to make collision computations faster (see Section 6.4), the fact that collisions with all the placed pieces need to be checked before a new piece can be positioned has a strong impact on performance.

A way in which this issue could be addressed would be by imposing a restriction on the size of the room, instead of allowing pieces to be positioned anywhere in an infinite 2D space. The space in which pieces can be positioned could be a rectangle of fixed size, and the space inside this rectangle could be partitioned into a grid of small squares. Each piece would be assigned the cell on which it is positioned. Then, when a new piece arrives on the board, collisions would be checked not with all the pieces, but only with those lying on nearby cells. Of course, certain edge cases would need to be considered, one of which would be that certain pieces could lie on intersections between cells, but this could be relatively easily addressed. This improvement would bring down the complexity of calculating collisions with previously placed pieces: if there are *N* pieces on the board, the complexity of calculating collisions with the current method is O(*N*), but could be lowered to O(1) using the described approach.

# Evaluation

## Performance tests

# Conclusion

# Running the programme

Need to explain the choice of validation conditions – close, medium, large.

# References

|  |  |
| --- | --- |
| [1] | “Brio (company),” [Online]. Available: https://en.wikipedia.org/wiki/Brio\_(company). [Accessed 18 March 2023]. |
| [2] | “Minecraft,” [Online]. Available: https://en.wikipedia.org/wiki/Minecraft. [Accessed 18 March 2023]. |
| [3] | “No Man's Sky,” [Online]. Available: https://en.wikipedia.org/wiki/No\_Man%27s\_Sky. [Accessed 18 March 2023]. |
| [4] | J. T. M. J. N. Noor Shaker, “Chapter 2: The search-based approach,” in *Procedural Content Generation in Games*, Springer, 2016, pp. 20-22. |
| [5] | J. T. e. al., “Towards automatic personalised content creation for racing games,” *IEEE Symposium on Computational Intelligence and Games,* pp. 252-259, 2007. |
| [6] | “BRIO Track Guide,” [Online]. Available: https://woodenrailway.info/track/brio-track-guide. [Accessed 28 January 2023]. |
| [7] | “2-D Layout Modeling: SketchUp,” [Online]. Available: http://woodenrailway.info. [Accessed 28 January 2023]. |
| [8] | A. Riškus, “Approximation of a cubic Bezier curve by circular arcs and vice versa,” *ISSN 1392 – 124X, Information Technology and Control, Vol. 35, No.4,* 2006. |
| [9] | BRIO, “Tunnel,” [Online]. Available: https://www.brio.co.uk/en-GB/products/brio-world/buildings-tunnels-bridges/tunnel-63373500. [Accessed 30 January 2023]. |
| [10] | A. Shpakovsky, “Javascript canvas drag-and-zoom library,” [Online]. Available: http://alexey.shpakovsky.ru/en/javascript-canvas-drag-and-zoom-library.html. [Accessed 21 January 2023]. |
| [11] | “The Meson Build System,” [Online]. Available: https://mesonbuild.com/. [Accessed 20 March 2023]. |