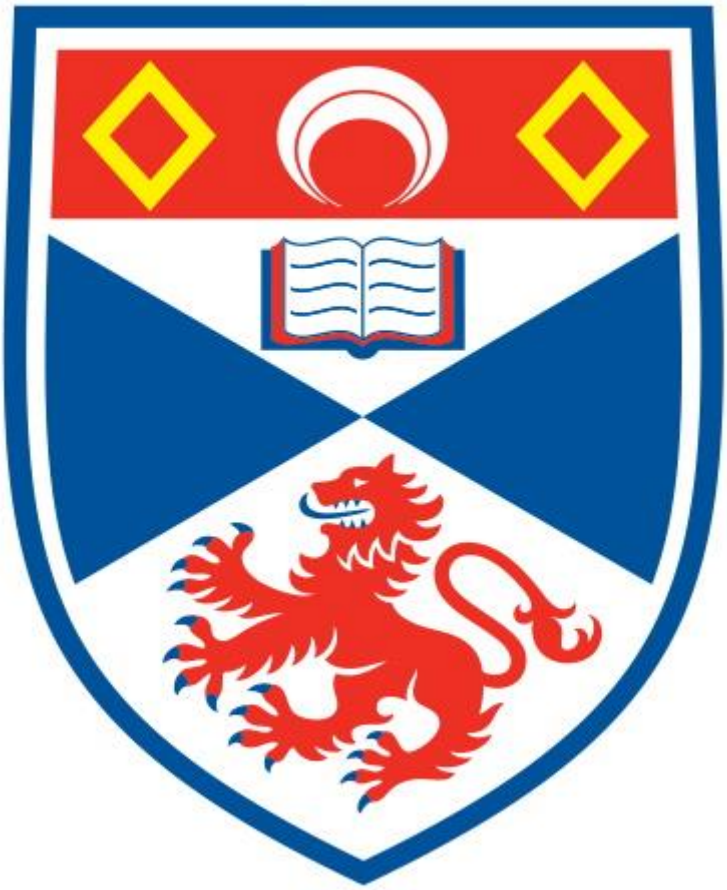

Generating BRIO™ layouts

Alexandre Kings

Supervisor: Tom Spink

Abstract

Declaration

1 Contents

2 Introduction 5
3 Requirements and design decisions 6
4 Characterization of the BRIO™ pieces 7
4.1 Required characteristics 7
5 User interface 13
5.1 Tools used 13
6 Track-Generation 17
6.1 Algorithm description 17
6.2 Piece positioning 18
6.3 Collisions 18
6.4 Ascending tracks 22
6.5 Multi-level tracks 22
6.6 Optimisations 22
7 Critical appraisal 23
8 Conclusion 24
9 Running the programme 25
10 References 26

2 Introduction

The goal of this project was to build a program for users to obtain automatically generated BRIO™ railway circuits from sets of input pieces. Users are able to use an interface to make a selection of BRIO™ pieces, generate a track layout from this selection, and visualise the circuit generated on a 2D display. The train tracks should be realistic, meaning that pieces should be connected in a way that is feasible in real life. The display should be intuitive and provide the users with enough information to be able to re-build the railway tracks with their own pieces at home.

3 Requirements and design decisions

The circuits generated should be non-trivial – they should contain at least one closed loop. The decision was made to completely prevent the generation of non-closed tracks. Such railway tracks are considered easy to construct and not worth any special interest.

The option to display generated layouts in 3D was considered at the start of the project. This would make it especially useful when it comes for multi-level circuits, which would allow users to directly see where the ascending pieces are, and 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.

A colour code is used as a workaround to display multi-level train tracks – pieces at a certain level are outlined in the colour specific to that level.

4 Characterization of the BRIO™ pieces

This section discusses the way in which a library of BRIO™ pieces was built, to allow for the generation of circuits and display of these on screen.

4.1 Required characteristics

For a program to generate track layouts, the very first step was to get a library of pieces and their characteristics. This library should give the circuit-generating program the following information on each railway piece:

- the piece's position,
- the orientation of a piece,
- whether or not two pieces could connect to each other,
- connect two pieces together,
- detect collision between pieces,

Also, more complex pieces than straight lines and simple curves exist, including switches and ascending pieces. From this initial list, a few important properties appeared. [include UML diagram for pieces] Each piece should have a list of connectors, each determined by a position, a direction and a type (male or female). This allows to connect pieces together, by rotating and translating the pieces so that the two connectors align and are at the same position.

Also, the general shape of each piece had to be captured in a way simple enough to allow fast and efficient collision calculations. The first option considered was outline each piece by a single polygon. This idea was not kept, however. This is because the Separating Axis Theorem (SAT) which will be used to determine collisions only works for convex shapes, and curved pieces and switches are convex shapes. A set of multiple cleverly determined polygons could have been used for these pieces, but this idea was not kept, for it seemed complicated to find the right set of polygons to use for some pieces, and a set of complex polygons for each piece would likely negatively impact the performance of collision-detection calculations. It was especially important that collisions could be calculated fast, considering that, to place one railway piece, the collision of it with every single one of the previously placed pieces had to 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 their distance squared. However, this option was not kept, for using circles to approximate rectangular shapes would be unnatural, and a rather large number of circles would then need to be used to approximate each shape, which would end up negatively impacting performance.

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 (BRIO Track Guide, 2023). These are relatively small curves and are well approximated using a small set of rectangles.

A tool was then developed to determine these characteristics for a set of initial pieces. It was decided to build a simple interface 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 would 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.

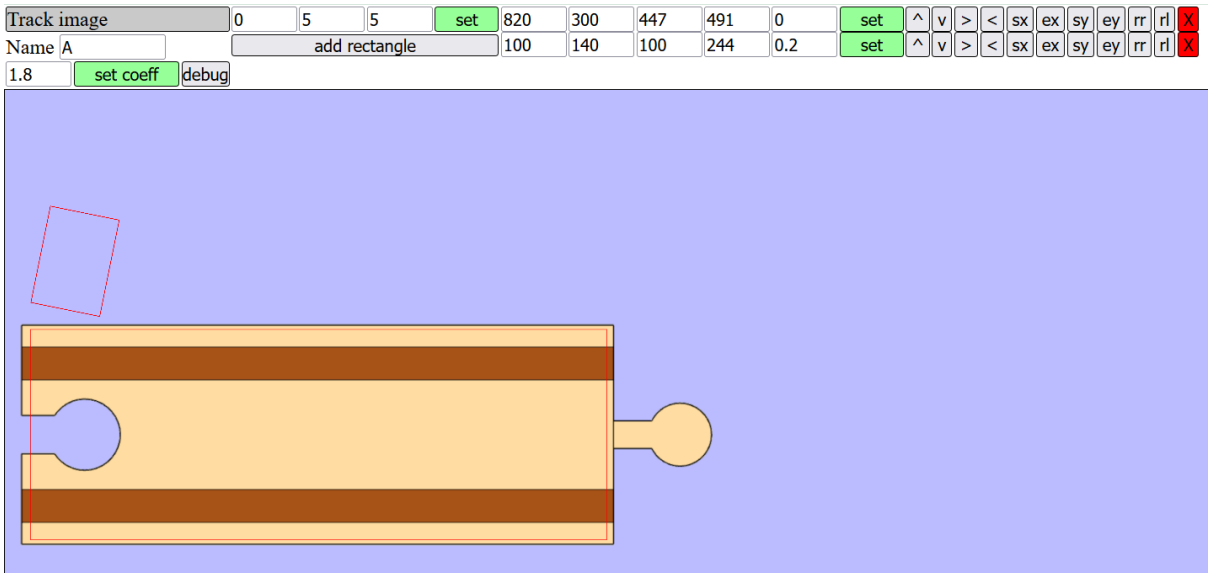

Figure 1: Tool initially developed for the determination of piece characteristics. Example of rectangles (in red) manually positioned, with an image of the BRIO™ A track in the background.

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 well placed for simple rectangular pieces, it was difficult to position them properly for more complex, curved pieces. Secondly, the direction of connectors for curved pieces, being manually positioned, would not be perfectly aligned, and this would result in strange-looking, imperfect generated tracks. The last, and possibly 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 (2-D Layout Modeling: SketchUp, 2023). Images of the pieces were exported as PNG files from SketchUp, using these resources. This way in which 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. This version of the tool was soon abandoned.

The idea for the next version of this tool was to make the determination of the characteristics of the pieces purely geometrical and avoid using images of the pieces as a basis to determining the characteristics of those. Not only does a purely geometrical approach lead to perfectly aligned pieces, it also naturally addresses the scale problem. The dimension of pieces was obtained from the BRIO™ Wooden Railway Guide (BRIO Track Guide, 2023). Each JavaScript Canvas unit is made to correspond to a millimetre of a real piece. Rectangular pieces are easily determined by two points only, one on each of the connectors of the piece. From these points, a single OBB is automatically generated, surrounding the bulk of the piece, and the two connectors are positioned to be aligned with the single axis of the piece. The tool also allows to determine the type of each connector, male or female (called “out” and “in”

respectively). Figure 2 shows an example of determination of the characteristics of a rectangular piece using the tool described.

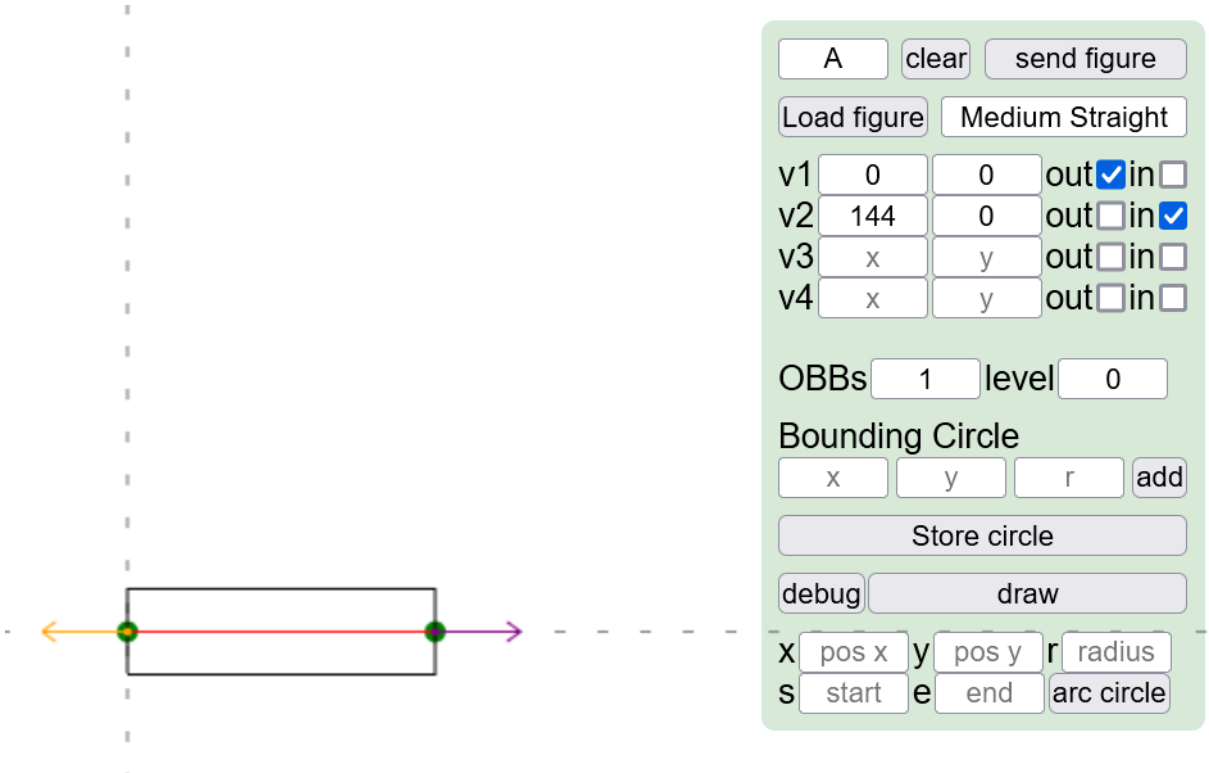

Figure 2: Determination of the characteristics of the A piece, using version 2 of the tool. HTML inputs, with minimal CSS styling, are used to position the different components of each piece.

Curved pieces are determined with the same tool, using a set of up to four 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 generated 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.

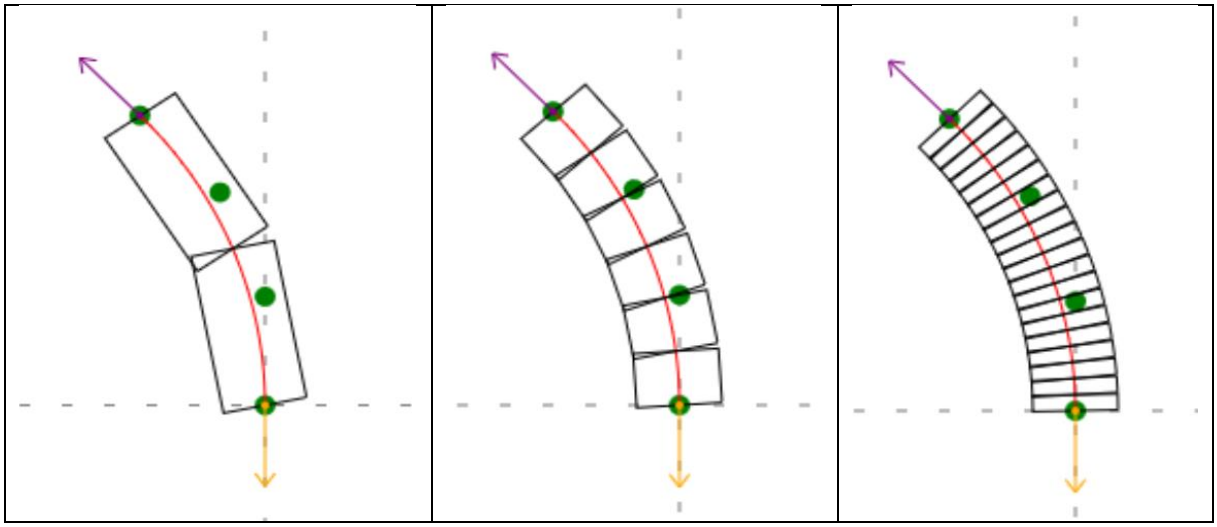

Figure 3: Determination of the E piece using the Bezier curve plotting tool, with 2, 6 and 20 OBBs. The green points represent the four Bezier control points, and the arrows show to position and direction of connectors.

The use of Bezier curves makes it easy to determine simple curved pieces using the arc circle approximation formulae. For an arc of angle β starting at point $P_1 = (x_1, y_1)$ and ending at point $P_4 = (x_4, y_4)$, the coordinates of the two Bezier control points are given by:

$$P_2 = (x_1 + k * r * sin(\beta/2), y_1 - k * r * cos(\beta/2))$$

$$P_3 = (x_4 + k * r * sin(\beta/2), y_4 + k * r * cos(\beta/2))$$

Using $k = 0.55191496$, an approximation of an arc circle with an error of $1.96 * 10^{-4}$ in the radius is obtained, which is perfectly acceptable in this case (Riškus, 2006). Figure 4 shows an example of a Bezier curve obtained using these formulae.

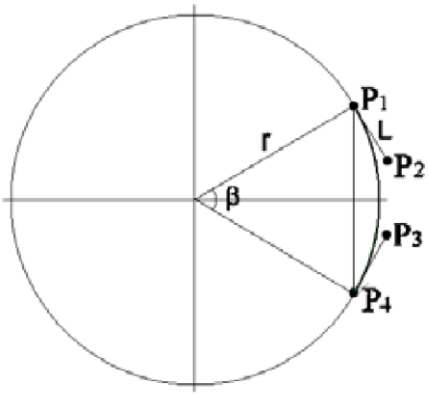

Figure 4: Approximation of an arc circle using a Bezier curve (Riškus, 2006).

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.

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.

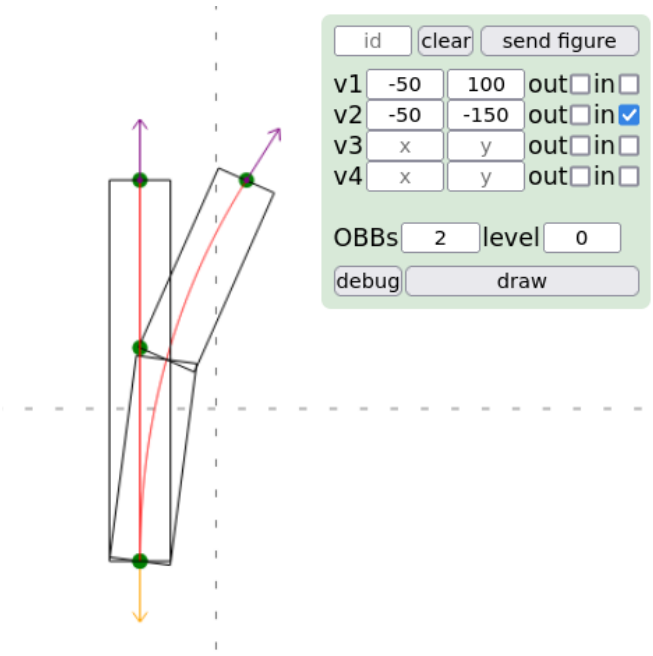

Figure 5: Determination of the characteristics of the L piece using the Bezier curve plotting tool. Bezier control points are shown in green, and the arrows show the position and direction of connectors.

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.

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, Figure 6 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 level 1 pieces just above it – the difference in height between levels zero and one being 6.4cm (BRIO, 2023), (BRIO Track Guide, 2023). 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.

Figure 6: A tunnel piece (BRIO, 2023).

An initial set of twelve pieces were initially determined, to which the standard ascending piece and two switch pieces were added, later in the project. Currently, fifteen pieces are available for users to choose from (see Figure 7). These pieces and their associated characteristics are stored in a single JSON file, accessible to different parts of the program.

Piece Type	ID
Straight	A, A1, A2, B, B1, B2, C, C1, C2, D
Curved	E, E1
Switch	L, M
Ascending	N

Figure 7: Table of currently available pieces.

- Addition of bounding circles + determination of the circles.

5 User interface

5.1 Tools used

An early decision was made that this program should be web-based, for ease of access for anyone wishing to make use of it. This also allows the program to run on any kind of device, mobile phones and tablets included. Making a complex user interface was not part of the plan for this project, 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.

It was decided that the generation of circuits would be made by a separate program dedicated to this task. Initially, the decision was made to communicate between the frontend user interface and this circuit-generating programme via a Node.js backend. This setup was later changed to a fully static website, with the track-generating algorithm compiled to WebAssembly and running directly on the client's machine.

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 2D applications like this one.

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 8.

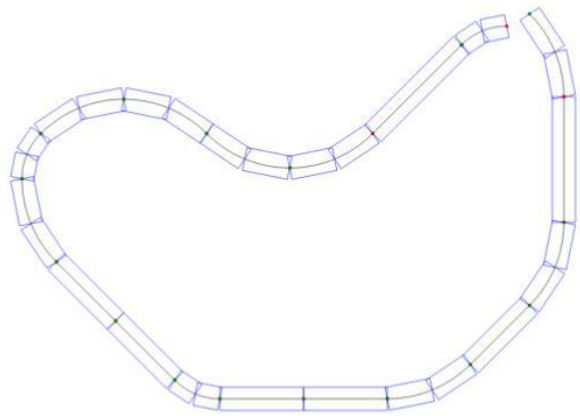

Figure 8: Initial display of a generated track on a HTML5 Canvas. The Bezier curve of each piece was made visible at this stage, but later removed.

Pieces were later colour-coded by their ID, and a simple shape was drawn at the location of each male 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 9 shows an example of track displayed on screen after these improvements were made.

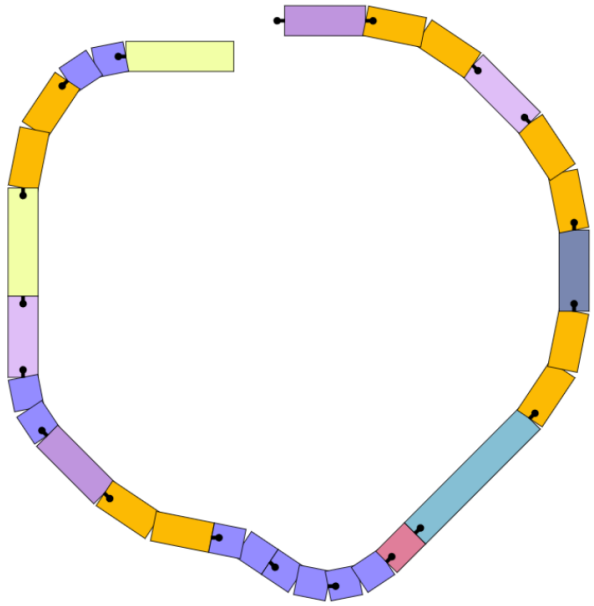

Figure 9: Example of a track displayed on screen, with the addition of connectors and a colour coding for pieces.

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 10.

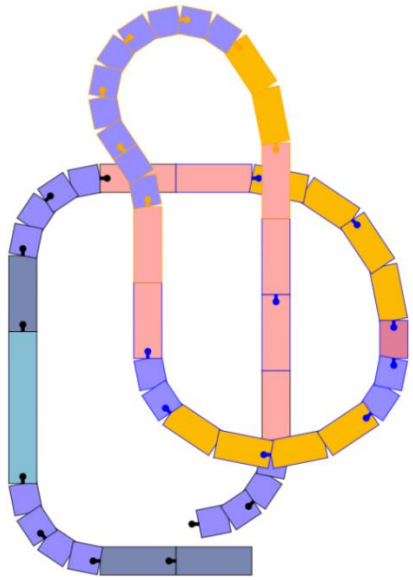

Figure 10: A three-level circuit, with pieces placed on level 1 outlined in black, level 2 in blue and level 3 in yellow.

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 lead 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, some level zero pieces were sometimes painted after higher level ones. This led to some strange clippings in the track and made it difficult to understand which pieces truly lied above others (see Figure 11). This problem was addressed by making use of a simple technique called the Painter's algorithm (see Figure 12). Pieces are first separated into their component OBBs and connectors, which are placed in a single list. The list is then sorted in ascending order of levels. Finally, each component is in order displayed on screen; 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.

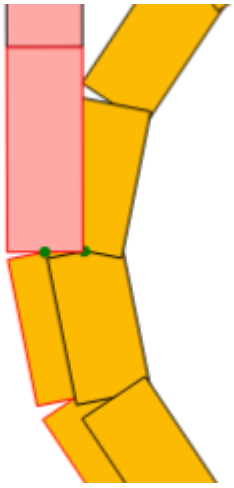

Figure 11: Example of bad display of a portion of multi-level track. Pieces outlined in black are at level zero, and a red outline is used for level one. The two pieces at level one, t the bottom of the image, should be painted above the ones at level zero.

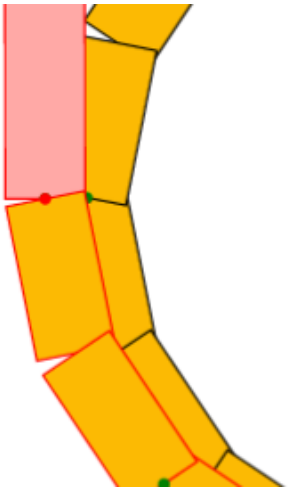

Figure 12: Same portion of track as the one displayed on Figure 11, with the addition of the Painter's algorithm, solving clipping issues.

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 library (Shpakovsky, 2023).

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 2^{nd}+ 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?)*

- *Painting of elements in order*
- *Colour code for elements at different levels*
- *Sorting of elements to pain first, to paint in order*
- *Explain that due to the way in which the circuits are generated, pieces are obtained in a list in random order.*
- *Painter's algorithm for drawing the elements in order on canvas.*

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

- *EXPLAIN VARIO SYSTEM.*

6 Track-Generation

This section details the method for generating BRIO™ track layouts.

6.1 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:

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 the conditions for the circuit to be considered closed: the available connector of the last placed piece must be close enough to the Validation Connector, and their directions need to align, with a certain margin of error. Another Validation Condition is that a certain proportion of the pieces chosen by the user need to be placed – this is to avoid obtaining very small, trivial circuits.

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.

6.2 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:

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

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

6.3 Collisions

After connecting a piece to the end of the generated track, the algorithm checks for collisions between the newly placed piece and all the other pieces.

Due to the way in which pieces are represented with OBBs, slight overlaps between connected curved pieces are inevitable – see Figure 13. This is why the direct neighbour to a piece 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 collide. 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 all the real pieces, including ones with special decors, are able to connect properly with no direct collisions.

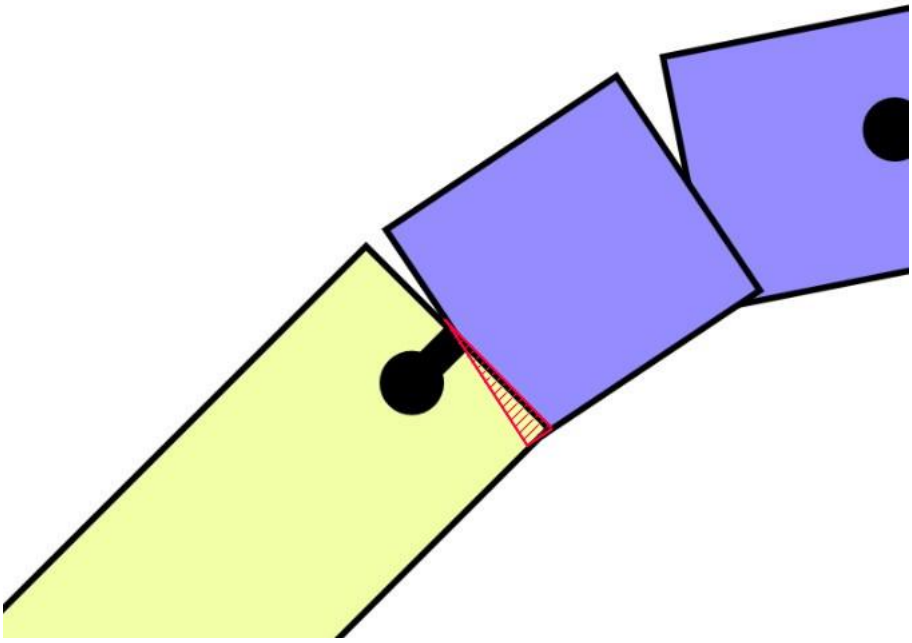

Figure 13: Connected pieces slightly overlapping. The overlap is shown as a red-striped triangle.

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 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 14 shows an example of collision computation for two rectangles using this method.

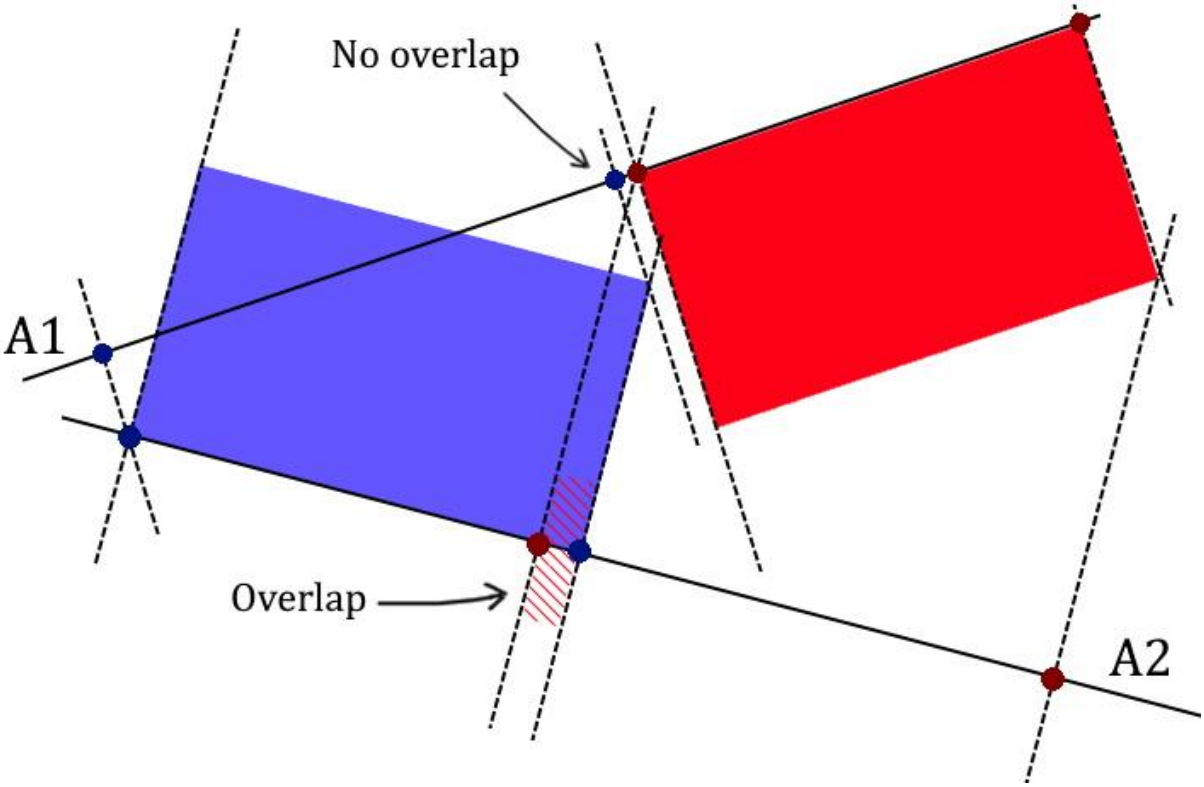

Figure 14: Collision verification for two rectangles. The maximum and minimum of the projections of the rectangles' corners overlap along axis A2, but do not overlap along axis A1. This means that the rectangles do not collide.

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 introduced for 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. An example of a piece's bounding circle is show in Figure 15.

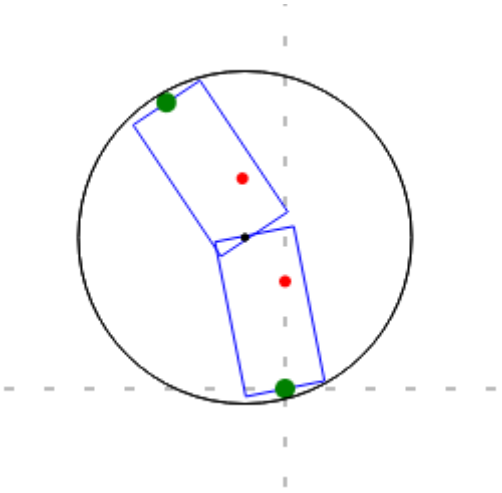

Figure 15: E piece, with OBBs shown in blue, Bezier control points in green and red, and bounding circle in black, with its centre a black dot.

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 $143{,}706/21{,}523.8 = 6.6766 \approx 7$, which is a very substantial performance gain.

Seed	1	2	3	4	5	6	7	8	9	10	**Average completion time** (in ms)
Completion time (in ms, with bounding circles)	3928	38536	32998	25983	4232	35312	5764	7901	50573	10011	21523.8
Completion time (in ms, with out bounding circles)	28253	250336	186734	177445	35221	235735	43350	52043	342449	85494	143706

Figure 16: Comparison of circuit generation completion time with and without bounding circles, for the input pieces: 100E, 30A, 100E1, 2L, 2M (close validation conditions)

6.4 Ascending tracks

The only ascending piece available for users is the N piece.

6.5 Multi-level tracks

6.6 Optimisations

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
-

7 Critical appraisal

8 Conclusion

9 Running the programme

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

10 References

2-D Layout Modeling: SketchUp. (2023, January 28). Retrieved from BRIO Wooden Railway Guide: http://woodenrailway.info

BRIO. (2023, January 30). *Tunnel*. Retrieved from BRIO: https://www.brio.co.uk/en-GB/products/brio-world/buildings-tunnels-bridges/tunnel-63373500

BRIO Track Guide. (2023, January 28). Retrieved from BRIO Wooden Railway Guide: https://woodenrailway.info/track/brio-track-guide

Riškus, A. (2006). Approximation of a cubic Bezier curve by circular arcs and vice versa. *ISSN 1392 – 124X, Information Technology and Control, Vol. 35, No.4*.